Algorytmika dla początkujących

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf".
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Algorytmika dla początkujących

Jacek Krzaczkowski

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Jacek Krzaczkowski
ALGORYTMIKA DLA POCZĄTKUJĄCYCH

Recenzent: Grzegorz Matecki

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedziba w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-36-7

Spis treści

Przedmowa **vii**

1 Na dobry początek **1**
1.1. Zanim zaczniemy implementować algorytmy 2
1.2. Elementy teorii złożoności 6

2 Struktury danych, abstrakcyjne typy danych, listy **11**
2.1. Abstrakcyjne typy danych 12
2.2. Listy 13
2.3. Proste algorytmy sortujące 35
2.4. Kolejka, stos, kolejka priorytetowa 48

3 Przeszukiwanie wyczerpujące **57**
3.1. Przeszukiwanie wszystkich możliwości 58
3.2. Przeszukiwanie z nawrotami 72

4 Metoda dziel i zwyciężaj **77**
4.1. O metodzie 78
4.2. Zmniejsz i zwyciężaj 90

5 Programowanie dynamiczne **97**
5.1. O metodzie 98
5.2. Metoda spamiętywania 116

6 Algorytmy zachłanne **121**
6.1. O metodzie 122
6.2. Przykłady 122

7 Drzewa **131**
7.1. Definicja drzew i sposoby ich reprezentacji 132
7.2. Przechodzenie drzew 135
7.3. Kopce, sortowanie przez kopcowanie 138

8 Grafy **151**

8.1. Definicja i sposoby reprezentacji grafu 152
8.2. Przechodzenie grafów 156

BIBLIOGRAFIA **161**

Przedmowa

Algorytmika i programowanie to podstawy informatyki. Bez nich trudno mówić o korzystaniu z komputerów i o informatyce w ogóle. Nie na darmo David Harel nadał swojej znanej książce tytuł „Rzecz o istocie informatyki. Algorytmika” [3]. Biorąc to pod uwagę zaskakująco często programowanie, a szczególnie implementacja mniej lub bardziej zaawansowanych algorytmów, sprawia dużą trudność wielu studentom informatyki. Najpopularniejsze podręczniki do algorytmiki zakładają, że ich czytelnikom programowanie nie sprawia problemu. Stąd, dla studentów, którzy nie nauczyli się jeszcze dobrze programować, a już muszą próbować implementować proste algorytmy, podręczniki te często nie są zbyt użyteczne. W odróżnieniu od wspomnianych podręczników, książka, którą Czytelniku trzymasz w ręku, próbuje uczyć podstaw algorytmiki i programowania jednocześnie. Na ile jest to próba udana, ocenisz sam.

Niniejszy skrypt nie jest klasycznym wykładem z algorytmiki. Jest on raczej pomyślany jako uzupełnienie podstawowego wykładu z algorytmiki przeznaczone przede wszystkim dla tych studentów, którzy nie radzą sobie z implementowaniem prostych algorytmów. Bardziej zaawansowanych czytelników mogą nużyć długie opisy kolejnych kroków implementacji prostych funkcji, ale mają one pomóc początkującemu czytelnikowi zrozumieć nie tylko ideę algorytmu, ale także to, jak go zaimplementować. Nieprzypadkowy jest także dobór materiału do książki. Skupiono się na najprostszych strukturach danych i podstawowych metodach konstruowania algorytmów. Wszystko to z myślą o czytelnikach, którzy, aby móc implementować bardziej skomplikowane algorytmy, muszą jeszcze popracować nad swoimi umiejętnościami programistycznymi. Stąd rozdział o listach i prostych algorytmach na nich jest znacznie dłuższy od rozdziałów o drzewach czy grafach.

Programowanie to przede wszystkim praktyka. Nie da się nauczyć programować wyłącznie czytając książki. Tak samo pisanie programów i niekompilowanie ich albo kompilowanie i nieuruchamianie nie pozwoli nauczyć się programowania. Na końcu poszczególnych rozdziałów czytelnik znajdzie proste zadania dotyczące omawianych zagadnień. Warto spróbować je zaimplementować i przetestować ich działanie. Ponadto wszystkie algorytmy przedstawione w książce zostały zaimplementowane w języku C++ w taki sposób, aby nadawały się do przekopiowania i użycia po minimalnych zmianach w programach czytelników. Warto spróbować samodzielnie zaimplementować omawiane algorytmy, a w razie problemów porównywać swój kod z tym zamieszczonym w książce.

Czytelnik, który zechce poszerzyć w swoją wiedzę w zakresie algorytmiki, może sięgnąć po którąś z pozycji wymienionych w bibliografii. Szczególnie godne polecenia są dostępne bezpłatnie w internecie kursy na stronie Studiów Informatycznych [7]. Ci, którzy wolą obcować z papierową książką, przede wszystkim mogą skorzystać z [2], a także z [1] i [4]. Jeżeli kogoś zainte-

resuje generowanie obiektów kombinatorycznych prezentowane w rozdziale 3.1, może sięgnąć po [5], zaś zainteresowani zgłębianiem teorii złożoności po [6].

Rozdział 1

Na dobry początek

1.1. Zanim zaczniemy implementować algorytmy **2**
1.2. Elementy teorii złożoności **6**

1.1. Zanim zaczniemy implementować algorytmy

Doświadczenie pokazuje, że wielu osobom, które jako tako opanowały podstawy swojego pierwszego języka programowania, problem sprawia przejście od pisania krótkich, kilku- lub kilkunastolinijkowych programów, do implementacji prostych algorytmów. Niedoświadczeni programiści mają problem zarówno z zaprojektowaniem szczegółów implementacji, jak i z zapanowaniem nad kodem programu, który nie mieści się w standardowym oknie edytora. Poniżej zostały przedstawione proste porady, których zastosowanie może oszczędzić wielu problemów. Znacznie łatwiej pisze się programy niż szuka w nich błędów. Warto więc pisać w taki sposób, żeby zminimalizować ryzyko popełnienia błędu, a w przypadku jego popełnienia, ułatwić jego odnalezienie.

Przemyślana implementacja Zanim przystąpi się do pisania programu, warto zastanowić się nad tym, co i jak chce się napisać. Nawet implementując stosunkowo krótkie algorytmy warto przemyśleć ich implementację, aby przyspieszyć samo kodowanie i uniknąć błędów w programie. Początkujący programiści stając przed zadaniem programistycznym często próbują je rozwiązywać pisząc programy metodą drobnych losowych zmian. Piszą kawałek programu i patrzą, co on robi. Jeżeli program nie działa, tak jak powinien, to losowo zmieniają lub dopisują niewielkie kawałki kodu i patrzą, jak zmieniło się działanie programu. Tak otrzymany ciąg programów w przypadku prostych zadań bywa zbieżny do poprawnego rozwiązania, jednak w przypadku bardziej skomplikowanych programów taka metoda zupełnie się nie sprawdza.

Podział programu na funkcje Wielu uczących się programować cały kod programu umieszcza w jednym miejscu (na przykład w przypadku programów napisanych w C++ w funkcji `main`). Powoduje to, że trudno jest zapanować nad kodem i równie trudno wyszukać w nim ewentualne błędy. Dzieląc kod programu na funkcje nie tylko umożliwiamy wielokrotne wykorzystanie w programie raz napisanego kodu, ale także ułatwiamy sobie analizę programu. Przy dobrze napisanym programie każdą funkcję możemy testować niezależnie, co znacznie ułatwia nam szukanie błędów.

Sensowne nazwy zmiennych i funkcji Stosowanie jedno-, dwuliterowych nazw zmiennych, czy równie enigmatycznych nazw funkcji jest powszechną praktyką wśród uczących się programować. Oczywiście nie ma nic złego w tym, że tak jak większość programistów całkowitoliczbową zmienną sterującą w pętli `for` nazwiemy „`i`”, o ile stale tak robimy. Nie ma bowiem w takim przypadku ryzyka pomylenia tej zmiennej z inną. Jednak gdy mamy kilka zmiennych, których nazwy nic nam nie mówią, to przy dłuższym programie bardzo łatwo będzie nam te zmienne

pomylić (dotyczy to szczególnie rzadziej używanych zmiennych). Dlatego warto używać nazw zmiennych i funkcji, które mówią nam, co pod nimi się kryje.

Przejrzysty kod Poza efektywnością programu powinniśmy dbać także o jego czytelność. To, że jakaś konstrukcja jest dopuszczalna w używanym przez nas języku programowania, nie oznacza, że musimy jej używać. Nie musimy walczyć o to, żeby nasz kod zajmował jak najmniej miejsca. Ważne, żebyśmy patrząc na jakiś fragment kodu wiedzieli, co tam się dzieje. Dzięki temu łatwiej się pisze program i szybciej szuka w nim ewentualnych błędów. Jak zwiększyć przejrzystość kodu? Na przykład:

— używając, tam gdzie jest to możliwe, instrukcji `switch` zamiast długich serii instrukcji `if else`,

— unikając zbytniego „kompresowania” kodu poprzez robienie wielu rzeczy na raz. Przykładem tego, jak nie należy programować jest chociażby następujący warunek w języku C:
`++i<(j=f(x))`,

— programując strukturalnie i proceduralnie,

— robiąc wcięcia w kodzie i oddzielając bloki instrukcji pustymi liniami.

Komentarze Komentarze ułatwiają zorientowanie się w kodzie długich programów. Gdy nasz program zawiera kilkadziesiąt lub choćby kilkanaście funkcji, łatwo się pogubić, co która z nich robi. Pisanie komentarzy może nam znacznie ułatwić pracę. Komentarze okazują się także niezwykle przydatne, gdy chcemy wrócić do pisania programu po chociażby dwutygodniowej przerwie. Niezależnie od tego jak długo i intensywnie nie pracowalibyśmy nad programem, to gdy wrócimy do niego po dwóch tygodniach, będziemy musieli włożyć sporo wysiłku, aby znowu go zrozumieć. Komentarze znacznie ułatwiają przypomnienie sobie o co chodzi w programie.

Kompilacja bez ostrzeżeń Pomimo tego, że wiele ostrzeżeń zwracanych przez kompilatory języków C i C++ może być zupełnie niegroźnych (tak jak na przykład ostrzeżenie o braku znaku końca linii na końcu programu zwracane przez kompilator `gcc`), to warto programować tak, żeby w trakcie kompilacji nie otrzymywać żadnego ostrzeżenia. Po pierwsze dlatego, że w gąszczu „niegroźnych” ostrzeżeń można nie zauważyć jakiegoś ważnego ostrzeżenia (na przykład, że napisaliśmy funkcję, która jako wartość zwraca wskaźnik do swojej zmiennej lokalnej). Po drugie, ponieważ prawie nie ma „niegroźnych” ostrzeżeń, większość ostrzeżeń może świadczyć o tym, że program zawiera błąd. Po trzecie zaś, ponieważ początkującemu programiście często trudno ocenić czy dane ostrzeżenie można zignorować czy nie.

Zdrowy rozsądek Stosując się do powyższych zasad należy przede wszyst-

kim pamiętać o zdrowym rozsądku. To, że zaleca się unikanie instrukcji `goto` nie oznacza, że w żadnej sytuacji nie możemy jej użyć. Czasami, gdy chcemy wyskoczyć z wielokrotnie w sobie zagnieżdżonych skomplikowanych pętli, użycie instrukcji `goto` mniej zakłóci czytelność kodu i jest obarczone mniejszym ryzykiem popełnienia błędu niż takie zorganizowanie kodu, żeby instrukcji `goto` nie trzeba było użyć.

Nie mniej ważna od umiejętności napisania programu jest umiejętność jego przetestowania oraz zlokalizowania w kodzie ewentualnych błędów. Często jest tak, że testowanie, lokalizowanie i poprawianie błędów w programie zajmuje więcej czasu niż wcześniej jego napisanie.

Testowanie służy wykryciu ewentualnych popełnionych przez nas błędów. W trakcie testowania poza kilkoma typowymi przypadkami warto sprawdzić także warunki brzegowe, czyli działanie programu dla „skrajnych" danych. Na przykład gdy nasz program operuje na liście, należałoby sprawdzić między innymi, jak zachowuje się, gdy lista jest pusta lub ma tylko jeden element. Wskazane jest też sprawdzenie tego, jak program działa dla dużych danych. Najlepiej jest wygenerować dane, o których wiemy, że mogą sprawić problem naszemu programowi. Jeżeli nie potrafimy wygenerować takich danych, to zwykle nieźle w takiej sytuacji sprawdzają się losowe zestawy danych.

Jeżeli wykryjemy sytuację, w której nasz program działa niepoprawnie, to, aby móc go poprawić, musimy wykryć miejsce, w którym popełniliśmy błąd. Niezależnie od tego, czy błędem jest niepoprawny algorytm, pomyłka w implementacji poprawnego algorytmu czy zwykła literówka, najprościej jest go zlokalizować śledząc działanie programu dla feralnych danych. W prostych przypadkach najszybciej możemy to zrobić, każąc programowi wypisywać w kluczowych momentach działania informacje o swoim stanie: na przykład wartości najważniejszych w danym momencie zmiennych lub informację o tym, że doszliśmy do danej linii kodu. W bardziej skomplikowanych przypadkach możemy posłużyć się debuggerem (z ang. dosłownie odpluskwiaczem, od *bug* – błąd, robak, pluskwa), czyli aplikacją wspomagającą śledzenie działania innych programów. Debuggery umożliwiają m.in. wykonywanie programu krok po kroku i jednoczesne śledzenie wartości wybranych zmiennych. Używając debuggerów należy pamiętać, że programy uruchomione w trybie debugowania mogą zachowywać się nieco inaczej niż normalnie: na przykład program, którego działanie normalnie dla pewnych danych testowych przerywa błąd wykonania w trakcie debugowania, dla tych samych danych, może wykonać się do końca. Debuggery są standardową częścią zintegrowanych środowisk programistycznych. Istnieją także debuggery uruchamiane z linii komend. Szukanie błędów, nawet przy użyciu

debuggera, bywa zajęciem żmudnym i czasochłonnym. Często łatwiej jest napisać program od początku niż znaleźć błąd w już istniejącym.

Na zakończenie tego podrozdziału przeanalizujemy kilka częstych błędów w programach:

Przepełnienie pamięci Występuje, gdy program próbuje zaalokować więcej pamięci niż jest dostępne. Przepełnienie pamięci może, lecz nie musi, być przyczyną przerwania działania programu (w zależności od sposobu alokacji pamięci). Szczególnie trudny do wykrycia jest przypadek, gdy przekroczenie pamięci występuje w wyniku próby utworzenia zbyt dużej globalnej tablicy automatycznej. Wtedy działanie programu jest przerywane na samym jego starcie.

Wyciek pamięci Jest to sytuacja, gdy program „zapomina" wskaźników do dynamicznie zaalokowanych obszarów pamięci. W dłuższej perspektywie wyciek pamięci może doprowadzić do przepełnienia pamięci. Wyciek pamięci jest często występującym i trudnym do wykrycia błędem. Aby zapobiec wyciekom pamięci w wielu językach programowania takich m.in. jak Java czy Python, wprowadzono odśmiecanie pamięci, czyli automatyczne usuwanie obiektów, na które nie wskazują żadne przechowywane w programie zmienne.

Użycie niepoprawnego wskaźnika Jest kilka typowych przyczyn użycia niepoprawnego wskaźnika do pamięci:
- użycie niepoprawnych indeksów przy próbie dostępu do elementów tablicy,
- nieumiejętne operowanie na elementach list wskaźnikowych, węzłach drzew itd., w wyniku którego program nie wstawia pustego wskaźnika zaznaczającego ostatni element listy, liść drzewa itd.
- używanie nieaktualnych wskaźników, czyli wskaźników do obiektów usuniętych już z pamięci.

W zależności od tego, na jaki obszar pamięci wskazuje używany niepoprawny wskaźnik i czy chcemy tylko przeczytać zawartość wskazywanego obszaru czy też chcemy tam coś zapisać, działanie programu może, ale nie musi, zostać przerwane. Jest to wyjątkowo nieprzyjemny rodzaj błędu, gdyż bywa trudny do wyśledzenia i może powodować „nieracjonalne" zachowanie programu. Na przykład w wyniku użycia niepoprawnego wskaźnika możemy zmienić wartość zupełnie przypadkowej zmiennej. Innym skutkiem takiego błędu może być to, że działanie programu zmieni się po dopisaniu teoretycznie nieistotnej instrukcji (na przykład `p=p;`).

Podanie niepoprawnych argumentów funkcji Przykładowo dzielenie przez 0 lub pominięcie `&` w argumentach funkcji `scanf` może skończyć się błędem wykonywania programu.

Wartości niemieszczące się w zmiennych Należy pamiętać, że zmienne danego typu mogą przyjmować wartości wyłącznie z określonego zbio-

ru. Wystarczy, że w jednym miejscu spróbujemy zapisać do zmiennej wartości spoza zakresu jej dopuszczalnych wartości, a wynik długiego ciągu obliczeń może stać się bezwartościowy.

Trudność w szukaniu błędów polega między innymi na tym, że mogą się one znajdować w zupełnie innych miejscach kodu niż te, w których wykryliśmy niepoprawne działanie programu. Na przykład dzielenie przez 0 może być efektem tego, że wcześniej, w wyniku użycia niepoprawnego wskaźnika, „przypadkowo" zmieniliśmy wartość dzielnika, albo że w wyniku użycia zbyt mało pojemnego typu jakiś ciąg obliczeń zwrócił niewłaściwy wynik.

1.2. Elementy teorii złożoności

Zanim przyjrzymy się pierwszym algorytmom, poznamy narzędzia, które pozwolą nam porównywać między sobą różne algorytmy i struktury danych. Takich narzędzi dostarcza nam teoria złożoności, która zajmuje się określaniem ilości zasobów potrzebnych do rozwiązywania problemów obliczeniowych [8]. W tym rozdziale czytelnik znajdzie minimum informacji z zakresu teorii informacji, niezbędne aby ze zrozumieniem przeczytać kolejne rozdziały skryptu. Czytelnik, którego zainteresuje ta tematyka, znacznie więcej informacji znajdzie w [6].

Zazwyczaj mierzymy dwa rodzaje zasobów: czas oraz pamięć. Oczywiście zużycie pamięci oraz czas działania programu zależy od wyboru modelu obliczeń, czyli „maszyny", na której dokonujemy obliczeń. Zużycie zasobów zależy też od danych wejściowych. Przyjęło się, że złożoność algorytmu określamy jako funkcję rozmiaru wejścia, czyli funkcję, która dla podanego rozmiaru wejścia zwraca ilość potrzebnych zasobów. Ponieważ dla różnych danych o tym samym rozmiarze zapotrzebowanie algorytmu na zasoby może być bardzo różne, rozważamy następujące rodzaje złożoności:

— **złożoność pesymistyczna** – ilość zasobów potrzebna w najgorszym wypadku algorytmowi dla danych o rozmiarze n. Jest to najprostszy do policzenia i najczęściej rozpatrywany rodzaj złożoności obliczeniowej.
— **złożoność średnia** – wartość oczekiwana ilości potrzebnych algorytmowi zasobów dla danych o rozmiarze n. Aby móc policzyć złożoność średnią danego algorytmu, musimy założyć jakiś rozkład prawdopodobieństwa wystąpienia poszczególnych instancji rozwiązywanego problemu (czyli danych wejściowych algorytmu).
— **złożoność zamortyzowana** jest miarą złożoności operacji na strukturach danych. Złożoność zamortyzowana operacji to pesymistyczna złożoność ciągu operacji podzielona przez ilość operacji. Złożoność zamortyzowana jest liczona dla złożoności czasowej operacji.

Mierząc złożoność czasową jakiegoś algorytmu nie możemy mierzyć po prostu czasu działania danego algorytmu zaimplementowanego na jakimś konkretnym komputerze, gdyż w takim przypadku wyniki otrzymane na różnych komputerach byłyby ze sobą nieporównywalne. Lepszym pomysłem jest zliczanie liczby wykonanych instrukcji, jednak także w tym przypadku otrzymane wartości mogą się różnić w zależności od komputera, użytego języka programowania itd. W związku z powyższym zwykle szacuje się liczbę kroków wykonywanych przez ustaloną abstrakcyjną maszynę taką jak maszyna Turinga czy maszyna RAM. Aby jeszcze bardziej uniezależnić się od konkretnej implementacji algorytmu zamiast wszystkich instrukcji zliczamy zwykle tylko instrukcje dominujące, czyli takie, których liczba decyduje o złożoności algorytmu (ich liczba jest proporcjonalna do czasu działania programu). W zależności od algorytmu wybiera się różne operacje dominujące (przykładowo mogą być to operacje arytmetyczne czy porównywanie elementów listy). W niniejszym skrypcie będziemy zazwyczaj zliczać instrukcje dominujące lub liczbę wykonanych instrukcji pseudokodu.

Przez złożoność pamięciową rozumiemy ilość pamięci, jaką potrzebuje program dla danych wejściowych o określonym rozmiarze. Pamięć możemy liczyć na różne sposoby. Często wydziela się pamięć zajmowaną przez dane wejściowe oraz pamięć zajmowaną przez wynik i nie wlicza się jej do złożoności pamięciowej (robi się tak na przykład w przypadku liczenia złożoności pamięciowej na maszynie Turinga). My będziemy szacowali pamięć zajmowaną przez wszystkie struktury danych używane w programie.

Gdy szacujemy złożoność obliczeniową, zwykle interesuje nas to, jak algorytm zachowuje się dla dużych danych. Do tego nie interesuje nas zazwyczaj dokładna złożoność algorytmu, ale rząd wielkości złożoności. W związku z powyższym do szacowania od góry złożoności algorytmów używamy notacji O (czyt. „O duże”), która upraszcza szacowania pozwalając pominąć w trakcie obliczeń stałe i wolniej rosnące składniki wzoru.

Definicja 1.1. *Mówimy, że funkcja* $g : \mathbb{N} \to \mathbb{N}$ *jest* $O(f(x))$
jeżeli istnieją a, x_0,
takie że dla dowolnego $x \geqslant x_0$ *zachodzi* $g(x) \leqslant a \cdot f(x)$

Równoważnie możemy zdefiniować notację O w następujący sposób:

Definicja 1.2. *Mówimy, że funkcja* $g : \mathbb{N} \to \mathbb{N}$ *jest* $O(f(x))$
jeżeli istnieją a, b,
takie że dla dowolnego x *zachodzi* $g(x) \leqslant a \cdot f(x) + b$.

Przy szacowaniu dolnego ograniczenia złożoności algorytmów używamy notacji Ω analogicznej do notacji O:

Definicja 1.3. *Funkcja* $g : \mathbb{N} \to \mathbb{N}$ *jest* $\Omega(f(x))$
wtedy i tylko wtedy gdy
$f(x)$ *jest* $O(g(x))$.

Zauważmy, że jeżeli $f(n)$ jest $O(a * g(n) + b)$, gdzie a i b są dowolnymi stałymi, to $f(n)$ jest $O(g(n))$. Podobnie gdy $f(n)$ jest $O(n^a + n^b)$, gdzie a, b są dowolnymi stałymi takimi, że $a \leq b$ to $f(n)$ jest $O(n^b)$.

Teraz używając notacji O możemy zdefiniować kilka najpopularniejszych funkcji złożoności. Powiemy, że algorytm ma złożoność (czasową lub pamięciową):
— **logarytmiczną**, gdy ma złożoność $O(\log n)$,
— **liniową**, gdy ma złożoność $O(n)$,
— **wielomianową**, gdy ma złożoność $O(n^k)$ dla pewnego stałego k,
— **wykładniczą**, gdy ma złożoność $O(k^n)$ dla pewnego stałego k.

Przyjrzymy się teraz przykładowi, który pokaże, że nawet pisząc proste programy, warto zastanowić się nad ich złożonością obliczeniową. Zobaczymy dwie funkcje, które dla podanej w argumencie liczby całkowitej `n` zwracają jako wartość `n`-ty element ciągu Fibonacciego zadanego wzorem rekurencyjnym:

$$f_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f_{n-1} + f_{n-2} & n \geqslant 2 \end{cases} .$$

Zapewne 9 na 10 początkujących programistów zaimplementowałoby funkcję liczącą wyrazy ciągu Fibonacciego w następujący sposób:

Listing 1.1. Funkcja licząca n-ty wyraz ciągu Fibonacciego – wersja 1

```
1 unsigned int f1(unsigned int n){
    if (n==0)||(n==1) return n;
3   return f1(n-1)+f1(n-2);
  }
```

Prosta analiza powyższego kodu pokazuje, że złożoność czasowa funkcji `f1` jest wykładnicza względem wartości n. Jednocześnie łatwo pokazać, że przedstawiona poniżej funkcja `f2` działa w czasie $O(n)$

Listing 1.2. Funkcja licząca n-ty wyraz ciągu Fibonacciego – wersja 2

```
1 unsigned int f2(unsigned int n){
    int i,w1,w2,w3;
3   if (n<=1)
      return n;
5   w1=0;
    w2=1;
7   for(i=2;i<=n;i++){
      w3=w2+w1;
```

```
9         w1=w2;
          w2=w3;
11      }
        return w2;
13  }
```

Na koniec zauważmy, że funkcja `f2` działa w czasie wykładniczym względem rozmiaru wejścia. Dzieje się tak, gdyż rozmiarem wejścia jest tutaj $\lceil \log_2 n \rceil$, czyli liczba bitów potrzebnych do zapisu liczby n.

Rozdział 2

Struktury danych, abstrakcyjne typy danych, listy

2.1. Abstrakcyjne typy danych **12**
2.2. Listy **13**
    2.2.1. Listy wskaźnikowe 14
    2.2.2. Tablicowa implementacja listy 24
2.3. Proste algorytmy sortujące **35**
    2.3.1. Sortowanie przez wybieranie 35
    2.3.2. Sortowanie przez wstawianie 39
    2.3.3. Sortowanie bąbelkowe 44
2.4. Kolejka, stos, kolejka priorytetowa **48**
    2.4.1. Kolejka 48
    2.4.2. Stos 50
    2.4.3. Kolejka priorytetowa 51

2.1. Abstrakcyjne typy danych

Programując wielokrotnie spotykamy się z sytuacją, gdy o efektywności działania programu decyduje odpowiednia organizacja danych przechowywanych przez program. Sposób uporządkowania danych na komputerze nazywamy strukturą danych. Nie trzeba rozwiązywać skomplikowanych problemów obliczeniowych, aby wybór struktury danych miał istotne znaczenie dla efektywności naszego rozwiązania. Rozpatrzymy teraz przykładowy prosty problem, w którym wybór właściwej struktury danych ma podstawowe znaczenie.

Problem 2.1. *Przechowujemy dwuelementowe podzbiory liczb całkowitych z zakresu od* 0 *do* $n-1$. *Zakładamy, że każda z liczb od* 0 *do* $n-1$ *występuje w dokładnie jednym podzbiorze z listy. Mamy napisać funkcję, która dla podanej listy podzbiorów oraz liczby* $k \in \{0, \ldots, n-1\}$ *zwraca jako wartość liczbę, która należy do tego samego podzbioru z listy co* k.

Jednym z rozwiązań jakie możemy zastosować jest przechowywanie listy podzbiorów jako tablicy struktur.

Listing 2.1. Funkcja szukająca drugiego elementu podzbioru – wersja 1

```
  struct podzbior {
2   unsigned int elem1, elem2;
  };
4
  unsigned int drugi_elem1(struct podzbior tab_podz1[],
6                                          int n, int k){
    int i;
8   for(i=0;i<n;i++)
      if (tab_podz1[i].elem1== k)
10      return tab_podz1[i].elem2;
      else if (tab_podz1[i].elem2 == k)
12      return tab_podz1[i].elem1;
  }
```

Powyższe rozwiązanie, choć może się wydawać naturalne, nie jest optymalne ani pod względem złożoności czasowej, ani pod względem złożoności pamięciowej. Aby zwiększyć efektywność rozwiązania zamiast przechowywać listę podzbiorów, będziemy przechowywali tablicę, w której element o indeksie i będzie zawierał informację o liczbie, która należy do tego samego podzbioru co i. Tak zdefiniowana tablica przechowuje dokładnie te same informacje, co tablica `tab_podz1` z listingu 2.1, ale zajmuje o połowę mniej miejsca i umożliwia rozwiązanie problemu 2.1 przy pomocy funkcji działającej w cza-

sie stałym, podczas gdy pierwsze rozwiązanie wymagało w pesymistycznym przypadku przejrzenia całej tablicy par.

Listing 2.2. Funkcja szukająca drugiego elementu podzbioru – wersja 2

```
1 unsigned int drugi_elem2(unsigned int tab_podz2[], int k){
    return tab_podz2[k];
3 }
```

Ze względu na ich wpływ na efektywność algorytmów, struktury danych są jedną z intensywnie badanych dziedzin informatyki. Okazuje się, że w wielu algorytmach wykorzystuje się struktury danych o podobnych właściwościach. Dlatego definiuje się *abstrakcyjne typy danych* (inaczej abstrakcyjne struktury danych), które grupują podobne struktury danych. Abstrakcyjne typy danych definiujemy podając wymagane własności przechowywanych danych oraz operacje jakie możemy na nich wykonywać. Definiując abstrakcyjny typ danych nie określamy sposobu jego implementacji. Większość abstrakcyjnych typów danych możemy implementować na wiele sposobów, wśród których często nie ma jednego najlepszego we wszystkich zastosowaniach. Dzięki użyciu abstrakcyjnych typów danych możemy projektować algorytmy na wyższym poziomie abstrakcji bez wnikania w szczegóły implementacji struktur danych.

W kolejnych rozdziałach poznamy przykłady abstrakcyjnych typów danych oraz sposoby ich implementacji.

2.2. Listy

Lista jest abstrakcyjnym typem danych służącym do przechowywania skończonych ciągów elementów udostępniającym następujące operacje:
- `create` – tworzy pustą listę,
- `empty(List)` – zwraca wartość `true`, jeżeli lista `List` jest pusta i `false` w przeciwnym przypadku,
- `first(List)` – zwraca pozycję pierwszego elementu listy `List`,
- `front(List)` – zwraca pierwszy element listy `List`,
- `last(List)` – zwraca pozycję ostatniego elementu listy `List`,
- `back(List)` – zwraca ostatni element listy `List`,
- `push_front(List,x)` – wstawia element x na początek listy `List`,
- `push_back(List,x)` – wstawia element x na końcu listy `List`,
- `pop_front(List)` – usuwa pierwszy element listy `List`,
- `pop_back(List)` – usuwa ostatni element listy `List`,
- `insert(List,p,x)` – wstawia do listy List element `x` na pozycji `p`,
- `erase(List, p)` – usuwa z listy `List` element o pozycji `p`,

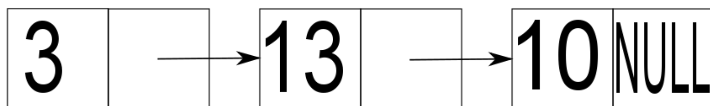
— `next(List,p)` – zwraca pozycję elementu następującego po elemencie o pozycji `p`,
— `at(List, p)` – zwraca referencję do elementu listy o podanej pozycji.

Lista jako abstrakcyjny typ danych nie ma jednego ustalonego i powszechnie uznawanego zbioru operacji. Przedstawiona powyżej lista operacji na liście stanowi wybór operacji pojawiających się w literaturze. W wielu funkcjach operujących na liście pojawia się pojęcie pozycji elementu. Co konkretnie oznacza to pojęcie, zależy od sposobu implementacji listy.

Istnieją dwa główne sposoby implementacji listy: tablicowa i wskaźnikowa. Oba te sposoby implementacji istnieją w wielu wersjach. Przedstawimy przykładowe sposoby implementacji listy liczb całkowitych.

2.2.1. Listy wskaźnikowe

Jako pierwszą pokażemy implementację przy pomocy jednokierunkowej listy wskaźnikowej. W jednokierunkowych listach wskaźnikowych każdy element listy przechowywany jest w oddzielnej strukturze, przechowującej poza nim także wskaźnik do struktury przechowującej następny element listy. Dla uproszczenia będziemy nazywać elementem listy całą strukturę przechowującą właściwy element listy. Ostatni element listy zamiast wskaźnika na następny element przechowuje wskaźnik pusty (NULL).



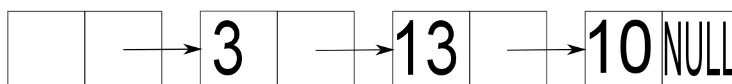
Rysunek 2.1. Przykład jednokierunkowej listy wskaźnikowej

Aby mieć dostęp do dowolnego elementu listy wystarczy pamiętać wskaźnik na pierwszy element (pierwszy element zawiera wskaźnik na drugi, drugi na trzeci itd.) i dlatego utożsamiamy listę ze wskaźnikiem na jej pierwszy element (tak jak w językach C i C++ utożsamiamy tablicę ze wskaźnikiem na jej pierwszy element). Oznacza to, że gdy mamy gdzieś podać jako argument listę, to podajemy wskaźnik na pierwszy element listy. W wielu operacjach na liście jako argument podajemy pozycję elementu listy. W przypadku jednokierunkowych list wskaźnikowych przez pozycję elementu `e` rozumiemy wskaźnik na element poprzedzający `e`. Taka definicja pozycji może wydawać się nieintuicyjna ale dzięki niej jesteśmy w stanie efektywnie zaimplementować usuwanie z listy elementu o podanej pozycji.

Pewnym problemem przy listach reprezentowanych przez wskaźnik na pierwszy element jest implementacja wstawiania i usuwania pierwszego elementu. Funkcje wykonujące te operacje muszą zwracać jako wartość

wskaźnik na pierwszy element uaktualnionej listy (problem ten nie występuje w przypadku implementacji listy jako klasy, gdyż wtedy funkcja modyfikująca początek listy może sama zaktualizować pole przechowujące wskaźnik na pierwszy element listy). Innym problemem jest pozycja pierwszego elementu, której nie da się podać zgodnie z naszą definicją. Jest tak, gdyż pierwszy element nie posiada poprzednika. Aby obejść ten problem, ustalamy pozycję pierwszego elementu jako jakąś ustaloną wartość np. `NULL`.

Rozwiązaniem problemów wspomnianych w poprzednim akapicie jest dodanie na początku listy sztucznego pustego elementu. Element taki, zwany głową listy, sprawia, że wskaźnik na listę się nie zmienia (jest nim zawsze wskaźnik na głowę) oraz nie mamy problemu z pozycją pierwszego elementu (jest nim wskaźnik na głowę). Poniższy rysunek przedstawia **trójelementową** jednokierunkową listę wskaźnikową z głową.



Rysunek 2.2. Przykład jednokierunkowej listy wskaźnikowej z głową

Przyjrzymy się teraz podstawowym operacjom na liście liczb całkowitych zaimplementowanej przy pomocy jednokierunkowej listy wskaźnikowej z głową. Na początku musimy zdefiniować typ elementu listy. Powinien on zawierać dane (w naszym przypadku liczbę całkowitą) oraz wskaźnik na następny element:

Listing 2.3. Typ elementu listy

```
1 struct element {
    int dane;
3   element * nastepny;
  };
```

Zaimplementujemy listę jako klasę, w której operacje na liście będą metodami klasy. W związku z tym w metodach implementujących poszczególne operacje nie musimy podawać listy jako argumentu. Wszystkie operacje będą wykonywane na liście przechowywanej w obiekcie, na rzecz którego zostanie wywołana dana metoda. Dodatkowo w obiektowej implementacji listy nie musimy implementować operacji `create`, jej rolę będzie spełniał konstruktor klasy.

Listing 2.4. Nagłówek klasy ListaWsk

```
  class ListaWsk {
2 private:
```

```
    element * glowa;
4 public:
    ListaWsk();
6   ~ListaWska();
    bool empty();
8   element* first();
    int front();
10  element * last();
    int back();
12  void push_front(int e);
    void push_back(int e);
14  void pop_front();
    void pop_back();
16  void insert(element * p, int e);
    void erase(element * e);
18  element * next(element * p);
    int& at(element * p)
20 };
```

Konstruktor tworzy tylko głowę listy i nadaje wartość `NULL` polu `nastepny` głowy.

Listing 2.5. Konstruktor klasy ListaWsk

```
  ListaWsk::ListaWsk(){
2   glowa = new element;
    glowa->nastepny = NULL;
4 }
```

Destruktor musi zwolnić pamięć po elementach listy. Kolejne elementy listy są usuwane w pętli. Używamy dwóch zmiennych `pom` i `glowa`, gdyż w momencie zwolnienia pamięci zajmowanej przez jakiś element nie możemy już odwołać się do jego pola `nastepny`, aby przejść do następnego elementu listy. Dlatego używamy zmiennej `pom`, która przechowuje wskaźnik na element następujący bezpośrednio po usuwanym elemencie

Listing 2.6. Destruktor klasy ListaWsk

```
  ListaWska::~ListaWsk(){
2   element * pom=glowa->nastepny;
    while (pom!=NULL){
4     delete glowa;
      glowa=pom;
6     pom=pom->nastepny;
    }
8   delete glowa;
  }
```

Sprawdzenie, czy lista jest pusta, wymaga jedynie sprawdzenia, czy pole `nastepnik` w głowie jest równe `NULL`:

Listing 2.7. Operacja empty

```
1 bool ListaWsk::empty(){
    return (glowa->nastepny == NULL);
3 }
```

Pozycja elementu listy wskaźnikowej to zgodnie z definicją wskaźnik do poprzedniego elementu, czyli pozycją pierwszego elementu jest wskaźnik do głowy:

Listing 2.8. Operacja first

```
1 element * ListaWsk::first(){
    return glowa;
3 }
```

Powyższa funkcja zwraca wskaźnik do głowy nawet dla pustej listy. Zachowanie się operacji `first` w takiej sytuacji jest kwestią umowną i dlatego wybraliśmy najprostsze rozwiązanie.

Zwrócenie wartości pierwszego elementu listy wymaga zwrócenia pola `dane` struktury znajdującej się na liście tuż za głową (wskazywanej przez pole `nastepny` głowy).

Listing 2.9. Operacja front

```
1 int ListaWsk::front(){
    return glowa->nastepny->dane;
3 }
```

Dla listy pustej powyższa funkcja nie zadziała (najczęściej zakończy działanie programu z błędem wykonania). Moglibyśmy wykrywać pustą listę i w takim przypadku rzucać wyjątkiem, ale nie robimy tego, gdyż niepotrzebnie skomplikowałoby to prostą funkcję. Oczywiście w komercyjnym programie powinniśmy jakoś obsłużyć sytuację, gdy lista jest pusta.

Metody z listingów 2.8 i 2.9, choć proste, dobrze ilustrują różnicę pomiędzy elementem listy a jego pozycją w liście wskaźnikowej. Jeżeli ktoś nie do końca rozumie tą różnicę, powinien dokładnie przeanalizować te metody.

Analogiczne do `first` i `front` operacje `last` i `back` są trudniejsze w implementacji, gdyż wymagają przejścia całej listy i odszukania jej ostatniego elementu. Moglibyśmy tego uniknąć pamiętając poza wskaźnikiem do głowy także wskaźnik do ostatniego elementu listy.

Listing 2.10. Operacje last i back

```
element* ListaWsk::last(){
  element * pom=glowa;
  while((pom->nastepny!=NULL)&&(pom->nastepny->nastepny!=NULL))
    pom = pom->nastepny;
  return pom;
}

int ListaWsk::back(){
  element * pom=glowa;
  while(pom->nastepny !=NULL)
    pom = pom->nastepny;
  return pom->dane;
}
```

Operacja `last` zwraca dla pustej listy wskaźnik do głowy. Podobnie jak w przypadku operacji `first` zachowanie się operacji `last` w tej sytuacji jest kwestią umowy. Podobnie jak nasza implementacja operacji `front`, także metoda `back` nie sprawdza, czy lista nie jest pusta. Zauważmy, że operację `back` można także zaimplementować z wykorzystaniem operacji `last`. Taka implementacja tej operacji jest znacznie prostsza..

Listing 2.11. Operacja back

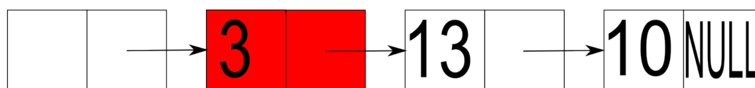
```
int ListaWsk::back(){
  return last()->next->dane;
}
```

Operacja `insert` wymaga tylko zaalokowania nowej struktury oraz przepięcia dwóch wskaźników:

Listing 2.12. Operacja insert

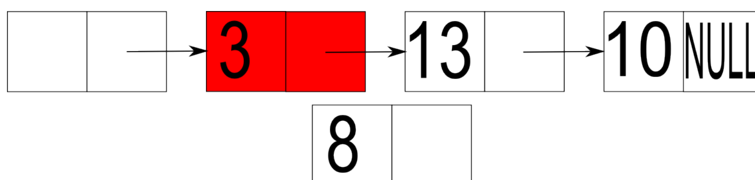
```
void ListaWsk::insert(element * poz, int e){
  element * pom = new element;
  pom->dane = e;
  pom->nastepny = poz->nastepny;
  poz->nastepny = pom;
}
```

Aby lepiej zrozumieć co się stało, przyjrzyjmy się rysunkom 2.3, 2.4, 2.5 i 2.6. Rysunek 2.2 zawiera przykładową listę z głową, w której będziemy wstawiali element o wartości `8` za elementem o wartości `3`. Aby to zrobić funkcji `insert` podajemy jako argument wskaźnik na element o wartości `3` i liczbę `8`:



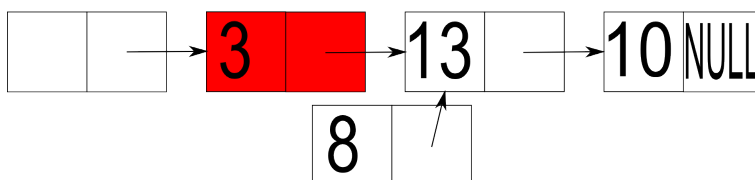
Rysunek 2.3. Operacja insert

Aby wstawić nowy element najpierw musimy go zaalokować w pamięci operatorem `new` i nadać polu `dane` wartość 8 (linie numer 2 i 3 w listingu 2.12).



Rysunek 2.4. Operacja insert

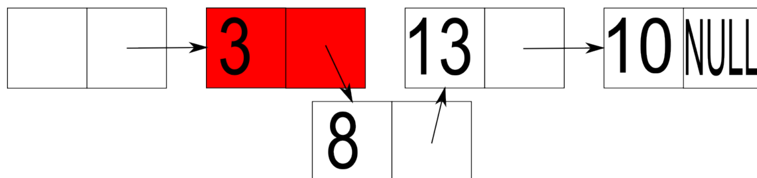
Następnie przypisujemy polu `nastepny` nowo utworzonej struktury wskaźnik na elementu następujący po elemencie o wartości `3` (linia numer 4 w listingu 2.12):



Rysunek 2.5. Operacja insert

Na koniec polu `nastepny` w strukturze przechowującej wartość `3` przypisujemy wskaźnik do dodawanego elementu (linia numer 5 w listingu 2.12, rysunek 2.6).

Przyczyną, dla której w listach jednokierunkowych pozycję elementu definiuje się jako wskaźnik do jego poprzednika, jest chęć efektywnego zaimplementowania operacji usuwania elementu. Aby usunąć element z listy musimy przepiąć wskaźnik w jego poprzedniku. Ponieważ przeglądając listy jednokierunkowe nie możemy się cofać, musimy podać jako argument funkcji `erase` wskaźnik do poprzednika usuwanego elementu. Alternatywą jest przejście listy od początku w poszukiwaniu tego poprzednika.

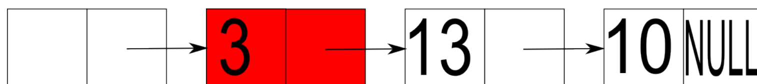


Rysunek 2.6. Operacja insert

Listing 2.13. Operacja erase

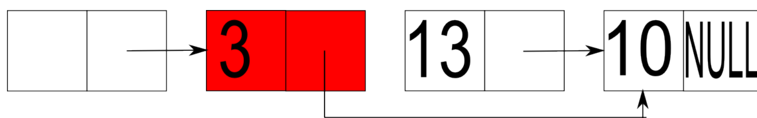
```
void ListaWsk::erase(element * poz){
2   element * pom = poz->nastepny;
    poz->nastepny = pom->nastepny;
4   delete pom;
}
```

Przeanalizujmy teraz działanie funkcji `erase` na przykładzie. Tym razem z listy z rysunku 2.2 chcemy usunąć element o wartości `13`. Aby to zrobić podajemy jego pozycję, czyli wskaźnik do poprzedniego elementu listy jako argument funkcji `erase`.



Rysunek 2.7. Operacja erase

W funkcji zapisujemy do zmiennej pomocniczej wskaźnik do usuwanej struktury oraz przypisujemy polu `nastepny` struktury przechowującej wartość `3` wskaźnik do elementu następującego po usuwanym elemencie:

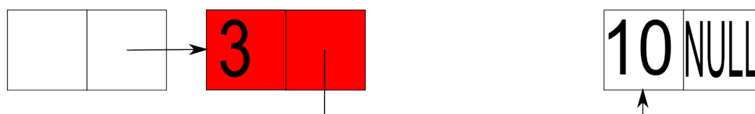


Rysunek 2.8. Operacja erase

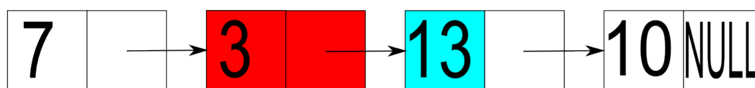
Teraz możemy zwolnić pamięć po usuwanym elemencie (rysunek 2.9) Implementacja funkcji `at` jest bardzo prosta:

Listing 2.14. Operacja at

```
1 int& ListaWsk::at(element * poz){
   return poz->nastepny->dane ;
3 }
```



Rysunek 2.9. Operacja erase



Rysunek 2.10. Operacja at

Podobnie jak w przypadku dwu poprzednich operacji przeanalizujmy przykład. Załóżmy, że w liście z rysunku 2.10 chcemy otrzymać referencję do pola dane w drugim elemencie listy (pole zaznaczone na niebiesko). Oznacza to, że jako argument podajemy wskaźnik do poprzedniego elementu listy, a więc do struktury przechowującej wartość `3` . Aby zwrócić referencję do interesującego nas pola `dane`, musimy przejść na liście o jeden element do przodu w stosunku do wskaźnika, który dostaliśmy w argumencie funkcji `at`.

Operacje `push_front` i `pop_front` różnią się od operacji `insert` i `erase` tylko tym, że zawsze wstawiamy/usuwamy element, którego pozycję opisuje wskaźnik na głowę.

Listing 2.15. Operacje push_front i pop_front

```
1  void ListaWsk::push_front(int e){
     element * nowy = new element;
3    nowy->dane = e;
     nowy->nastepny = glowa->nastepny;
5    glowa->nastepny = nowy;
   }
7
   void ListaWsk::pop_front(){
9    element * pom = glowa->nastepny;
     glowa->nastepny = pom->nastepny;
11   delete pom;
   }
```

Zauważmy, że dzięki temu, iż implementujemy listę z głową w metodzie `push_front`, nie musimy oddzielnie rozważać przypadku, gdy lista jest pusta. W takiej sytuacji do pola `nastepny` wstawianej struktury zostanie przypisana wartość `NULL` z odpowiedniego pola głowy.

Implementując operację `push_back` musimy najpierw odnaleźć ostatni element listy.

Listing 2.16. Operacja push_back

```
void ListaWsk::push_back(int e){
  element * pom = glowa;
  while(pom->nastepny != NULL)
    pom = pom->nastepny;

  ...

}
```

Dopiero wtedy możemy wstawić nowy element.

Listing 2.17. Operacja push_back

```
void ListaWsk::push_back(int e){
  element * pom = glowa;
  while(pom->nastepny != NULL)
    pom = pom->nastepny;
  pom->nastepny = new element;
  pom->nastepny->dane = e;
  pom->nastepny->nastepny = NULL;
}
```

Operację `push_back` możemy przyspieszyć zapamiętując wskaźnik do ostatniego elementu listy. Niestety nie przyspieszy to wykonania operacji `pop_back` podobnie jak nie przyspieszy jej zapamiętanie pozycji ostatniego elementu listy (a więc wskaźnika do przedostatniego elementu listy). Wynika to z faktu, że jeżeli pamiętamy pozycję ostatniego elementu i tenże ostatni element usuniemy, to żeby uzyskać pozycję nowego ostatniego elementu musimy przejść całą listę od początku.

Listing 2.18. Operacja pop_back

```
void ListaWsk::pop_back(){
  element * pom = glowa;
  while(pom->nastepny->nastepny != NULL)
    pom = pom->nastepny;
  delete pom->nastepny;
  pom->nastepny = NULL;
}
```

Podobnie jak miało to miejsce w przypadku operacji `back`, także operacje `push_back` i `pop_back` można zaimplementować wykorzystując wcześniej zaimplementowanie operacje. W przypadku operacji `pop_back` znacznie upraszcza to kod funkcji. W przypadku operacji `push_back` zysk nie jest tak wyraźny, gdyż oddzielnie trzeba rozważyć przypadek gdy lista jest pusta (w przypadku pustej listy działanie operacji `pop_back` jest nieokreślone).

Listing 2.19. Operacje push_back i pop_back

```
1 void ListaWsk::push_back(int e){
    if (glowa
3 ->next==NULL)
      insert(glowa, e);
5   else {
      element * pom = last()
7     insert(pom->next,e);
    }
9 }

11 void ListaWsk::pop_back(){
    erase(last());
13 }
```

Na koniec prosta do zaimplementowania operacja `next`:

Listing 2.20. Operacja next

```
1 element * ListaWsk::next(element * poz){
   return poz->nastepny;
3 }
```

Złożoność obliczeniowa wszystkich przedstawionych metod, poza metodami operującymi na końcu listy, była stała ($O(1)$). Przedstawione implementacje operacji działających na końcu listy (`last`, `back`, `push_back` i `pop_back`) działały w czasie liniowym, ale przy założeniu, że będziemy pamiętali wskaźnik do ostatniego elementu, można je, za wyjątkiem operacji `pop_back`, zaimplementować tak, żeby działały w pesymistycznym czasie stałym. Jeżeli zależy nam na szybkim operowaniu na obu końcach listy, to warto rozważyć listy dwukierunkowe, czyli takie, w których każdy element listy pamięta zarówno wskaźnik do swojego następnika, jak i poprzednika. Listy dwukierunkowe zajmują nieco więcej miejsca w pamięci od list jednokierunkowych, ale za to wszystkie operacje na nich wykonuje się w czasie stałym. W tabeli 2.1 prezentujemy podsumowanie złożoności poszczególnych operacji w różnych wersjach implementacji listy przy pomocy list wskaźnikowych.

Zadanie 1. Zaimplementuj modyfikację klasy `ListaWsk` pamiętającą poza wskaźnikiem na głowę także wskaźnik na ostatni element listy.

Zadanie 2. Zaimplementuj listę przy pomocy dwukierunkowej listy wskaźnikowej. Czym w tej implementacji będzie pozycja elementu?

Tabela 2.1. Złożoność operacji na listach wskaźnikowych

operacja	Lista jednokierunkowa	Lista jednokierunkowa + wskaźnik na koniec	Lista dwukierunkowa + wskaźnik na koniec
`create`	$O(1)$	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$	$O(1)$
`first`	$O(1)$	$O(1)$	$O(1)$
`front`	$O(1)$	$O(1)$	$O(1)$
`last`	$O(n)$	$O(1)$	$O(1)$
`back`	$O(n)$	$O(1)$	$O(1)$
`push_front`	$O(1)$	$O(1)$	$O(1)$
`push_back`	$O(n)$	$O(1)$	$O(1)$
`pop_front`	$O(1)$	$O(1)$	$O(1)$
`pop_back`	$O(n)$	$O(n)$	$O(1)$
`insert`	$O(1)$	$O(1)$	$O(1)$
`erase`	$O(1)$	$O(1)$	$O(1)$
`next`	$O(1)$	$O(1)$	$O(1)$
`at`	$O(1)$	$O(1)$	$O(1)$

2.2.2. Tablicowa implementacja listy

Drugim podstawowym sposobem implementacji listy jest implementacja tablicowa. W tej implementacji kolejne elementy listy są przechowywane w kolejnych komórkach tablicy. Pozycję elementu w tablicowej implementacji listy możemy zdefiniować jako numer pozycji elementu w liście (zakładamy, że pozycje numerujemy od 0). Na początku rozważymy najprostszą wersję implementacji tablicowej, w której kolejne elementy listy są ułożone w tablicy po kolei począwszy od jej pierwszej komórki.

Klasa `ListaTab` będzie zawierać prywatną tablicę `dane`, w której będą przechowywane elementy listy oraz pola `rozmiar` i `liczba_el` przechowujące odpowiednio rozmiar tablicy i liczbę elementów listy (aby zmniejszyć złożoność czasową operacji dodawania do listy, tablica `dane` będzie miała zwykle więcej komórek niż przechowywana lista elementów).

Listing 2.21. Nagłówek klasy ListaTab

```
1 class ListaTab {
  private:
3  int * dane;
   unsigned int rozmiar, liczba_el;
5 public:
    ListaTab();
7   ~ListaTab();
```

```
    bool empty();
9   unsigned int first();
    int front();
11  unsigned int last();
    int back();
13  void push_front(int e);
    void push_back(int e);
15  void pop_front();
    void pop_back();
17  void insert(unsigned int p, int e);
    void erase(unsigned int p);
19  unsigned int next(unsigned int p);
    int& at(unsigned int p)
21 };
```

W konstruktorze musimy tylko zaalokować tablicę `dane` oraz zainicjować wartości pól `rozmiar` i `liczba_el`. Jeżeli nie znamy żadnego rozsądnego ograniczenia na maksymalny rozmiar listy, to początkowy rozmiar tablicy nie ma wpływu na asymptotyczną złożoność obliczeniową poszczególnych operacji.

Listing 2.22. Konstruktor klasy ListaTab

```
1 ListaTab::ListaTab(){
    dane = new int[100];
3   rozmiar = 100;
    liczba_el =0;
5 }
```

W destruktorze musimy jedynie zwolnić pamięć po tablicy `dane`.

Listing 2.23. Destruktor klasy ListaTab

```
1 ListaTab::~ListaTab(){
     delete [] dane;
3 }
```

Ogromną zaletą tablicowej implementacji listy jest łatwość dostępu do dowolnego elementu listy:

Listing 2.24. Operacje first, front, last, back i at

```
1 usigned int ListaTab::first(){
    return 0;
3 }

5 int ListaTab::front(){
    return dane[0];
7 }
```

```
9 unsigned int ListaTab::last(){
    return liczba_el-1;
11 }

13 int ListaTab::back(){
    return dane[liczba_el-1];
15 }

17 int& ListaTab::at(unsigned int p){
    return dane[p];
19 }
```

Podobnie jak w przypadku implementacji listowej funkcje `empty` i `next` są bardzo proste w implementacji:

Listing 2.25. Operacje empty i next

```
1 bool ListaTab::empty(){
    return (liczba_el==0);
3 }

5 unsigned int ListaTab::next(unsigned int p){
    return p+1;
7 }
```

Słabą stroną tablicowej implementacji listy jest dodawanie i usuwanie elementów, zwłaszcza gdy dotyczy to elementów ze środka listy. Musimy wtedy przesuwać pozostałe elementy listy, co zwiększa złożoność czasową operacji. Dodatkowym problemem jest konieczność powiększania tablicy w przypadku jej przepełnienia (musimy wtedy przepisać zawartość starej tablicy do nowej).

Implementując funkcję `insert` musimy rozważyć dwa przypadki. Pierwszy przypadek, w którym tablica jest wypełniona – trzeba wtedy utworzyć nową większą tablicę i przepisać do niej zawartość starej – oraz drugi przypadek, w którym w tablicy są jeszcze wolne miejsca. W tym przypadku część elementów listy musimy przesunąć, aby zrobić miejsce dla nowego elementu.

Zaczniemy od tego pierwszego przypadku. Najpierw musimy utworzyć nową tablicę o dwa razy większym rozmiarze niż stara. Wskaźnik do nowej tablicy przypisujemy do zmiennej pomocniczej (wartość pola `dane` zaktualizujemy po przepisaniu zawartości starej tablicy do nowej).

Listing 2.26. Operacja insert

```
1 void ListaTab::insert(unsigned int p, int e){
    if (lczba_el == rozmiar){
3     int * pom = new int[rozmiar*2];
```

```
5        ...

7    }
     else {
9
         ...
11
     }
13 }
```

Następnie przepiszemy zawartość starej tablicy do nowej. Przepisując tablicę od razu wstawiamy w środku nowy element.

Listing 2.27. Operacja insert

```
1  void ListaTab::insert(unsigned int p, int e){
     if (lczba_el==rozmiar){
3      int * pom = new int[rozmiar*2];
       for(unsigned int i=0;i<p;i++)
5        pom[i]=dane[i];
       pom[p]=e;
7      for(unsigned int i=p+1;i<=liczba_el;i++)
         pom[i]=dane[i-1];
9
         ...
11
     }
13   else {

15       ...

17   }
   }
```

Na koniec tego przypadku musimy zaktualizować wartości pól `rozmiar`, `liczba_el` i `dane` oraz zwolnić pamięć po starej tablicy.

Listing 2.28. Operacja insert

```
   void ListaTab::insert(unsigned int p, int e){
2    if (lczba_el==rozmiar){
       int * pom = new int[rozmiar*2];
4      for(unsigned int i=0;i<p;i++)
         pom[i]=dane[i];
6      pom[p]=e;
       for(unsigned int i=p+1;i<=liczba_el;i++)
8        pom[i]=dane[i-1];
       rozmiar*=2;
10     liczba_el++;
```

```
    delete [] dane;
12  dane = pom;
  }
14 else {

16    ...

18 }
}
```

W sytuacji, gdy w tablicy są jeszcze wolne komórki, musimy najpierw przesunąć elementy z końca listy, aby zrobić miejsce dla nowego elementu.

Listing 2.29. Operacja insert

```
1 void ListaTab::insert(unsigned int p, int e){
    if (lczba_el==rozmiar){
3     int * pom = new int[rozmiar*2];
      for(unsigned int i=0;i<p;i++)
5       pom[i]=dane[i];
      pom[p]=e;
7     for(unsigned int i=p+1;i<=liczba_el;i++)
        pom[i]=dane[i-1];
9     rozmiar*=2;
      liczba_el++;
11    delete [] dane;
      dane = pom;
13  }
    else {
15    for(unsigned int i=liczba_el;i>p;i--)
        dane[i]=dane[i-1];
17
      ...
19
    }
21 }
```

Na koniec wstawiamy nowy element i aktualizujemy pole `liczba_el`.

Listing 2.30. Operacja insert

```
1 void ListaTab::insert(unsigned int p, int e){
    if (lczba_el==rozmiar){
3     int * pom = new int[rozmiar*2];
      for(unsigned int i=0;i<p;i++)
5       pom[i]=dane[i];
      pom[p]=e;
7     for(unsigned int i=p+1;i<=liczba_el;i++)
        pom[i]=dane[i-1];
9     rozmiar*=2;
```

```
      liczba_el++;
11    delete [] dane;
      dane = pom;
13  }
    else {
15    for(unsigned int i=liczba_el;i>p;i--)
        dane[i]=dane[i-1];
17    dane[p]=e;
      liczba_el++;
19  }
  }
```

Jak łatwo oszacować złożoność czasowa powyższej operacji dla listy o rozmiarze n jest w pesymistycznym przypadku $O(n)$. Zwróćmy uwagę, że w przypadku przepełnienia tablicy, nowa tablica jest dwa razy większa od starej. Będzie to miało znaczenie przy szacowaniu zamortyzowanej złożoności operacji `push_back`.

Operacja `push_front` to w naszej implementacji szczególny przypadek operacji `insert`.

Listing 2.31. Operacja push_front

```
  void ListaTab::push_front(int e){
2   insert(0,e);
  }
```

W dalszej części tego rozdziału pokażemy, jak można zmodyfikować tablicową implementację listy, tak żeby przyspieszyć działanie operacji `push_front`.

Operacja `push_back` jest istotnie prostsza od operacji `insert`, którą wykorzystuje tylko w przypadku konieczności powiększenia tablicy.

Listing 2.32. Operacja push_back

```
1 void ListaTab::push_back(int e){
    if (liczba_el==rozmiar)
3     insert(liczba_el,e);
    else {
5     dane[liczba_el]=e;
      liczba_el++;
7   }
  }
```

Pesymistyczna złożoność czasowa operacji `push_back` dla listy o rozmiarze n to $O(n)$. Przyczyną tego jest konieczność przepisywania w przypadku braku miejsca w tablicy wszystkich jej elementów do nowej większej tablicy. Na szczęście jednak zamortyzowana złożoność operacji `push_back` to $O(1)$. Poniżej znajduje się prosty dowód tego faktu.

Zauważmy, że gdyby nie konieczność powiększania tablicy i co za tym idzie przepisywania elementów listy do powiększonej tablicy, to złożoność czasowa operacji `push_back` byłaby $O(1)$. Pokażemy, że sumaryczna złożoność wszystkich powiększeń tablicy przechowującej listę o końcowym rozmiarze n jest $O(n)$ (czyli że pojedynczy element jest przepisywany średnio $O(1)$ razy).

Zauważmy, że co najwyżej połowa elementów listy przechowywanej w tablicy była przepisywana przy powiększaniu listy dokładnie jeden raz. Podobnie co najwyżej jedna czwarta elementów listy była przepisywana dwa razy itd. Dla listy o rozmiarze n dostajemy więc, że sumaryczna liczba przepisań elementów przy wszystkich powiększeniach tablicy razem wziętych wynosi co najwyżej:

$$1 \cdot \frac{1}{2} \cdot n + 2 \cdot \frac{1}{4} \cdot n + 3 \cdot \frac{1}{8} \cdot n + \dots \tag{2.1}$$

Łatwo pokazać, że dla $n \geqslant 3$ zachodzi $\frac{n}{2^n} \leqslant \left(\frac{3}{4}\right)^{n-2}$. Stad otrzymujemy, że suma z równania (2.1) jest mniejsza od:

$$\frac{1}{2} \cdot n + \frac{1}{2} \cdot n + \left(\frac{3}{4}\right)^1 \cdot n + \left(\frac{3}{4}\right)^2 \cdot n + \dots =$$
$$n + \left(\frac{3}{4}\right)^1 \cdot n + \left(\frac{3}{4}\right)^2 \cdot n + \dots = 4 \cdot n$$

Z powyższego otrzymujemy, że sumaryczna liczba przepisań elementów listy przy powiększaniu tablicy była mniejsza od $4 \cdot n$, a więc pojedynczy element jest średnio przepisywany nie więcej niż 4 razy. Powyższe stwierdzenie implikuje, że zamortyzowana złożoność operacji `push_back` wynosi w naszej implementacji $O(1)$.

Jako ostatnie zaimplementujemy operacje służące do usuwania elementów listy w implementacji tablicowej. Usunięcie elementu z dowolnego miejsca listy wymaga przesunięcia wszystkich elementów, które w liście znajdują się za usuwanym elementem:

Listing 2.33. Operacja erase

```
void ListaTab::erase(usigned int p){
  for(int i=p;i<liczba_el-1;i++)
    dane[i]=dane[i+1];
  liczba_el--;
}
```

Jak łatwo oszacować złożoność operacji `erase` w n-elementowej liście wynosi $O(n)$.

Operacja `pop_front` jest w naszej implementacji szczególnym przypadkiem operacji `erase`:

Listing 2.34. Operacja pop_front

```
1 void ListaTab::pop_front(){
    erase(0);
3 }
```

Podobnie jak operację `push_front` także operację `pop_front` można zaimplementować efektywniej, przy pewnej modyfikacji tablicowej implementacji listy.

W rozważanej implementacji dużo prostsze od usuwania pierwszego elementu listy jest usuwanie ostatniego elementu. Zauważmy, że w naszej wersji operacji `pop_back` nie zamazujemy ani nie usuwamy fizycznie ostatniego elementu listy, tylko przesuwamy indeks końca.

Listing 2.35. Operacja pop_back

```
1 void ListaTab::pop_back(){
    liczba_el--;
3 }
```

Głównym mankamentem tablicowej implementacji listy jest konieczność przesuwania elementów listy w trakcie usuwania lub dodawania elementu w środku listy. W przedstawionej implementacji problem ten dotyczy także operacji na początku listy, łatwo jednak tak poprawić naszą implementację, żeby złożoności operacji na początku i na końcu listy były takie same. Wystarczy zrezygnować z warunku, że pierwszy element listy ma znajdować się w pierwszej komórce tablicy i pozwolić żeby lista się „zawijała”, czyli żeby po elemencie listy znajdującym się w ostatniej komórce tablicy następował element znajdujący się w pierwszej komórce tablicy. Na rysunku 2.11 widać przykład takiej listy. Pierwszy element przedstawionej na tym rysunku listy ma wartość 3 i znajduje się w piątej komórce tablicy, zaś ostatni element listy ma wartość 20 i znajduje się w drugiej komórce tablicy. Zwróćmy uwagę, że piąty element listy znajduje się w ostatniej komórce tablicy, zaś następujący po nim element szósty w pierwszej komórce tablicy.

Rysunek 2.11. „Zawijana” tablicowa implementacja listy

W takiej implementacji listy musimy pamiętać, gdzie znajduje się początek listy (będziemy pamiętali indeks w tablicy pierwszego elementu listy). Musimy też pamiętać rozmiar listy oraz rozmiar tablicy. Nagłówek klasy

przechowującej naszą „ulepszoną” implementację listy nie będzie się zbytnio różnił od nagłówka klasy `ListaTab`:

Listing 2.36. Nagłówek klasy ListaTab2

```
class ListaTab2 {
private:
 int * dane;
 unsigned int rozmiar, liczba_el, poczatek;
public:
  ListaTab2();
  ~ListaTab2();
  bool empty();
  unsigned int first();
  int front();
  unsigned int last();
  int back();
  void push_front(int e);
  void push_back(int e);
  void pop_front();
  void pop_back();
  void insert(unsigned int p, int e);
  void erase(unsigned int p);
  unsigned int next(unsigned int p);
  int& at(unsigned int p)
};
```

Przedstawimy implementację tylko wybranych operacji listy. Implementację pozostałych operacji zostawimy jako ćwiczenie do samodzielnego zrobienia. Na początku przyjrzyjmy się, jak zmienią się operacje `next` i `at` w naszej ulepszonej implementacji:

Listing 2.37. Operacje next i at

```
unsigned int next(unsigned int p){
  return (p+1)%rozmiar;
}

int& at(unsigned int p){
  return dane[(poczatek+p)%rozmiar];
}
```

Jak widać indeks komórki tablicy, w której znajduje się element listy o podanej pozycji w liście, można policzyć prostym wzorem.

Podstawowym celem prezentowanej modyfikacji tablicowej implementacji listy jest przyspieszenie działania operacji `pop_front` i `push_front`.

Listing 2.38. Operacje pop_front i push_front

```
void pop_front(){
  poczatek = next(poczatek);
  liczba_el--;
}

void push_front(int e){
  if (liczba_el == rozmiar)
    insert(0,e);
  else {
    poczatek = (poczatek + rozmiar -1)%rozmiar;
    dane[poczatek] = e;
    liczba_el++;
  }
}
```

Złożoność czasowa tak zaimplementowanej operacji `pop_front` to w pesymistycznym przypadku $O(1)$. Pesymistyczna złożoność operacji `push_front` to $O(n)$ ze względu na konieczność powiększania od czasu do czasu tablicy przechowującej listę. Zamortyzowana złożoność tej operacji to $O(1)$, a dowód tego faktu jest analogiczny do dowodu złożoności operacji `push_back` w poprzedniej implementacji listy.

Na zakończenie pokażemy jeszcze implementację operacji `insert`:

Listing 2.39. Operacja insert

```
void insert(unsigned int p, int e){
  if (liczba_el==rozmiar){
    int * pom = new int[2*rozmiar];
    for(unsigned int i=0;i<p;i++)
      pom[i]=at(i);
    pom[p]=e;
    for(unsigned int i=p+1;i<=liczba_el;i++)
      pom[i]=at(i-1);
    delete [] dane;
    dane = pom;
    rozmiar*=2;
  }
  else {
    for(i=liczba_el;i>p;i--)
      at(i)=at(i-1);
    at(p) = e;
  }
  liczba_el++;
}
```

Pesymistyczna złożoność czasowa operacji `insert` w naszej implementacji wynosi $O(n)$. Jest to związane z koniecznością zrobienia miejsca dla wsta-

wianego elementu, co wymaga przesunięcia w tablicy części elementów listy. Zauważmy, że w naszej implementacji wykorzystaliśmy wcześniej zdefiniowaną metodę `at`, której użycie znacznie uprościło implementację funkcji. Implementację pozostałych operacji pozostawiamy czytelnikowi jako ćwiczenie.

W tabeli 2.2 podsumowujemy pesymistyczną złożoność czasową operacji na liście zaimplementowanych przy pomocy tablicy. W nawiasach podano złożoność amortyzowaną tam gdzie się ona różni od złożoności pesymistycznej. Szczególnie ciekawa jest ostatnia kolumna pokazująca nieprzekraczalne ograniczenia tablicowej implementacji listy. Nawet znając maksymalny potrzebny rozmiar tablicy i stosując „zawijaną" implementację listy nie jesteśmy w stanie istotnie zmniejszyć złożoności czasowej operacji `insert` i `erase`.

Tabela 2.2. Złożoność operacji w tablicowych implementacjach listy

operacja	Prosta implementacja tablicowa	„Zawijana" implementacja tablicowa	„Zawijana" implementacja tablicowa przy stałym rozmiarze tablicy
`create`	$O(1)$	$O(1)$	$O(1)$
`empty`	$O(1)$	$O(1)$	$O(1)$
`first`	$O(1)$	$O(1)$	$O(1)$
`front`	$O(1)$	$O(1)$	$O(1)$
`last`	$O(1)$	$O(1)$	$O(1)$
`back`	$O(1)$	$O(1)$	$O(1)$
`push_front`	$O(n)$	$O(n)$ $(O(1))$	$O(1)$
`push_back`	$O(n)$ $(O(1))$	$O(n)$ $(O(1))$	$O(1)$
`pop_front`	$O(n)$	$O(1)$	$O(1)$
`pop_back`	$O(1)$	$O(1)$	$O(1)$
`insert`	$O(n)$	$O(n)$	$O(n)$
`erase`	$O(n)$	$O(n)$	$O(n)$
`next`	$O(1)$	$O(1)$	$O(1)$
`at`	$O(1)$	$O(1)$	$O(1)$

Zadanie 3. Zaimplementuj modyfikację klasy `ListaTab`, w której rozmiar tablicy przechowującej listę jest argumentem konstruktora i nie ulega zmianie w trakcie używania obiektu tej klasy.

Zadanie 4. Zaimplementuj brakujące metody klasy `ListaTab2` zawierającej „zawijaną" tablicową implementację listy.

Zadanie 5. Zmodyfikuj operację `insert` z listingu 2.39 w taki sposób, żeby robiąc miejsce na nowy element przesuwała elementy sprzed lub zza wstawianego elementu w zależności od tego, których jest mniej.

2.3. Proste algorytmy sortujące

W tym rozdziale poznamy proste algorytmy operujące na listach. Będą to algorytmu sortujące, czyli algorytmy, które będą zmieniały kolejność elementów na liście w taki sposób, żeby elementy wynikowej listy tworzyły ciąg niemalejący. Inaczej mówiąc wynikowa lista będzie uporządkowana według zadanego porządku. Niektórzy autorzy nazywają sortowaniem dowolne monotoniczne uporządkowanie listy dopuszczając w ten sposób także uporządkowanie nierosnące. Przy naszej, przyjętej za [7], definicji sortowania, sortowanie nierosnące to sortowanie przy odwróconym porządku.

Algorytmy, które poznamy w tym rozdziale są proste, ale niezbyt efektywne. Szybsze algorytmy sortujące czytelnik pozna w rozdziałach 4.1.1, 4.1.2 i 7.3.3.

W przypadku algorytmów sortujących przyjęło się uznawać porównywanie elementów listy za operację dominującą. Dlatego szacując złożoność przedstawionych algorytmów, oprócz liczby wszystkich operacji, będziemy zliczali także liczbę porównań.

2.3.1. Sortowanie przez wybieranie

Algorytm sortowania przez wybieranie jest jednym z najprostszych i najbardziej naturalnych algorytmów sortujących. W algorytmie tym w nieposortowanej liście wyszukujemy kolejne pod względem wielkości elementy i przenosimy je po kolei do listy wynikowej. Przenoszone elementy usuwamy z listy wejściowej, co oznacza, że za każdym razem w liście wejściowej szukamy największego z pozostałych elementów.

Implementację algorytmu sortowania przez wstawianie rozpoczniemy od funkcji szukającej pozycji największego elementu w podanej w argumencie liście.

Listing 2.40. Sortowanie przez wybieranie

```
1 Pozycja Najwiekszy(Lista Wejscie){

3     ...

5 }
```

W funkcji `Najwiekszy` będziemy potrzebowali dwóch zmiennych: `pom` przyjmującej jako wartości pozycje kolejnych elementów otrzymanej w argumencie listy oraz `max` przechowującej pozycję elementu listy o największej dotychczas znalezionej wartości. Do zmiennych `max` i `pom` przypiszemy na początek pozycję pierwszego elementu listy.

Listing 2.41. Sortowanie przez wybieranie

```
1 Pozycja Najwiekszy(Lista Wejscie){
    Pozycja pom, max;
3   pom = max = Wejscie.first();

5   ...

7 }
```

Największego elementu listy będziemy szukali przy pomocy pętli `while`, w której zmienna `pom` będzie przebiegać pozycje kolejnych elementów listy. Pętla skończy działanie, gdy zmienna `pom` dojdzie do pozycji ostatniego elementu. Po zakończeniu pętli wartością zmiennej `max` będzie pozycja największego elementu listy `Wejscie`:

Listing 2.42. Sortowanie przez wybieranie

```
1 Pozycja Najwiekszy(Lista Wejscie){
    Pozycja pom, max;
3   pom = max = Wejscie.first();
    while(pom != Wejscie.last()){
5
      ...
7
    }
9   return max;
  }
```

Pozostało nam jeszcze uzupełnienie wnętrza pętli `while`. Na pewno musi się tam znaleźć linia, w której będziemy zmieniać wartość zmiennej `pom` tak, żeby przyjmowała jako wartości pozycje kolejnych elementów listy `Wejscie`. Wykorzystamy do tego funkcję `next`. Ponadto dla każdej wartości zmiennej `pom` (czyli w każdym obrocie pętli), musimy sprawdzić czy element listy, którego pozycję przechowuje `pom`, jest większy od największego dotychczas znalezionego elementu listy (jeżeli jest, to aktualizujemy wartość zmiennej `max`).

Listing 2.43. Sortowanie przez wybieranie

```
   Pozycja Najwiekszy(Lista Wejscie){
 2   Pozycja pom, max;
     pom = max = Wejscie.first();
 4   while(pom != Wejscie.last()){
       pom = Wejscie.next(pom)
 6     if (Wejscie.at(max) < Wejscie.at(pom))
         max = pom;
 8   }
     return max;
10 }
```

Zauważmy, że najpierw zmieniamy wartość zmiennej `pom`, a dopiero potem sprawdzamy, czy na pozycji wskazywanej przez tą zmienną jest element większy lub równy największemu dotychczas znalezionemu. Gdybyśmy wykonywali te czynności w odwrotnej kolejności, to nie wzięlibyśmy pod uwagę wartości ostatniego elementu listy.

Teraz zaimplementujemy funkcję sortującą, która będzie wykorzystywała funkcję `Najwiekszy`. Funkcja ta dostanie jako argument nieposortowaną listę i zwróci jako wartość posortowaną listę o takich samych elementach. Aby takie rozwiązanie działało poprawnie w języku C++, należy przeciążyć operator przypisania dla typu `Lista`.

Listing 2.44. Sortowanie przez wybieranie

```
   Lista SelectionSort(Lista Wejscie){
 2
     ...
 4
   }
```

W funkcji `SelectionSort` będziemy potrzebowali dwóch zmiennych: `Wyjscie`, która będzie przechowywała posortowaną listę i którą na koniec zwrócimy jako wartość funkcji oraz `pom`, która będzie przechowywała pozycję największego elementu w wejściowej liście.

Listing 2.45. Sortowanie przez wybieranie

```
 1 Lista SelectionSort(Lista Wejscie){
     Lista Wyjscie;
 3   Pozycja pom;
     ...
 5   return Wyjscie;
   }
```

Będziemy przenosili elementy listy `Wejscie` do listy `Wyjscie` w pętli `while` tak długo, dopóki lista `Wejscie` nie będzie pusta.

Listing 2.46. Sortowanie przez wybieranie

```
  Lista SelectionSort(Lista Wejscie){
2   Lista Wyjscie;
    Pozycja pom;
4   while (!Wejscie.empty()){
    ...
6   }
    return Wyjscie;
8 }
```

W każdym obrocie pętli `while` wykonywamy trzy instrukcje: najpierw znajdziemy największy w danym momencie element listy `Wejscie` przy pomocy funkcji `Najwiekszy`, następnie wstawimy ten element do listy `Wyjscie` i usuniemy go z listy `Wejscie`.

Listing 2.47. Sortowanie przez wybieranie

```
   Lista SelectionSort(Lista Wejscie){
 2   Lista Wyjscie;
     Pozycja pom;
 4   while (!Wejscie.empty()){
       pom = Najwiekszy(Wejscie);
 6     Wyjscie.push_front(Wejscie.at(pom));
       Wejscie.erase(pom);
 8   }
     return Wyjscie;
10 }
```

Teraz oszacujemy złożoność czasową funkcji `SelectionSort`. Zrobimy to szacując liczbę operacji dominujących, czyli w tym wypadku porównań. Łatwo oszacować, że wykonanie funkcji `Najwiekszy` dla listy o rozmiarze k wymaga $k-1$ porównań, zaś pętla `while` w funkcji `SelectionSort` dla listy o rozmiarze n wykona się n razy wywołując za każdym razem funkcję `Najwiekszy` dla listy mniejszej o jeden niż w poprzednim obrocie pętli. W związku z tym złożoność algorytmu sortującego to:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n}{2} \cdot (n-1),$$

a to jest $O(n^2)$. Gdybyśmy, zamiast liczby porównań, szacowali liczbę wszystkich wykonanych operacji, to przy typowych funkcjach porównujących i dowolnej sensownej implementacji listy także dostalibyśmy $O(n^2)$ jako ograniczenie na złożoność czasową sortowania listy o rozmiarze n. Zauważmy też,

że złożoność algorytmu sortowania przez wybieranie nie zależy od kolejności elementów wejściowej listy.

Na koniec pokażemy, jak zaimplementować algorytm sortowania przez wybieranie nie korzystając z operacji listy, ale operując wprost na tablicy. Zaletą implementacji z listingu 2.48 jest m.in. to, że sortuje ona elementy tablicy „w miejscu" bez wykorzystywania tablicy pomocniczej do przechowywania posortowanej listy.

Listing 2.48. Sortowanie przez wybieranie

```
void SelectionSort(Element Wejscie[], unsigned int rozmiar){
  unsigned int max;
  Element pom;
  for(unsigned int i=rozmiar;i>1;i--){
    max=0;
    for(unsigned int j=1;j<i;j++)
      if (Wejscie[max]<Wejscie[j])
        max = j;
    pom = Wejscie[max];
    Wejscie[max] = Wejscie[i-1];
    Wejscie[i-1]=pom;
  }
}
```

2.3.2. Sortowanie przez wstawianie

Podobnym algorytmem do algorytmu sortowania przez wybieranie jest algorytm sortowania przez wstawianie. Tym razem bierzemy po kolei elementy nieposortowanej listy i dodajemy je w odpowiednim miejscu nowej posortowanej listy.

Nasza funkcja jako argument dostanie nieposortowaną listę i zwróci jako wartość listę posortowaną.

Listing 2.49. Sortowanie przez wstawianie

```
Lista InsertSort(Lista Wejscie){

  ...

}
```

Będziemy potrzebowali dwu zmiennych: `Wyjscie`, której użyjemy do przechowywania posortowanej listy i zwrócimy na końcu jako wartość funkcji oraz `poz`, która będzie przechowywała pozycje kolejnych elementów listy `Wyjscie` w trakcie wstawiania do niej nowych elementów.

Listing 2.50. Sortowanie przez wstawianie

```
1 Lista InsertSort(Lista  Wejscie){
    Pozycja poz;
3   Lista Wyjscie;

5   ...

7   return Wyjscie;
  }
```

Funkcja `InsertSort` będzie przenosiła kolejne elementy listy `Wejscie` do posortowanej listy `Wyjscie` tak długo, aż nie przeniesie wszystkich elementów listy `Wejscie`. Użyjemy do tego pętli `while`, która będzie działać tak długo, dopóki lista `Wejscie` nie będzie pusta.

Listing 2.51. Sortowanie przez wstawianie

```
  Lista InsertSort(Lista  Wejscie){
2   Pozycja poz;
    Lista Wyjscie;
4   while(!Wejscie.empty()){

6     ...

8     Wejscie.pop_front();
    }
10  return Wyjscie;
  }
```

W każdym obrocie pętli `while` musimy odnaleźć miejsce w tablicy `Wyjscie`, w którym należy wstawić pierwszy element listy `Wyjscie`. Najpierw rozważymy przypadek w którym wstawiany element jest większy lub równy od wszystkich elementów listy `Wyjscie`.

Listing 2.52. Sortowanie przez wstawianie

```
1 Lista InsertSort(Lista  Wejscie){
    Pozycja poz;
3   Lista Wyjscie;
    while(!Wejscie.empty()){
5     if (Wejscie.front() >= Wyjscie.back())
        Wyjscie.push_back(Wejscie.front());
7
      ...
9     Wejscie.pop_front();
    }
11  return Wyjscie;
  }
```

Teraz możemy założyć, że pierwszy element listy `Wyjscie` jest mniejszy od ostatniego elementu listy `Wyjscie`. To założenie uprości nam warunek w pętli `while` szukającej miejsce w liście `Wyjscie`, na które trzeba wstawić pierwszy element listy `Wejscie` (możemy założyć, że znajdziemy to miejsce przed dojściem do końca listy `Wyjscie`).

Listing 2.53. Sortowanie przez wstawianie

```
Lista InsertSort(Lista  Wejscie){
  Pozycja poz;
  Lista Wyjscie;
  while(!Wejscie.empty()){
    if (Wejscie.front() >= Wyjscie.back())
      Wyjscie.push_back(Wejscie.front());
    else {
      poz = Wyjscie.first();
      while(Wejscie.front() <= Wyjscie.at(poz))
        poz = Wyjscie.next(poz);

      ...

    }
    Wejscie.pop_front();
  }
  return Wyjscie;
}
```

Po wyjściu z wewnętrznej pętli `while` zmienna `poz` przechowuje pozycję w liście `Wyjscie`, na którą należy wstawić pierwszy element listy `Wejscie`.

Listing 2.54. Sortowanie przez wstawianie

```
Lista InsertSort(Lista  Wejscie){
  Pozycja poz;
  Lista Wyjscie;
  while(!Wejscie.empty()){
    if (Wejscie.front() >= Wyjscie.back())
      Wyjscie.push_back(Wejscie.front());
    else {
      poz = Wyjscie.first();
      while(Wejscie.front() <= Wyjscie.at(poz))
        poz = Wyjscie.next(poz);
      Wyjscie.insert(poz, Wejscie.front());
    }
    Wejscie.pop_front();
  }
  return Wyjscie;
}
```

Pesymistyczna złożoność powyższego algorytmu dla listy o długości n wynosi $O(n^2)$. Wynika to z faktu, że rozpoczynająca się w czwartej linii listingu 2.54 zewnętrzna pętla `while` obraca się n razy, natomiast w każdym jej obrocie wykonywanych jest $O(n)$ porównań (taka jest liczba obrotów wewnętrznej pętli, taka też jest liczba porównań w wewnętrznej pętli). Taką samą złożoność dostaniemy jeżeli będziemy zliczać wszystkie kroki powyższego algorytmu i to przy każdej z poznanych implementacji listy. Nawet przy takiej, w której operacje `push_back`, `pop_front` lub `insert` mają złożoność liniową (każdą z tych operacji wykonamy co najwyżej n razy). Średnia złożoność algorytmu sortowania przez wstawianie (przy takim samym prawdopodobieństwie wystąpienia wszystkich permutacji listy) jest taka sama, jak jego złożoność pesymistyczna, czyli wynosi $O(n^2)$. Niezależnie od tego czas działania algorytmu sortowania przez wstawianie zależy ściśle od sortowanej permutacji. Zauważmy, że w przypadku sortowania listy uporządkowanej odwrotnie niż tego chcemy, złożoność powyższego algorytmu jest liniowa (o ile zastosujemy implementację listy, w której złożoność operacji `pop_front` jest stała, zaś złożoność operacji `insert` jest $O(b)$, gdzie b jest odległością wstawianego elementu od początku listy – warunek ten w oczywisty sposób spełniają listy wskaźnikowe, gdyż złożoność obu rozważanych operacji wynosi $O(1)$). Dzieje się tak, gdyż w takiej sytuacji program ani razu nie wejdzie do wewnętrznej pętli `while`.

Teraz zobaczymy jak zaimplementować omawiany algorytm tak by sortował jednokierunkową list wskaźnikową bez głowy nie używając operacji listy.

Listing 2.55. Sortowanie przez wstawianie – lista wskaźnikowa

```
   struct Lista {
 2   Element el;
     Lista * next;
 4 };

 6 Lista* InsertSort(Lista * Wejscie){
     Lista * Wyjscie = NULL, pom1,pom2;
 8   while(Wejscie != NULL){
       if (Wyjscie == NULL){
10       Wyjscie = Wejscie;
         Wejscie = Wejscie->next;
12       Wyjscie->next = NULL:
       }
14     else if (Wejscie->el < Wyjscie->el) {
         pom1=Wyjscie;
16       Wyjscie = Wejscie;
         Wejscie = Wejscie->next;
18       Wyjscie->next = pom1;
       }
```

```
20      else {
          pom1 = Wyjscie;
22        while((pom1->next!=NULL)&&
                (pom1->next->el < Wejscie->el))
24          pom1 = pom1->next;
          pom2 = pom1->next;
26        pom1->next = Wejscie;
          Wejscie = Wejscie->next;
28        pom1->next->next = pom2;
        }
30    }
      return Wyjscie;
32  }
```

Poniżej przedstawiamy funkcję sortującą „w miejscu" otrzymaną tablicę przy pomocy algorytmu sortowania przez wstawianie. Posortowana już część listy jest przechowywana w początkowej części tablicy, zaś jej część nieposortowana jest przechowywana na końcu tablicy. W każdym obrocie głównej pętli algorytmu usuwamy pierwszy element znajdujący się w nieposortowanej części listy i wstawiamy go w odpowiednie miejsce części posortowanej. Szukając miejsca dla elementu przenoszonego z nieposortowanej części listy do części posortowanej, zamieniamy go z jego kolejnymi poprzednikami tak długo, dopóki nie znajdzie się on w odpowiednim miejscu. W każdym obrocie zewnętrznej pętli część posortowana zwiększa się o jeden, a część nieposortowana zmniejsza o jeden.

Listing 2.56. Sortowanie przez wstawianie – tablica

```
    void InsertSort(Element tab[], unsigned int n){
 2    unsigned int i,j,k;
      Element pom;
 4    for(i=1;i<n;i++){
        pom=tab[i];
 6      for(j=i;(j>0)&&(tab[j-1]>pom);j--)
          tab[j]=tab[j-1];
 8      tab[j]=pom;
      }
10  }
```

Zauważmy, że w przeciwieństwie do dwu poprzednich implementacji sortowania przez wstawianie, powyższa funkcja działa w czasie liniowym dla listy bliskiej posortowanej (w poprzednich implementacjach taką złożoność programy osiągały dla danych w odwrotnej kolejności).

2.3.3. Sortowanie bąbelkowe

Ostatnim z prostych algorytmów sortujących, który zaprezentujemy w tym rozdziale jest algorytm sortowania bąbelkowego. Polega on na porównywaniu ze sobą kolejnych par elementów (pierwszego z drugim, drugiego z trzecim itd.) i zamienianiu ich miejscami w przypadku, gdy pierwszy element z pary jest większy. Łatwo zauważyć, że po jednej takiej serii porównań na końcu listy znajdzie się jej największy element, po dwóch seriach porównań na przedostatnim miejscu pojawi się drugi co do wielkości element itd., aż po co najwyżej $n-1$ seriach porównań n-elementowa lista będzie już posortowana.

Algorytm sortowania bąbelkowego jest jednym z najprostszych do implementacji algorytmów sortujących tablice. Jeżeli jednak chcemy posortować listę używając jedynie standardowych operacji listy, to implementacja się komplikuje. Dlatego tym razem zaczniemy od wersji sortującej zawartość zwykłej tablicy, a na koniec zobaczymy, że sortowanie bąbelkowe można zaimplementować także operując wyłącznie na standardowych operacjach na listach.

Nasza funkcja sortująca jako argumenty otrzyma tablicę oraz jej rozmiar i posortuje „w miejscu" otrzymaną w argumencie tablicę. W związku z tym implementowana funkcja nie będzie zwracała żadnej wartości.

Listing 2.57. Sortowanie bąbelkowe

```
void BubleSort(Element Wejscie[], unsigned int rozmiar){
2
    ...
4
}
```

W funkcji `BubleSort` będziemy potrzebowali zmiennej pomocniczej, która umożliwi nam zamianę wartościami sąsiednich elementów tablicy.

Listing 2.58. Sortowanie bąbelkowe

```
1 void BubleSort(Element Wejscie[], unsigned int rozmiar){
    Element pom;
3
    ...
5
}
```

Zasadniczą częścią algorytmu są dwie zagnieżdżone w sobie pętle `for`. Zmienna sterująca zewnętrznej pętli będzie przyjmowała jako wartość indeks ostatniego elementu nieposortowanej części tablicy.

Listing 2.59. Sortowanie bąbelkowe

```
void BubleSort(Element Wejscie[], unsigned int rozmiar){
  Element pom;
  for(unsigned int i = rozmiar-1;i>0;i--)

  ...

}
```

Zmienna sterująca w wewnętrznej pętli będzie przebiegać po wartościach od 0 do $i-1$ (czyli po indeksach pierwszych elementów kolejnych par).

Listing 2.60. Sortowanie bąbelkowe

```
void BubleSort(Element Wejscie[], unsigned int rozmiar){
  Element pom;
  for(unsigned int i = rozmiar-1;i>0;i--)
    for(unsigned int j=0;j<i;j++)

  ...

}
```

W wewnętrznej pętli zamieniamy miejscami wartości w kolejnych komórkach tablicy, jeżeli zakłócają one porządek, do którego dążymy (czyli jeżeli pierwszy element w parze jest większy od drugiego).

Listing 2.61. Sortowanie bąbelkowe

```
void BubleSort(Element Wejscie[], unsigned int rozmiar){
  Element pom;
  for(unsigned int i=rozmiar-1;i>0;i--)
    for(unsigned int j=0;j<i;j++)
      if (Wejscie[j]>Wejscie[j+1]){
        pom=Wejscie[j];
        Wejscie[j]=Wejscie[j+1]
        Wejscie[j+1]=pom;
      }
}
```

Złożoność algorytmu sortowania bąbelkowego to $O(n^2)$ i nie zależy ona w zaimplementowanej przez nas wersji tego algorytmu od sposobu uporządkowania elementów nieposortowanej listy.

Pierwszym usprawnieniem, które możemy wprowadzić w naszym algorytmie jest sprawdzanie, czy w danym obrocie zewnętrznej pętli jakieś dwa elementy zostały zamienione miejscami. Jeżeli żadne dwa elementy nie zostały zamienione, to oznacza, że lista jest już posortowana. Do sprawdzenia

faktu, czy została dokonana jakaś zamiana, wykorzystamy zmienną logiczną `zamienilem`, której wartość będzie zmieniana po wejściu do operacji `if` (czyli przy każdej zamianie miejscami elementów). Aby nasze rozwiązanie zadziałało, musimy umieścić zmienną `zamienilem` w warunku zewnętrznej pętli `for` oraz resetować jej wartość (nadawać jej wartość `false`) przed każdym wejściem do wewnętrznej pętli.

Listing 2.62. Sortowanie bąbelkowe

```
void BubleSort(Element Wejscie[], unsigned int rozmiar){
  Element pom;
  bool zamienilem = true;
  for(unsigned int i=rozmiar-1;(i>0)&&(zamienilem);i--){
    zamienilem = false;
    for(unsigned int j=0;j<i;j++)
      if (Wejscie[j]>Wejscie[j+1]){
        pom=Wejscie[j];
        Wejscie[j]=Wejscie[j+1]
        Wejscie[j+1]=pom;
        zamienilem = true;
      }
  }
}
```

Innym usprawnieniem, które możemy wprowadzić, jest szybsze ograniczanie zakresu wartości, jakie przebiega zmienna sterująca wewnętrznej pętli (zmienna `j`). Zauważmy, że jeżeli w którymś z wykonań wewnętrznej pętli ostatnią parą, dla której dokonamy zamiany, jest para elementów o indeksach $(x, x+1)$, to oznacza, że po tej zamianie od elementu o indeksie $x+1$ aż do swojego końca lista osiągnęła już ostateczny kształt. W związku z powyższym będziemy zapamiętywali miejsce ostatniej zamiany w zmiennej `ost_zamiana` i na koniec każdego obrotu zewnętrznej pętli przypisywali tą wartość zmiennej `i`. W nowej wersji nie będziemy już potrzebowali zmiennej `zamienilem` i będziemy mogli uprościć warunek w zewnętrznej pętli `for` (w przypadku, gdy w którymś wykonaniu wewnętrznej pętli `for` nie dokonamy żadnej zamiany, zmienna `i` będzie miała wartość 0 i wyjdziemy z zewnętrznej pętli).

Listing 2.63. Sortowanie bąbelkowe

```
void BubleSort(Element Wejscie[], unsigned int rozmiar){
  Element pom;
  unsigned int ost_zamiana;
  for(unsigned int i=rozmiar-1;(i>0);i--){
    ost_zamiana = 0;
    for(unsigned int j=0;j<i;j++)
      if (Wejscie[j]>Wejscie[j+1]){
```

```
8           pom=Wejscie[j];
            Wejscie[j]=Wejscie[j+1]
10          Wejscie[j+1]=pom;
            ost_zamiana = j+1;
12        }
        i=ost_zamiana;
14    }
  }
```

Zauważmy, że gdybyśmy w jedenastej linii powyższego kodu do zmiennej `ost_zamiana` przypisywali `j` zamiast `j+1`, to potem nie potrzebowalibyśmy zmniejszać w zewnętrznej pętli wartości zmiennej `i`. Poniżej nieco uproszczony kod, w którym zewnętrzną pętlę `for` zastąpiliśmy pętlą `while`.

Listing 2.64. Sortowanie bąbelkowe

```
1 void BubleSort(Element Wejscie[], unsigned int rozmiar){
    Element pom;
3   unsigned int ost_zamiana;
    unsigned int i=rozmiar-1;
5   while (i>0){
      ost_zamiana = 0;
7     for(unsigned int j=0;j<i;j++)
        if (Wejscie[j]>Wejscie[j+1]){
9           pom=Wejscie[j];
            Wejscie[j]=Wejscie[j+1]
11          Wejscie[j+1]=pom;
            ost_zamiana = j;
13        }
        i=ost_zamiana;
15    }
  }
```

Dokonane przez nas modyfikacje algorytmu sortowania bąbelkowego mogą w wielu przypadkach znacznie przyspieszyć działanie algorytmu. Niestety jego pesymistyczna i średnia złożoność czasowa dla listy o długości n wciąż wynosi $O(n^2)$. Co więcej algorytm sortowania bąbelkowego działa w czasie $O(n^2)$ nawet dla prawie posortowanych danych, jeżeli któryś z najmniejszych elementów listy znajduje się blisko jej końca (na przykład dla listy $2, 3, 4, \ldots, n, 1$ algorytm będzie potrzebował $\frac{n \cdot (n-1)}{2}$ porównań).

Na koniec pokażemy, jak zaimplementować algorytm sortowania bąbelkowego wyłącznie przy pomocy operacji listy. Tym razem do funkcji przekażemy referencję do sortowanej listy i dzięki temu funkcja sortująca nie będzie musiała zwracać wartości.

Listing 2.65. Sortowanie bąbelkowe

```
void BubleSort(Lista& Wejscie){
  Element pom;
  Pozycja ost_zamiana, poz, koniec=Wejscie.last();
  while (koniec!=Wejscie.first()){
    ost_zamiana = Wejscie.first();
    for(poz = Wejscie.first(); poz != koniec ;
                                poz=Wejscie.next(poz))
      if (Wejscie.at(poz)>Wejscie.at(Wejscie.next(poz))){
        pom=Wejscie.at(poz);
        Wejscie.at(poz)=Wejscie.at(Wejscie.next(poz))
        Wejscie.at(Wejscie.next(poz))=pom;
        ost_zamiana = poz;
      }
    koniec = ost_zamiana;
  }
}
```

Złożoność powyższego algorytmu będzie podobna dla każdej rozsądnej implementacji listy.

Zadanie 6. Napisz funkcję, która dowolnym algorytmem sortuje malejąco (czyli tak, żeby elementy listy były ułożone od największego do najmniejszego) otrzymaną w argumencie tablicę liczb.

Zadanie 7. Napisz funkcję, która dostaje jako argument listę par liczb i sortuje otrzymaną listę w pierwszej kolejności rosnąco po pierwszej z liczb, a w drugiej kolejności (w przypadku równych pierwszych liczb) malejąco po drugiej z liczb.

2.4. Kolejka, stos, kolejka priorytetowa

Na koniec tego rozdziału zaprezentujemy trzy proste abstrakcyjne struktury danych, które możemy łatwo zaimplementować wykorzystując do tego celu listy.

2.4.1. Kolejka

Kolejka to rodzaj listy umożliwiający dodawanie elementów na jednym końcu listy, a czytanie i usuwanie elementów na drugim końcu listy. Kolejkę nazywamy też listą FIFO (z ang. *first in first out*). Podstawowe operacje na kolejce to:

— `create` – tworzy pustą kolejkę,
— `empty(Queue)` – zwraca wartość `true`, jeżeli kolejka `Queue` jest pusta i `false` w przeciwnym przypadku,

— `push(Queue,e)` – wstawia element `e` na końcu kolejki `Queue`,
— `front(Queue)` – zwraca pierwszy element kolejki `Queue`,
— `pop(Queue)` – usuwa pierwszy element kolejki `Queue`,

Jeżeli przyjrzymy się powyższym operacjom, od razu widzimy, że są to po prostu wybrane operacje listy. W związku z tym łatwo jest zaimplementować kolejkę używając dowolnej gotowej implementacji listy.

Listing 2.66. Nagłówek klasy Queue

```
class Queue {
private:
  List Lista;
public:
  bool empty();
  void push(Element e);
  Element front();
  void pop();
};
```

W powyższej implementacji nie potrzebujemy metody `create`, gdyż zastępuje ją domyślny konstruktor klasy `Queue`. Prywatne pole `Lista` jest typu `List`, o którym zakładamy, że jest implementacją listy. Operacje kolejki będą bardzo proste, gdyż będą polegały wyłącznie na uruchomieniu odpowiednich metod klasy `List`.

Listing 2.67. Metody klasy Queue

```
bool Queue::empty(){
  return Lista.empty()
}

void Queue::push(Element e){
  Lista.push_back(e);
}

Element Queue::front(){
  return Lista.front();
}

void Queue::pop(){
  Lista.pop_front()
}
```

Złożoność poszczególnych operacji zależy od sposobu implementacji listy. Jeżeli listę zaimplementujemy przy pomocy listy wskaźnikowej, to wszystkie operacje kolejki będą miały pesymistyczną złożoność stałą. W przypadku implementacji listy przy pomocy „zawijanej” tablicy operacja `push_back`

będzie miała pesymistyczną złożoność liniową, ale zamortyzowaną złożoność stałą. Najmniej efektywna będzie zwykła tablicowa implementacja listy, gdyż wtedy złożoność operacji `pop_front` będzie liniowa zarówno w pesymistycznym jak i zamortyzowanym przypadku.

Zadanie 8. Zaimplementuj kolejkę bez korzystania z gotowych implementacji listy.

2.4.2. Stos

Kolejną abstrakcyjną strukturą danych, będącą rodzajem listy z ograniczonym zbiorem dostępnych operacji jest stos. Tym razem jest to struktura, w której wszystkich operacji dokonujemy na jednym końcu listy. Stos nazywamy też listą LIFO (z ang. *last in first out*). Podstawowe operacje na stosie to:
- `create` – tworzy pusty stos,
- `empty(Stack)` – zwraca `true`, jeżeli stos `Stack` jest pusty i `false` w przeciwnym przypadku,
- `push(Stack, e)` – połóż na stosie `Stack` (wstaw na koniec listy) element `e`,
- `top(Stack)` – zwraca wartość z wierzchu stosu `Stack` (ostatni elementu listy),
- `pop(Stack)` – zdejmuje element z wierzchu stosu `Stack` (usuwa ostatni element listy).

Podobnie jak w przypadku kolejki do implementacji stosu użyjemy listy:

Listing 2.68. Nagłówek klasy Stack

```
1 class Stack {
  private:
3   List Lista;
  public:
5   bool empty();
    void push(Element e);
7   Element top();
    void pop();
9 };
```

Podobnie jak w przypadku listy, wszystkie operacje na stosie odpowiadają jakimś operacjom na liście:

Listing 2.69. Metody klasy Stack

```
1   bool empty() {
      return Lista.empty();
3   }
```

```
5    void push(Element e){
       Lista.push_back(e);
7    }

9    Element top(){
       return Lista.back();
11   }

13   void pop(){
       Lista.pop_back();
15   }
```

Tym razem używając tablicowej implementacji listy, zarówno zwykłej, jak i „zawijanej” dostajemy tą samą złożoność operacji na liście. W obu tych implementacjach operacje `empty`, `top` oraz `pop` mają pesymistyczną złożoność stałą, natomiast operacja `push` ma pesymistyczną złożoność liniową i zamortyzowaną złożoność stałą. Natomiast aby osiągnąć stałą złożoność czasową wszystkich operacji przy implementacji listy za pomocą listy wskaźnikowej powinniśmy operować na początku listy zamiast jej końca (czyli używać operacji `push_front`, `front` i `pop_front` zamiast `push_back`, `back` i `pop_back`) lub użyć listy dwukierunkowej.

Zadanie 9. Zaimplementuj stos bez korzystania z gotowej implementacji listy.

Zadanie 10. Napisz program, który wczytuje ze standardowego wejścia liczbę n oraz n liczb całkowitych i wypisuje na standardowym wyjściu wczytane liczby w odwróconej kolejności.

2.4.3. Kolejka priorytetowa

Kolejka priorytetowa jest strukturą danych, w której z każdym elementem powiązana jest dodatkowa wartość nazywana priorytetem. O miejscu poszczególnych elementów w strukturze decyduje nie kolejność dodawania ich do struktury, ale priorytety. Kolejka priorytetowa udostępnia następujące operacje:

— `create` – tworzy pustą kolejkę priorytetową,
— `empty(PriorityQueue)` – zwraca `true` jeżeli kolejka `PriorityQueue` jest pusta, i `false` w przeciwnym przypadku,
— `push(PriorityQueue,e,p)` – wstawia do kolejki `PriorityQueue` element `e` o priorytecie `p`,
— `pop(PriorityQueue)` – usuwa z kolejki `PriorityQueue` element o najwyższym priorytecie,

— `top(PriorityQueue)` – zwraca wartość elementu kolejki `PriorityQueue` o najwyższym priorytecie.

Co prawda kolejka priorytetowa nie jest po prostu rodzajem zwykłej listy, ale możemy ją zaimplementować używając listy. W niniejszym rozdziale zobaczymy dwie implementacje kolejki priorytetowej. Pierwszą, w której elementy kolejki są przechowywane w nieuporządkowanej liście (pozwala to na szybkie wstawianie elementów do kolejki, ale wymaga więcej czasu przy wyszukiwaniu elementu o największym priorytecie i jego usuwaniu) oraz drugą, w której kolejkę przechowujemy w liście posortowanej względem priorytetów (w tej wersji więcej czasu zabiera wstawianie elementów do kolejki, ale za to operacje `top` i `pop` możemy wtedy wykonać w czasie stałym). W rozdziale 7.3.2 poznamy najbardziej popularną implementację kolejki priorytetowej – implementację przy pomocy kopców.

Pierwszą zobaczymy implementację kolejki przy pomocy listy nieposortowanej. W tej implementacji użyjemy lekko zmodyfikowanej funkcji `Najwiekszy` z listingu 2.43 oraz typu `Element_kol`, który będzie strukturą zawierającą zarówno przechowywany element, jak i jego priorytet.

Listing 2.70. Nagłówek klasy PriorityQueueUnsorted

```
1  class PriorityQueueUnsorted {
   private:
3    struct Element_kol{
       Element el;
5      unsigned int priorytet;
       Element_kol(Element e, unsigned int p):el(e),
7                                          priorytet(p){}
     };
9    List Lista;
     Pozycja Najwiekszy();
11 public:
     bool empty();
13   void push(Element e, unsigned int p);
     Element top();
15   void pop();
   };
```

Zakładamy, że elementy prywatnego pola `Lista` będą typu `Element_kol`. Prywatna metoda `Najwiekszy` będzie różniła się od funkcji o tej samej nazwie z listingu 2.43 tym, że będzie szukała największego elementu zawsze tej samej listy przechowywanej w polu `Lista` i że przy wyszukiwaniu będzie brała pod uwagę wyłącznie wartość pola `priorytet`:

Listing 2.71. Operacja Największy

```
Pozycja PriorityQueueUnsorted::Najwiekszy(){
  Pozycja pom, max;
  pom = max = Wejscie.first();
  while(pom != Wejscie.last()){
    pom = Wejscie.next(pom)
    if (Wejscie.at(max).priorytet<Wejscie.at(pom).priorytet)
      max = pom;
  }
  return max;
}
```

Mając zaimplementowaną metodę `Najwiekszy` implementacja pozostałych metod jest prosta:

Listing 2.72. Metody klasy PriorityQueueUnsorted

```
  bool PriorityQueueUnsorted::empty() {
    return Lista.empty();
  }

  void PriorityQueueUnsorted::push(Element e,
                                   unsigned int p){
    Lista.push_back(Element_kol(e,p));
  }

  Element PriorityQueueUnsorted::top(){
    return Lista.at(Najwiekszy()).el;
  }

  void PriorityQueueUnsorted::pop(){
    Lista.erase(Najwiekszy());
  }
```

Złożoność poszczególnych operacji w powyższej implementacji kolejki priorytetowej jest podobna dla typowych implementacji listy. Operacje `top` i `pop` mają złożoność liniową zarówno przy tablicowej implementacji listy, jak i w implementacji listy przy pomocy listy wskaźnikowej (o ile, w tej implementacji pamiętamy wskaźnik na ostatni element listy, w innym przypadku metoda `Najwiekszy`, a co za tym idzie operacje `top` i `pop`, mają złożoność kwadratową). Operacja `push` ma pesymistyczną złożoność stałą w przypadku list wskaźnikowych i liniową w przypadku list implementowanych przy pomocy tablic. W tym drugim przypadku złożoność zamortyzowana operacji `push` jest stała.

Jako drugą przedstawimy implementację kolejki priorytetowej przy wykorzystaniu posortowanej listy. Deklaracja klasy będzie podobna jak w przy-

padku posortowanej listy, z tą różnicą, że tym razem nie będziemy potrzebowali pomocniczej funkcji `Najwiekszy`.

Listing 2.73. Nagłówek klasy PriorityQueueSorted

```
class PriorityQueueSorted {
private:
  struct Element_kol{
    Element el;
    unsigned int priorytet;
    Element_kol(Element e, unsigned int p):el(e),
                                    priorytet(p){}
  };
  List Lista;
public:
  bool empty();
  void push(Element e, unsigned int p);
  Element top();
  void pop();
};
```

Zakładamy, że zmienna `Lista` zawiera elementy kolejki priorytetowej uporządkowane nierosnąco ze względu na priorytety. W takiej sytuacji operacje `top` i `pop` są bardzo proste:

Listing 2.74. Operacja empty, top i pop

```
  bool PriorityQueueSorted::empty(){
    return Lista.empty();
  }

  Element PriorityQueueSorted::top(){
    return Lista.front();
  }

  void PriorityQueueSorted::pop(){
    Lista.pop_front();
  }
```

W operacji `push` możemy wykorzystać, po niewielkich zmianach, fragment kodu algorytmu sortowania przez wstawianie:

Listing 2.75. Operacja push

```
void PriorityQueueSorted::push(Element e, unsigned int p){
  if (e >= Lista.back().priorytet)
    Lista.push_back(Element_kol(e,p));
  else {
    Pozycja poz = Lista.first();
    while(p < Lista.at(poz).priorytet))
```

```
7        poz = Lista.next(poz);
       Lista.insert(poz, Element_kol(e,p));
9    }
   }
```

Ze względu na konieczność przejrzenia w najgorszym przypadku całej listy, przy dowolnej z poznanych implementacji listy pesymistyczna złożoność operacji `push` jest liniowa. Mniejszą złożoność mają operacje `empty`, `top` i `pop`. Przy implementacji listy za pomocą listy wskaźnikowej lub w „zawijanej" implementacji tablicowej złożoność tych operacji to $O(1)$.

Kolejkę priorytetową można wykorzystać do sortowania listy. Wystarczy umieścić wszystkie elementy listy w kolejce priorytetowej z priorytetami równymi ich wartości i potem pobierać je w posortowanej kolejności operacją `top`. Jeżeli do sortowania użyjemy kolejki priorytetowej zaimplementowanej przy pomocy listy nieuporządkowanej, to jako wynik dostaniemy algorytm podobny do algorytmu sortowania przez wybieranie. Jeżeli zaś użyjemy kolejki priorytetowej zaimplementowanej przy pomocy listy posortowanej, to w efekcie dostaniemy algorytm przypominający algorytm sortowania przez wstawianie.

Zadanie 11. Zaimplementuj kolejkę priorytetową nie wykorzystując gotowych implementacji listy.

Zadanie 12. Zaimplementuj funkcję sortującą podaną w argumencie listę przy pomocy kolejki priorytetowej.

Rozdział 3

Przeszukiwanie wyczerpujące

3.1. Przeszukiwanie wszystkich możliwości **58**
3.1.1. Elementy iloczynu kartezjańskiego 58
3.1.2. Podzbiory 63
3.1.3. Permutacje 68
3.2. Przeszukiwanie z nawrotami **72**

Wszelkie dane są zapisywane na komputerach w postaci ciągów zer i jedynek. W szczególności rozwiązania wszelkich problemów obliczeniowych to z punktu widzenia komputera skończone ciągi zer i jedynek, do tego zwykle o ograniczonym z góry rozmiarze. Stąd pomysł, żeby spróbować rozwiązywać problemy obliczeniowe poprzez sprawdzenie wszystkich możliwych ciągów zer i jedynek nie dłuższych niż `k`, gdzie `k` jest pewną wartością zależną od rozmiaru danych wejściowych i struktury problemu. Ważne, żebyśmy potrafili szybko sprawdzić, czy dany ciąg zer i jedynek jest rozwiązaniem danego problemu.

Choć rozwiązywanie problemów algorytmicznych poprzez implementację przeszukiwania wszystkich możliwości może wydawać się bardzo nieefektywne, to w wielu przypadkach nie znamy istotnie lepszego rozwiązania. Tym bardziej, że często poprzez analizę problemu jesteśmy w stanie znacznie ograniczyć zbiór potencjalnych rozwiązań, które musimy rozważyć.

3.1. Przeszukiwanie wszystkich możliwości

Umiejętność generowania wszystkich potencjalnych rozwiązań jest ważną częścią warsztatu każdego programisty. W tym rozdziale przyjrzymy się temu, jak można generować podstawowe obiekty kombinatoryczne.

3.1.1. Elementy iloczynu kartezjańskiego

Chyba najprostszymi obiektami kombinatorycznymi są elementy iloczynu kartezjańskiego. Iloczyn kartezjański dwóch zbiorów X oraz Y oznaczymy przez $X \times Y$ i definiujemy jako zbiór wszystkich par (x, y) takich, że $x \in X$ i $y \in Y$. Iloczyn trzech zbiorów $X \times Y \times Z$ definiujemy jako złożenie dwóch iloczynów kartezjańskich $X \times (Y \times Z)$ itd.

Aby wygenerować wszystkie liczby całkowite z zakresu domkniętego od `p` do `k` najprościej jest użyć pętli `for` takiej jak w poniższym listingu.

Listing 3.1. Generowanie wszystkich liczb z określonego zakresu

```
for(int i=p; i<=k;i++){

  //tutaj jest miejsce na zrobienie z i tego
  //co chcemy zrobic
  //z kazda liczba z zakresu od p do k

}
```

Jeżeli chcemy wygenerować wszystkie możliwe pary liczb, takie że pierwsza liczba jest z domkniętego przedziału od `p1` do `k1`, zaś druga liczba jest z zakresu od `p2` do `k2`, to możemy to zrobić za pomocą pętli w pętli:

Listing 3.2. Generowanie wszystkich par

```
1 for(int i=p1; i<=k1;i++)
    for(int j=p2;j<k2;j++){
3

5     //tutaj jest miejsce na zrobienie z para (i,j)
      //tego co chcemy zrobic
7     //z kazda para liczb z odpowiednich zakresow
    }
```

Stosując dalej tą metodę, chcąc generować wszystkie elementy iloczynu kartezjańskiego k zbiorów, musimy użyć k pętli. Dla dużych k jest to niewygodne, a w przypadku, gdy k nie jest stałą znaną w momencie pisania programu, to użycie powyższej metody jest niemożliwe.

Innym sposobem generowania wszystkich elementów iloczynu kartezjańskiego jest zdefiniowanie funkcji, która dla podanego w argumencie elementu iloczynu kartezjańskiego generuje następny element w jakiejś ustalonej kolejności. W naszym przypadku będzie to kolejność leksykograficzna, czyli uogólniona kolejność alfabetyczna, w której porównując dwa ciągi najpierw porównujemy ich pierwsze elementy, w przypadku, gdy są równe, porównujemy drugie elementy itd.

Stworzymy dwie funkcje, `Pierwszy`, która nadaje podanej w argumencie zmiennej wartość elementu pierwszego w kolejności leksykograficznej i `Nastepny`, która zmienia wartość podanej w argumencie zmiennej na kolejną w kolejności leksykograficznej. Dla ostatniej leksykograficznie wartości funkcja `Nastepny` wygeneruje pierwszą leksykograficznie wartość. Zacznijmy od najprostszego przypadku, czyli od pojedynczej zmiennej. Zakładamy, że przyjmuje ona wartości z przedziału od `p` do `k` oraz, że `p` i `k` są zmiennymi globalnymi.

Listing 3.3. Generowanie wszystkich liczb z zakresu

```
  void Pierwszy(int& wart){
2   wart = p;
  }
4
  void Nastepny(int& wart){
6   if (wart == k)
      wart = p;
8   else
      wart++;
10 }
```

W powyższym programie użyliśmy referencji, aby zmienić wartość zmiennej podanej jako argument. Bardziej naturalnym rozwiązaniem byłoby zapew-

ne, gdyby funkcje `Pierwszy` i `Nastepny` po prostu zwracały wartość odpowiednio pierwszego lub następnego elementu przedziału. Jednak w przypadku używania struktur z dynamicznie alokowanymi tablicami (a będziemy za chwilę takich używać), wymagałoby to od nas przeciążania operatora przypisania.

Teraz zaimplementujemy funkcje, które można wykorzystać do generowania wszystkich par elementów z dwóch przedziałów. Do przechowywania par będziemy potrzebowali specjalnego typu.

Listing 3.4. Typ do przechowywania pary

```
struct para{
  int pierwsza, druga, p1, p2, k1, k2;
};
```

Pola `p1`, `p2`, `k1` i `k2` oznaczają odpowiednio początki i końce przedziału, z którego pochodzą pierwsza i druga wartość pary.

Ponieważ będziemy generować pary w porządku leksykograficznym, to pierwszą parą będzie para początkowych wartości obu przedziałów:

Listing 3.5. Generowanie wszystkich par

```
void Pierwszy(para& wart){
  wart.pierwsza = wart.p1;
  wart.druga = wart.p2;
}
```

Generując następną parę będziemy używali algorytmu podobnego do algorytmu inkrementacji liczby zapisanej w dowolnym systemie pozycyjnym. Najpierw będziemy próbowali zwiększyć o jeden drugą liczbę.

Listing 3.6. Generowanie wszystkich par

```
void Nastepny(para& wart){
  if (wart.druga < wart.k2)
    wart.druga++;

  ...

}
```

Jeżeli nie możemy zwiększyć drugiej liczby pary, gdyż ma wartość `k2`, to nadajemy jej wartość `p2` i próbujemy zwiększyć o jeden pierwszą liczbę.

Listing 3.7. Generowanie wszystkich par

```
void Nastepny(para& wart){
```

```
  if (wart.druga < wart.k2)
3   wart.druga++;
  else {
5   wart.druga = wart.p2;
    if (wart.pierwsza < wart.k1)
7     wart.pierwsza++;

9   ...

11  }
}
```

Jeżeli pierwsza liczba też ma maksymalną dopuszczalną wartość, to nadajemy jej wartość `wart.p1` (to jest przypadek, gdy jako argument otrzymaliśmy ostatnią leksykograficznie parę i generujemy parę pierwszą leksykograficznie).

Listing 3.8. Generowanie wszystkich par

```
void Nastepny(para& wart){
2   if (wart.druga < wart.k2)
      wart.druga++;
4   else {
      wart.druga = wart.p2;
6     if (wart.pierwsza < wart.k1)
        wart.pierwsza++;
8     else
        wart.pierwsza = wart.p1;
10  }
}
```

Jako ostatni rozpatrzymy przypadek generowania elementów iloczynu kartezjańskiego n zbiorów. Podobnie jak w przypadku generowania par, zdefiniujemy nowy typ do przechowywania elementów iloczynu kartezjańskiego.

Listing 3.9. Typ do przechowywania elementów iloczynu kartezjańskiego

```
1 struct element{
    int *wspolrzedne, * poczatki, * konce, n;
3 };
```

Tym razem współrzędne oraz początki i końce przedziałów, do których mają należeć poszczególne współrzędne, będziemy przechowywali w tablicach. Pole `n` będzie przechowywało liczbę współrzędnych.

Funkcja `Pierwszy`, tak jak w przypadku par będzie nadawała wszystkim współrzędnym najmniejsze dopuszczalne wartości.

Listing 3.10. Generowanie elementów iloczynu kartezjańskiego

```
void Pierwszy(element& wart){
  for(int i=0;i<wart.n;i++)
    wart.wspolrzedne[i] = wart.poczatki[i];
}
```

Funkcja Nastepny, podobnie jak jej wersja dla par, będzie działała analogicznie do inkrementacji liczb zapisanych w dowolnym systemie pozycyjnym. Będziemy przeglądali od tyłu wartości poszczególnych współrzędnych. Będziemy to robić tak długo, dopóki nie znajdziemy współrzędnej, której wartość jest mniejsza od maksymalnej możliwej na tej współrzędnej lub dopóki nie przejrzymy wszystkich współrzędnych.

Listing 3.11. Generowanie elementów iloczynu kartezjańskiego

```
void Nastepny(element& wart){
  unsigned int i = wart.n-1;
  for (;(i>=0)&&(wart.wspolrzedne[i]==wart.konce[i]);i--)

  ...

}
```

Kolejność warunków w pętli for nie jest przypadkowa. Jeżeli `i` będzie mniejsze od 0, to zgodnie z zasadami działania operator `&&` w języku C++, druga cześć warunku nie zostanie sprawdzona (nie ma więc ryzyka, że będziemy próbowali sprawdzić wartość elementów tablicy o indeksach spoza zakresu).

Przeglądając współrzędne osiągające wartości z końców przedziałów, będziemy im nadawali wartości początków przedziałów (podobnie jak dodając 1 do zapisanej w systemie dziesiętnym liczby 129999 zamieniamy wszystkie znajdujące się na końcu dziewiątki na zera).

Listing 3.12. Generowanie elementów iloczynu kartezjańskiego

```
void Nastepny(element& wart){
  unsigned int i = wart.n-1;
  for (;(i>=0)&&(wart.wspolrzedne[i]==wart.konce[i]);i--)
    wart.wartosci[i]=wart.poczatki[i];

  ...

}
```

Na koniec, jeżeli znaleźliśmy współrzędną, której wartość możemy zwiększyć, to robimy to.

Listing 3.13. Generowanie elementów iloczynu kartezjańskiego

```
void Nastepny(element& wart){
  unsigned int i = wart.n-1;
  for(;(i>=0)&&(wart.wspolrzedne[i]==wart.konce[i]);i--)
    wart.wartosci[i]=wart.poczatki[i];
  if (i>=0)
    wart.wspolrzedne[i]++;
}
```

Zwróćmy uwagę, że w powyższej funkcji zmienną `i` celowo zadeklarowaliśmy przed pętlą `for` (a nie wewnątrz tej pętli). Dzięki temu mogliśmy odczytać jej wartość także po zakończeniu działania pętli `for`.

Pesymistyczna złożoność czasowa funkcji `Nastepny` to $O($`wart.n`$)$ (tyle razy w pesymistycznym przypadku obróci się pętla `for`). Prosta analiza podobna do tej, której dokonywaliśmy już szacując złożoności operacji na listach pokazuje, że zamortyzowana złożoność powyższej funkcji jest stała.

Zadanie 13. Uprość listingi 3.9, 3.10 i 3.13, tak żeby funkcje `Pierwszy` i `Nastepny` generowały iloczyn kartezjański n zbiorów $\{0, 1, \ldots, k-1\}$, gdzie k i n są polami poprawionej struktury `element`.

Zadanie 14. Napisz funkcję, która dla otrzymanych w argumencie tablicy dwuwymiarowej `tab` o wymiarach 3×3 oraz liczby n, szuka takiego ciągu liczb `x(1)`, `x(2)`, ..., `x(n)`, żeby spełniony był następujący warunek:

```
tab[... tab[tab[x(1)][x(2)]][x(3)]...][x(n)]=
=tab[x(1)][tab[x(2)][...tab[x(n-1)][x(n)]...]]
```

3.1.2. Podzbiory

Innymi podstawowymi obiektami kombinatorycznymi przydatnymi w informatyce są podzbiory. Od elementów iloczynu kartezjańskiego odróżniają je trzy podstawowe różnice, które będą miały wpływ na sposób ich generowania:

- — wszystkie elementy podzbioru pochodzą z tego samego zbioru,
- — kolejność elementów podzbioru nie ma znaczenia ($(0, 1)$ i $(1, 0)$ to dwie różne pary, ale $\{0, 1\}$ i $\{1, 0\}$ to dwa sposoby opisu tego samego zbioru),
- — każdy element może występować w podzbiorze co najwyżej jeden raz ($(1, 1)$ jest poprawną parą, ale $\{1, 1\}$ to niepoprawny sposób zapisu zbioru $\{1\}$).

Pierwsza z powyższych własności uprości nam generowanie wszystkich podzbiorów. My uprościmy ją sobie jeszcze bardziej poprzez założenie, że generujemy podzbiory przedziału domkniętego od 0 do $k-1$, gdzie k jest dowolną

wartością większą od 0. Przypadek podzbiorów dowolnego przedziału liczb całkowitych zostawiamy czytelnikowi jako ćwiczenie.

Zaczniemy od podzbiorów dwuelementowych. Podobnie jak miało to miejsce w przypadku par, dwuelementowe podzbiory najprościej jest wygenerować przy pomocy dwu pętli. Aby zagwarantować sobie, że nie rozważamy wielokrotnie tego samego podzbioru, zmienna sterująca wewnętrznej pętli będzie zawsze większa od zmiennej sterującej zewnętrznej pętli (osiągamy to dzięki temu, że dla każde wartości `i` inicjujemy `j` wartością `i+1`):

Listing 3.14. Generowanie dwuelementowych podzbiorów

```
1 for(int i=0; i<k-1;i++)
    for(int j=i+1;j<k;j++){
3     //tutaj jest miejsce na zrobienie z podzbiorem {i,j}
      //tego co chcemy zrobic
5     //z kazdym dwuelementowym podzbiorem
      //zbioru {0,1,...,k-1}
7   }
```

Gdybyśmy chcieli generować w ten sposób podzbiory o większej liczbie elementów, musielibyśmy zagnieździć w sobie większą liczbę pętli. Oczywiście jest to możliwe tylko wtedy, gdy liczność generowanych podzbiorów jest znana przed kompilacją programu.

Teraz będziemy generować dwuelementowe podzbiory używając funkcji `Pierwszy`, `Nastepny` oraz specjalnej struktury służącej do przechowywania dwuelementowych podzbiorów. Najpierw zdefiniujemy strukturę, która będzie zawierała pola przechowujące pierwszy i drugi element zbioru oraz ograniczenie k na koniec przedziału, z którego mają być elementy podzbioru.

Listing 3.15. Struktura do przechowywania dwuelementowych podzbiorów

```
1 struct podzbior2el {
    unsigned int pierwszy, drugi, k;
3 }
```

Pierwszym podzbiorem będzie podzbiór $\{0, 1\}$. Będziemy porządkowali elementy zbioru ze względu na ich wartość. W związku z tym w pierwszym zbiorze pierwszym elementem będzie `0`, a drugim `1`.

Listing 3.16. Pierwszy spośród dwuelementowych podzbiorów

```
1 void Pierwszy (podzbior2el& podzbior) {
    podzbior.pierwszy = 0;
3   podzbior.drugi = 1;
  }
```

Funkcja generująca następny podzbiór w pierwszej kolejności będzie próbowała zwiększyć drugi element podzbioru.

Listing 3.17. Generowanie dwuelementowych podzbiorów

```
  void Nastepny (podzbior2el& podzbior) {
2   if (podzbior.drugi<podzbior.k-1)
      podzbior.drugi++;
4
    ...
6 }
```

Jeżeli drugi element podzbioru będzie miał już maksymalną możliwą wartość, spróbujemy zwiększyć pierwszy element zbioru, a drugiemu elementowi zbioru nadać wartość o jeden większą od pierwszego.

Listing 3.18. Generowanie dwuelementowych podzbiorów

```
  void Nastepny (podzbior2el& podzbior) {
2   if (podzbior.drugi<podzbior.k-1)
      podzbior.drugi++;
4   else if (podzbior.pierwszy<podzbior.k-2){
      podzbior.pierwszy++;
6     podzbior.drugi = podzbior.pierwszy+1;
    }
8   ...
  }
```

Zauważmy, że pierwszy element podzbioru może mieć wartość co najwyżej `k-2`. Wynika to z faktu, że drugi element zbioru ma być większy od pierwszego, a więc gdyby pierwszy element był równy $k-1$, to drugi musiałby być równy k, a ta wartość nie należy do zbioru, którego podzbiory generujemy.

W przypadku gdy oba elementy otrzymanego podzbioru mają swoje maksymalne dopuszczalne wartości, funkcja `Nastepny` jako następny wygeneruje pierwszy podzbiór.

Listing 3.19. Generowanie dwuelementowych podzbiorów

```
1 void Nastepny (podzbior2el& podzbior) {
    if (podzbior.drugi<podzbior.k-1)
3     podzbior.drugi++;
    else if (podzbior.pierwszy<podzbior.k-2){
5     podzbior.pierwszy++;
      podzbior.drugi = podzbior.pierwszy+1;
7   }
    else
9     Pierwszy(podzbior);
  }
```

Na koniec pokażemy funkcje służące do generowania podzbiorów o dowolnej liczbie elementów. Do tego celu zdefiniujemy typ `podzbiornel`, w którym będziemy przechowywali elementy podzbioru, liczność podzbioru oraz górne ograniczenie na wartość elementów podzbiorów.

Listing 3.20. Struktura do przechowywania dowolnych podzbiorów

```
struct podzbiornel{
  unsigned int * elementy,
  unsigned int n; //licznosc podzbioru
  unsigned int k; //ograniczenie wartosci elementow podzb.
}
```

Analogicznie do rozwiązania, które zastosowaliśmy w przypadku podzbiorów dwuelementowych, elementy podzbioru o dowolnej liczności będziemy przechowywali w posortowanej rosnąco tablicy. Dzięki temu, że tablica będzie rosnąco posortowana, będziemy mieli pewność, że elementy tablicy się nie powtarzają i że nie rozważamy dwa razy tego samego podzbioru tylko inaczej ułożonego w tablicy (każdej rosnąco uporządkowanej tablicy odpowiada inny podzbiór). Pierwszym podzbiorem będzie podzbiór zawierający n najmniejszych elementów zbioru $\{0, 1, \ldots, k-1\}$.

Listing 3.21. Pierwszy spośród podzbiorów

```
void Pierwszy(podzbiornel& podzbior) {
  for(unsigned int =0;i<podzbior.n;i++)
    podzbior.elementy[i] = i;
}
```

Generując następnik podanego w argumencie elementu, w pierwszym kroku odnajdziemy pierwsze miejsce, licząc od tyłu, w którym nie znajduje się największa dopuszczalna w danym miejscu wartość.

Listing 3.22. Generowanie podzbiorów o dowolnej ustalonej liczności

```
void Nastepny(podzbiornel& pdzb) {
   unsigned int i=1;
  for(;(i<=pdzb.n)&&(pdzb.elementy[pdzb.n-i]==pdzb.k-i);
                                                    i++);

  ...

}
```

W przypadku gdy znajdziemy taki element, zwiększamy go o jeden, zaś elementom znajdującym się za nim nadajemy najmniejsze możliwe war-

tości, przy których tablica elementów będzie uporządkowana rosnąco (czyli kolejne elementy będą o jeden większe od swoich poprzedników).

Listing 3.23. Generowanie podzbiorów o dowolnej ustalonej liczności

```
void Nastepny(podzbiornel& pdzb) {
   unsigned int i=1;
  for(;(i<=pdzb.n)&&(pdzb.elementy[pdzb.n-i]==pdzb.k-i);
                                                          i++);
  if (i<=pdzb.n) {
    pdzb.elementy[pdzb.n-i]++;
    for(i=i-1;i>0;i--)
      pdzb.elementy[pdzb.n-i]=pdzb.elementy[pdzb.n-i-1]+1;
  }

  ...

}
```

Na koniec jeżeli okaże się, że wszystkie elementy osiągają swoją maksymalną dopuszczalną wartość, generujemy pierwszy podzbiór.

Listing 3.24. Generowanie podzbiorów o dowolnej ustalonej liczności

```
void Nastepny(podzbiornel& pdzb) {
   unsigned int i=1;
  for(;(i<=pdzb.n)&&(pdzb.elementy[pdzb.n-i]==pdzb.k-i);
                                                          i++);
  if (i<=pdzb.n) {
    pdzb.elementy[pdzb.n-i]++;
    for(i=i-1;i>0;i--)
      pdzb.elementy[pdzb.n-i]=pdzb.elementy[pdzb.n-i-1]+1;
  }
  else
    Pierwszy(pdzb);
}
```

Pesymistyczna złożoność czasowa funkcji `Nastepny` to $O($`pdzb.n`$)$. Wynika to z faktu, że powyższy algorytm składa się z dwóch niezależnych pętli, z których każda wykona co najwyżej `pdzb.n` obrotów oraz wywołania funkcji `Pierwszy` o złożoności $O($`pdzb.n`$)$. Szacując nieco dokładniej można pokazać, że zamortyzowany czas operacji `Nastepny` jest stały. Jak często w takich sytuacjach wynika to z faktu, że najczęściej będziemy zmieniać tylko ostatni element podzbioru, rzadziej przedostatni itd.

Zadanie 15. Napisz funkcję generującą kolejne podzbiory dwuelementowe domkniętego przedziału liczb całkowitych od p do k, gdzie p i k

mogą przyjmować dowolne wartości całkowitoliczbowe, takie że $p < k$.

Zadanie 16. Napisz funkcję generującą kolejne podzbiory o dowolnej zadanej liczbie elementu domkniętego przedziału liczb całkowitych od p do k, gdzie p i k mogą przyjmować dowolne wartości całkowitoliczbowe, takie że $p < k$.

Zadanie 17. Wymyśl i zaimplementuj sposób generowania wszystkich podzbiorów podanego zbioru (do tej pory generowaliśmy tylko podzbiory o ustalonej liczności).

3.1.3. Permutacje

Permutacja to formalnie wzajemnie jednoznaczne przekształcenie zbioru na samego siebie (różnowartościowa funkcja $f : A \to A$ taka, że $f(A) = A$). Każda taka funkcja na zbiorze skończonym może być utożsamiona z pewnym liniowym porządkiem na elementach tego zbioru. Dla nas permutacje będą właśnie liniowymi porządkami na zbiorach, czyli inaczej mówiąc sposobami ustawienia w ciąg wszystkich elementów zbioru.

W tym rozdziale będziemy rozważać permutacje zbioru $\{0, 1, \ldots, n-1\}$, gdzie n jest dowolną dodatnią liczba całkowitą. Permutacje będziemy przechowywali w tablicy, w której każdy element zbioru będzie występował dokładnie raz, a kolejność elementów zbioru w tablicy będzie wyznaczała permutację. Zdefiniujemy typ do przechowywania permutacji:

Listing 3.25. Typ do przechowywania permutacji

```
struct permutacja {
  unsigned int * elem, n;
};
```

Permutacje będziemy generować w kolejności leksykograficznej, więc pierwszą permutacją będzie taka, w której elementy zbioru są uporządkowane rosnąco.

Listing 3.26. Funkcja generująca pierwszą permutację

```
void Pierwsza(permatucja & perm){
  for(unsigned int i=0;i<perm.n;i++)
    perm.elem[i]=i;
}
```

Algorytm generujący kolejną permutację jest nieco bardziej skomplikowany niż dotychczas rozważane algorytmy generujące różne obiekty kombinatoryczne. Słowami można go opisać w następujący sposób:

Algorytm 3.1.

1. *Znajdź maksymalny sufiks (końcowy fragment) permutacji uporządkowany malejąco.*
2. *Jeżeli znaleziony sufiks jest równy całej permutacji, to wygeneruj pierwszą permutację i zakończ działanie.*
3. *Niech* e *będzie elementem zbioru bezpośrednio poprzedzającym znaleziony sufiks.*
4. *Znajdź najmniejszy element sufiksu większy od* e *i zamień go miejscami z* e.
5. *Odwróć kolejność elementów sufiksu.*

Przyjrzyjmy się na przykładowi działania powyższego algorytmu.

Przykład 3.1.
Załóżmy, że generujemy permutację następującą po permutacji

$$\mathrm{a} = (2, 1, 4, 3, 0).$$

Najpierw znajdujemy maksymalny uporządkowany malejąco sufiks a*:*

$$(2, 1, \mathbf{4}, \mathbf{3}, \mathbf{0}).$$

Liczba 1 *do tego sufiksu nie należy gdyż* 4 *jest większe od* 1*. Następnie szukamy najmniejszego elementu sufiksu większego od elementu bezpośrednio poprzedzającego sufiks. W naszym przypadku bezpośrednio przed sufiksem znajduje się* 1*, a więc szukamy najmniejszego elementu sufiksu większego od* 1*. W rozważanym przypadku jest to liczba* 3*:*

$$(2, 1, \mathbf{4}, 3, \mathbf{0}).$$

Teraz zamieniamy miejscami 1 *i* 3*:*

$$(2, 3, \mathbf{4}, 1, \mathbf{0})$$

oraz odwracamy kolejność elementów sufiksu:

$$(2, 3, \mathbf{0}, \mathbf{1}, \mathbf{4}).$$

W ten sposób otrzymaliśmy w wyniku permutację kolejną w porządku leksykograficznym względem a*.*

Omawiany algorytm generujący kolejne permutacje zaimplementujemy krok po kroku. Zaczniemy od odnalezienia maksymalnego uporządkowanego malejąco sufiksu.

Listing 3.27. Funkcja generująca następną permutację

```
void Nastepna(permutacja & perm){
  int i,j;
  for(i=perm.n-2;(i>=0)&&(perm.elem[i+1]<perm.elem[i]);i--);

  ...

}
```

Jeżeli `i` po przejściu pętli jest mniejsze od zera, to oznacza, że cała permutacja jest posortowana malejąco i mamy wygenerować pierwszą permutację.

Listing 3.28. Funkcja generująca następną permutację

```
void Nastepna(permutacja & perm){
  int i,j;
  for(i=perm.n-2;(i>=0)&&(perm.elem[i+1]<perm.elem[i]);i--);
  if (i<0){
    Pierwsza(perm);
  }
  else {

    ...

  }
}
```

Jeżeli `i` jest większe lub równe zeru, to szukamy najmniejszego elementu prefiksu większego lub równego od elementu `perm.elem[i]`, po czym zamieniamy miejscami oba wspomniane elementy.

Listing 3.29. Funkcja generująca następną permutację

```
void Nastepna(permutacja & perm){
  int i,j;
  for(i=perm.n-2;(i>=0)&&(perm.elem[i+1]<perm.elem[i]);i--);
  if (i<0){
    Pierwsza(perm);
  }
  else {
    for(j=perm.n-1;(perm.elem[j]<perm.elem[i]);j--);
    Zamien(perm.elem[i],perm.elem[j]);

    ...

  }
}
```

Funkcję `Zamien` zaimplementujemy na końcu.

To, co nam zostało do zrobienia w funkcji `Nastepna`, to odwrócenie kolejności elementów należących do prefiksu.

Listing 3.30. Funkcja generująca następną permutację

```
void Nastepna(permutacja & perm){
  int i,j;
  for(i=perm.n-2;(i>=0)&&(perm.elem[i+1]<perm.elem[i]);i--);
  if (i<0){
    Pierwsza(perm);
  }
  else {
    for(j=perm.n-1;(perm.elem[j]<perm.elem[i]));j--);
    Zamien(perm.t[i],perm.t[j]);
    for(unsigned int k=1;k<=(perm.n-i-1)/2;k++)
      Zamien(perm.elem[i+k],perm.elem[perm.n-k]);
  }
}
```

Na koniec zaimplementujemy jeszcze funkcję `Zamien`:

Listing 3.31. Funkcja zamieniające wartości

```
void Zamien(unsigned int & a, unsigned int & b){
  unsigned int pom = a;
  a = b;
  b = pom;
}
```

Łatwo pokazać, że w pesymistycznym przypadku funkcja `Nastepna` z listingu 3.30 wykonuje $O($`perm.n`$)$ operacji. Wynika to z faktu, że wewnątrz funkcji `Nastepna` znajdują się trzy niezależne pętle `for`, z których każda może wykonać co najwyżej `perm.n` obrotów. Uważnego czytelnika nie zdziwi też informacja, że zamortyzowana złożoność powyższej funkcji to $O(1)$. Tradycyjnie wynika to z faktu, że co drugie wywołanie funkcji `Nastepna` zamieni tylko dwa ostatnie elementy w permutacji, co czwarte wywołanie funkcji zamieni 3 ostatnie elementy itd.

Zadanie 18. Napisz program, który wczytuje ze standardowego wejścia liczbę n oraz n różnych imion i wypisuje na standardowym wyjściu wszystkie możliwe permutacje wczytanych na wejściu imion.

3.2. Przeszukiwanie z nawrotami

Przeszukiwanie z nawrotami (ang. *backtracking*) to algorytm, który stopniowo generuje kandydatów do bycia rozwiązaniem problemu (kolejne coraz pełniejsze potencjalne „częściowe rozwiązania" problemu). Gdy na jakimś etapie generowania potencjalnego rozwiązania algorytm odkryje, że dana ścieżka obliczeń nie prowadzi do poprawnego rozwiązania, to cofa się krok wstecz i próbuje innej możliwości. Przeszukiwanie z nawrotami w naturalny sposób implementuje się przy pomocy funkcji rekurencyjnej. Na każdym poziomie rekurencji generujemy kolejną część rozwiązania. W przypadku, gdy w którymś rekurencyjnym wywołaniu funkcji okazuje się, że konstruowane potencjalne rozwiązanie nie prowadzi do poprawnego rozwiązania problemu (mówiąc nieformalnie gdy zabrniemy w ślepą uliczkę), dane wywołanie funkcji kończy swoje działanie, co automatycznie cofa nas na wyższy poziom rekurencji. Implementację algorytmu przeszukiwania z nawrotami przeanalizujemy na przykładzie problemu ustawiania hetmanów na szachownicy:

Problem 3.1. *Problem ustawienia hetmanów na szachownicy to problem, w którym na wejściu otrzymujemy liczbę n, a na wyjściu wypisujemy sposób ustawienia n hetmanów na szachownicy $n \times n$ tak, żeby żadne dwa hetmany się nie szachowały lub informację, że takie ustawienie nie istnieje.*

W naszym rozwiązaniu problemu hetmanów użyjemy rekurencyjnej implementacji backtrackingu. Zauważmy, że jeżeli na szachownicy $n \times n$ chcemy ustawić n hetmanów, tak żeby się wzajemnie nie szachowały, to musimy postawić po jednym hetmanie w każdej linii. Wykorzystując ten fakt na każdym kolejnym poziomie rekurencji będziemy ustawiali hetmana w linii o numerze odpowiadającym poziomowi rekurencji w taki sposób, żeby nie szachował się z wcześniej postawionymi hetmanami. Pola, na których stoją hetmany, będą przechowywane w tablicy `tab_hetmany`. Komórka o indeksie `i` będzie zawierała numer kolumny, w której stoi hetman w linii o numerze `i`. Nasza funkcja będzie zwracała wartość logiczną `true`, jeżeli da się ustawić hetmany zgodnie z wymaganiami zadania i `false` w przeciwnym wypadku. Jeżeli wartość funkcji będzie `true`, to tablica `tab_hetmany` będzie zawierała poszukiwane ustawienie hetmanów.

Implementowana rekurencyjna funkcja `hetmany_rek` otrzyma jako argumenty stan planszy (a więc tablicę `tab_hetman`), jej rozmiar oraz poziom rekurencji, na którym się znajduje i będzie zwracała wartość logiczną informującą, czy udało się znaleźć odpowiednie ustawienie hetmanów.

Listing 3.32. Rozwiązanie problemu ustawienia hetmanów

```
bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
                                        unsigned int poziom_rek){

  ...

} //hetmany
```

Na i-tym poziomie rekurencji będziemy próbowali postawić hetmana w i-tym wierszu. W tym celu będziemy sprawdzali, które z kolejnych pól i-tego wiersza nie są szachowane przez dotychczas postawione hetmany. Do sprawdzania, czy dane pole nie jest szachowane, wykorzystamy funkcję `szachowane`, którą zaimplementujemy później. Funkcja `szachowane` będzie otrzymywać jako argumenty współrzędne pól, na których są ustawione hetmany oraz współrzędne sprawdzanego pola i będzie zwracać `true` wtedy i tylko wtedy, gdy sprawdzane pole jest szachowane przez któregoś z wcześniej postawionych hetmanów.

Listing 3.33. Rozwiązanie problemu ustawienia hetmanów

```
bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
                                        unsigned int poziom_rek){
  for(int i=0;i<n;i++){
    if (!szachowane(tab_hetmany,i,poziom_rek)){

      ...

    } //if
  } //for
} //hetmany
```

Jeżeli sprawdzane pole nie jest szachowane, to stawiamy tam hetmana. Zauważmy, że poziom rekurencji mówi nam, który na liście będzie ten hetman.

Listing 3.34. Rozwiązanie problemu ustawienia hetmanów

```
bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
                                        unsigned int poziom_rek){
  for(int i=0;i<n;i++){
    if (!szachowane(tab_hetmany,i,poziom_rek)){
      tab_hetmany[poziom_rek]=i;

        ...
    } //if
  } //for
} //hetmany
```

Jeżeli postawiliśmy właśnie hetmana w ostatnim wierszu, to znaczy, że znaleźliśmy rozwiązanie i możemy zakończyć poszukiwania.

Listing 3.35. Rozwiązanie problemu ustawienia hetmanów

```
bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
                                unsigned int poziom_rek){
  for(int i=0;i<n;i++){
    if (!szachowane(tab_hetmany,i,poziom_rek)){
      tab_hetmany[poziom_rek]=i;
      if (poziom_rek==n-1)
        return true;

       ...

    } //if
  } //for
} //hetmany
```

W przeciwnym wypadku sprawdzamy, czy da się do obecnego ustawienia tak dostawić hetmany, żeby uzyskać rozwiązanie problemu (robimy to przy pomocy rekurencyjnego wywołania funkcji `hetmany_rek`).

Listing 3.36. Rozwiązanie problemu ustawienia hetmanów

```
bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
                                unsigned int poziom_rek){
  for(int i=0;i<n;i++){
    if (!szachowane(tab_hetmany,i,poziom_rek)){
      tab_hetmany[poziom_rek]=i;
      if (poziom_rek==n-1)
        return true;
      hetmany_rek(tab_hetmany, n, poziom_rek+1);
       ...

    } //if
  } //for
} //hetmany
```

W przypadku gdy rozważane rozwiązanie da się uzupełnić do pełnego rozwiązania, kończymy działanie algorytmu. Jeżeli nie da się uzupełnić aktualnego potencjalnego „częściowego rozwiązania”, szukamy kolejnego nieszachowanego pola w rozważanym na danym poziomie rekurencji wierszu i próbujemy tam postawić hetmana.

Listing 3.37. Rozwiązanie problemu ustawienia hetmanów

```
1 bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
                                   unsigned int poziom_rek){
3   for(int i=0;i<n;i++){
      if (!szachowane(tab_hetmany,i,poziom_rek)){
5       tab_hetmany[poziom_rek]=i;
        if (poziom_rek==n-1)
7         return true;
        if (hetmany_rek(tab_hetmany, n, poziom_rek+1))
9         return true;
      } //if
11  } //for
  } //hetmany
```

Na koniec jeżeli rozważymy wszystkie pola w danym wierszu i postawienie hetmana na żadnym z nich nie doprowadziło nas do sukcesu, to zwracamy wartość `false`.

Listing 3.38. Rozwiązanie problemu ustawienia hetmanów

```
  bool hetmany_rek(unsigned int *tab_hetmany, unsigned int n,
2                                  unsigned int poziom_rek){
    for(int i=0;i<n;i++){
4     if (!szachowane(tab_hetmany,i,poziom_rek)){
        tab_hetmany[poziom_rek]=i;
6       if (poziom_rek==n-1)
          return true;
8       if (hetmany_rek(tab_hetmany, n, poziom_rek+1))
          return true;
10    } //if
    } //for
12  return false;
  } //hetmany
```

Musimy jeszcze zaimplementować funkcję `szachowane`. Funkcja ta przejrzy listę hetmanów i sprawdzi, czy któryś z nich nie szachuje podanego w argumencie pola. Jeżeli okaże się, że któryś z hetmanów szachuje rozważane pole, zwrócimy wartość `true`. Jeżeli sprawdzimy wszystkie postawione wcześniej hetmany i żaden z nich nie szachuje podanego w argumencie funkcji pola, to zwrócimy `false`. W funkcji `szachowane` wykorzystamy fakt, iż wszystkie rozważane hetmany leżą na polach o indeksach wierszy mniejszych niż indeks wiersza, w którym leży sprawdzane pole (dzięki wspomnianemu faktowi możemy rozważyć mniej przypadków).

Listing 3.39. Rozwiązanie problemu ustawienia hetmanów

```
bool szachowane(unsigned int *tab_hetmany, unsigned int x,
                                            unsigned int y){
  for(unsigned int i=0;i<y;i++)
    if ((tab_hetmany[i]==x)||(y-i==abs(tab_hetmany[i]-x)))
      return true;
  return false;
}
```

Do zaimplementowania została nam jeszcze funkcja, która otrzymuje jako argument liczbę n i wypisuje na standardowym wyjściu wynik działania funkcji `hetmany_rek`. Funkcja ta musi także zadeklarować tablicę `tab_hetmany` wykorzystywaną przez funkcję `hetmany_rek`.

Listing 3.40. Rozwiązanie problemu ustawienia hetmanów

```
void hetmany(unsigned int n){
  unsigned int tab_hetmany[n];
  if (hetmany_rek(tab_hetmany, n, 0)){
    cout<<"Hetmany nalezy ustawic na polach     :"<<endl;
    for(unsigned int i=0;i<n;i++)
      cout<<tab_hetmany[i]<<" "<<i<<endl;
  }
  else
    cout<<"Nie da sie ustawic hetmanow"<<endl;
}
```

Zadanie 19. Napisz funkcję, która dla podanej na wejściu liczby całkowitej `n` wypisuje na wyjściu wszystkie możliwe rozwiązania problemu n hetmanów.

Zadanie 20. Napisz funkcję, która dostaje jako argument częściowo wypełnioną planszę sudoku (tablicę dwuwymiarową liczb całkowitych, w której wolne pola mają wartość ujemną) i wypisuje na standardowym wyjściu poprawnie uzupełnioną planszę otrzymaną w argumencie lub słowo `NIE`, jeżeli otrzymanej w argumencie planszy nie da się uzupełnić.

Rozdział 4

Metoda dziel i zwyciężaj

4.1. O metodzie **78**
4.1.1. Sortowanie szybkie 78
4.1.2. Mergesort 84
4.2. Zmniejsz i zwyciężaj **90**
4.2.1. Wyszukiwanie binarne 90
4.2.2. Algorytm Euklidesa 94

4.1. O metodzie

Metoda dziel i zwyciężaj to metoda tworzenia algorytmów, w której problem dzielimy na podproblemy tego samego lub podobnego rodzaju i rozwiązujemy je rekurencyjnie tak długo, aż dojdziemy do podproblemów tak prostych, że potrafimy je rozwiązać wprost. Wyniki dla podproblemów scalamy uzyskując wynik dla całego problemu.

Metoda dziel i zwyciężaj jest jedną z najpopularniejszych metod konstrukcji algorytmów. Przyjrzymy się jej na przykładzie sortowania listy. Zastanówmy się, jak mógłby wyglądać algorytm sortowania wykorzystujący strategię dziel i zwyciężaj. Na wysokim poziomie abstrakcji możemy stwierdzić, że taki algorytm musiałby mieć następujący schemat:

Algorytm 4.1.

1. *Jeżeli lista jest wystarczająco mała, na przykład jednoelementowa, sortujemy ją w jakikolwiek nierekurencyjny sposób i kończymy działanie algorytmu.*
2. *W przeciwnym wypadku dzielimy listę na dwie lub więcej rozłącznych podlist.*
3. *Wszystkie podlisty sortujemy wywołując rekurencyjnie tą samą funkcję sortującą.*
4. *Posortowane listy w jakiś sposób łączymy uzyskując posortowaną wyjściową listę.*

Korzystając z powyższego schematu można skonstruować różne algorytmy o różnej złożoności obliczeniowej. To, w jaki sposób podzielimy na początku listę na podlisty, będzie wpływało na to, w jaki sposób na końcu będziemy scalać posortowane podlisty i jaki w efekcie otrzymamy algorytm. Poniżej prezentujemy dwa najpopularniejsze algorytmy sortujące, oparte na powyższym schemacie.

4.1.1. Sortowanie szybkie

Sortowanie szybkie (ang. *quicksort*), to jeden z najpopularniejszych i najprostszych w implementacji algorytmów sortujących. W algorytmie tym listę dzielimy na dwie części względem jej wybranego elementu: dokonujemy podziału na elementy większe i mniejsze od wybranego, obie części sortujemy rekurencyjnie tą samą metodą i na koniec sklejamy wstawiając pomiędzy nie element dzielący.

Poniżej idea algorytmu quicksort w punktach:

Algorytm 4.2.

1. *Jeżeli lista jest jednoelementowa, to zwracamy ją bez zmian.*

2. *Wybieramy dowolny element* e *listy i dzielimy listę na dwie podlisty:*
   — *A – elementów mniejszych od* e*,*
   — *B – elementów większych równych* e *(ale bez* e*).*
3. *Sortujemy rekurencyjnie listę A i listę B.*
4. *Wynikową listę uzyskujemy sklejając ze sobą posortowaną listę A, element* e *i posortowaną listę B (w takiej kolejności).*

Zanim przystąpimy do implementacji algorytmu sortowania szybkiego, przyjrzyjmy się przykładowi jego działania:

Przykład 4.1. *Załóżmy, że chcemy posortować listę*

$$(6, 2, 4, 7, 5, 1, 9, 8, 3, 6).$$

Najpierw dokonamy podziału tej listy względem jej wybranego elementu. Niech tym wybranym elementem będzie 6*, czyli pierwszy element list. W takim przypadku w wyniku podziału otrzymamy następujące dwie listy:*

$$A = (2, 4, 5, 1, 3),\ B = (7, 9, 8, 6).$$

Następnie sortujemy rekurencyjnie listy A i B:

$$A = (1, 2, 3, 4, 5),\ B = (6, 7, 8, 9).$$

Na koniec sklejamy A, (6) *i B:*

$$A + (6) + B = (1, 2, 3, 4, 5) + (6) + (6, 7, 8, 9) = (1, 2, 3, 4, 5, 6, 6, 7, 8, 9).$$

Teraz zaimplementujemy algorytm sortowania szybkiego używając do tego gotowej implementacji listy. Najpierw, żeby zwiększyć przejrzystość kodu, napiszemy funkcję sklejającą dwie listy otrzymane w argumencie. Nasza funkcja dostanie referencje do dwóch list i doklei drugą listę na koniec pierwszej.

Listing 4.1. Sortowanie szybkie

```
void Sklej(Lista  &Lista1, Lista &Lista2){
2
    ...
4
}
```

Na początku sprawdzimy, czy druga lista nie jest pusta (jeżeli jest, to nic nie musimy robić).

Listing 4.2. Sortowanie szybkie

```
void Sklej(Lista &Lista1, Lista &Lista2){
  if (!Lista2.empty()){

    ...

  }
}
```

Jeżeli druga lista nie jest pusta, to będziemy przeglądali kolejne elementy drugiej listy dodając je na koniec pierwszej listy.

Listing 4.3. Sortowanie szybkie

```
void Sklej(Lista &Lista1, Lista &Lista2){
  if (!Lista2.empty()){
    Pozycja poz = Lista2.first();
    while(poz!=Lista2.last()){
      Lista1.push_back(Lista2.at(poz));
      poz = Lista2.next(poz);
    }

    ...

  }
}
```

Zauważmy, że w powyższej funkcji na koniec listy `Lista1` nie dodamy ostatniego elementu listy `Lista2`. Stanie się tak, gdyż zaraz po tym, jak zmienna `poz` otrzyma jako wartość pozycję ostatniego elementu, program wyskoczy z pętli `while`. Aby tego uniknąć moglibyśmy zamienić miejscami linie kodu numer 5 i 6, ale wtedy nie dodalibyśmy do pierwszej listy pierwszego elementu drugiej listy. Rozwiązaniem w tej sytuacji jest dodanie do listy `Lista1` ostatniego elementu listy `Lista2` poza pętlą.

Listing 4.4. Sortowanie szybkie

```
void Sklej(Lista &Lista1, Lista &Lista2){
  if (!Lista2.empty()){
    Pozycja poz = Lista2.first();
    while(poz!=Lista2.last()){
      Lista1.push_back(Lista2.at(poz));
      poz = Lista2.next(poz);
    }
    Lista1.push_back(Lista2.at(poz));
  }
}
```

Teraz napiszemy funkcję sortującą, która dostaje jako argument listę i zwraca jako wartość posortowaną listę:

Listing 4.5. Sortowanie szybkie

```
Lista QuickSort(Lista Wejscie){
2
    ...
4
}
```

Jeżeli lista jest pusta albo zawiera jeden element, to nie ma czego sortować.

Listing 4.6. Sortowanie szybkie

```
1 Lista QuickSort(Lista Wejscie){
    if ((!Wejscie.empty())&&(Wejscie.first()!=Wejscie.last())){
3
      ...
5
    }
7   else
      return Wejscie;
9 }
```

W przeciwnym wypadku musimy wybrać element, według którego będziemy dzielić listę `Wejscie` (w naszym przypadku będzie to pierwszy element listy) i zainicjować puste listy `ListaA` i `ListaB`.

Listing 4.7. Sortowanie szybkie

```
1 Lista QuickSort(Lista Wejscie){
    if ((!Wejscie.empty())&&(Wejscie.first()!=Wejscie.last())){
3     Element e = Wejscie.front()
      Lista ListaA, ListaB;
5
      ...
7
    }
9   else
      return Wejscie;
11 }
```

Następnie musimy przejrzeć całą listę `Wejscie`.

Listing 4.8. Sortowanie szybkie

```
1 Lista QuickSort(Lista Wejscie){
    if ((!Wejscie.empty())&&(Wejscie.first()!=Wejscie.last())){
```

```
3       Element e = Wejscie.front()
        Lista ListaA, ListaB;
5       Pozycja poz = Wejscie.first();
        while(poz!=Wejscie.last()) {
7         poz = Wejscie.next(poz);

9           ....
        }
11
        ...
13    }
      else
15      return Wejscie;
    }
```

Przeglądając listę `Wejscie` dzielimy jej elementy na dwie części (elementów mniejszych i większych równych `e`)

Listing 4.9. Sortowanie szybkie

```
    Lista QuickSort(Lista  Wejscie){
2     if ((!Wejscie.empty())&&(Wejscie.first()!=Wejscie.last())){
        Element e = Wejscie.front()
4       Lista ListaA, ListaB;
        Pozycja poz = Wejscie.first();
6       while(poz!=Wejscie.last()) {
          poz = Wejscie.next(poz);
8          if (Wejscie.at(poz)< e)
            ListaA.push_back(Wejscie.at(poz));
10        else
            ListaA.push_back(Wejscie.at(poz));
12      }

14      ...
      }
16    else
        return Wejscie;
18  }
```

Na koniec musimy posortować rekurencyjnie tą samą metodą listy `ListaA` i `ListaB` oraz skleić `ListaA`, `e` i `ListaB`.

Listing 4.10. Sortowanie szybkie

```
    Lista QuickSort(Lista  Wejscie){
2     if ((!Wejscie.empty())&&(Wejscie.first()!=Wejscie.last())){
        Element e = Wejscie.front()
4       Lista ListaA, ListaB;
        Pozycja poz = Wejscie.first();
6       while(poz!=Wejscie.last()) {
```

```
        poz = Wejscie.next(poz);
8       if (Wejscie.at(poz)< e)
          ListaA.push_back(Wejscie.at(poz));
10      else
          ListaA.push_back(Wejscie.at(poz));
12    }

14    ListaA = QuickSort(ListaA);
      ListaB = QuickSort(ListaB);
16    ListaA.push_back(e);
      sklej(ListaA, ListaB);
18    return ListaA;
    }
20  else
      return Wejscie;
22 }
```

Analizując powyższy kod czytelnik może się zacząć zastanawiać, czy to na pewno jest jeden z najprostszych algorytmów sortujących. Za chwilę przekonamy się, że można zaimplementować algorytm sortowania szybkiego prościej. W poniższej implementacji algorytmu quicksort sortujemy w miejscu elementy tablicy. Argumentami funkcji oprócz sortowanej tablicy są początek i koniec sortowanego przedziału tablicy (dokładnie indeks pierwszego elementu przedziału i indeks elementu znajdującego się tuż za ostatnim elementem przedziału). W rekurencyjnych wywołaniach funkcji będziemy podawali jako argumenty tą samą tablicę ale mniejsze przedziały do posortowania.

Listing 4.11. Sortowanie szybkie

```
  void QuickSort(Element Wejscie[], unsigned int p,
2                                    unsigned int k){
    if (k-p>1){
4     Element pom;
      unsigned int podzial=p;
6     for(i=p+1;i<k;i++)
        if (Wejscie[podzial]>Wejscie[i]){
8         pom=Wejscie[podzial];
          Wejscie[podzial]=Wejscie[i];
10        Wejscie[i]=Wejscie[podzial+1];
          Wejscie[podzial+1]=pom;
12        podzial++;
        }
14
      QuickSort(Wejscie,p,podzial);
16    QuickSort(Wejscie,podzial+1,k);
    }
18 }
```

Najistotniejsza część kodu w listingu 4.11 znajduje się od trzeciej do dwunastej linii, w których to zamieniamy miejscami elementy tablicy w taki sposób, że w komórkach o indeksach od `0` do `podzial-1` znajduje się lista A, w komórce `Wejscie[podzial]` element według którego dokonujemy podziału, zaś w komórkach o indeksach od `podzial+1` do `k-1` lista B. Dzięki temu po posortowaniu list A i B nie musimy ich już sklejać, gdyż są na dobrych miejscach.

Pesymistyczna złożoność algorytmu quicksort to $O(n^2)$. Taką złożoność sortowanie szybkie osiąga przykładowo dla posortowanej listy. Wtedy we wszystkich kolejnych podziałach sortowanej listy wszystkie elementy trafiają do jednej podlisty. Daje nam to liczbę porównań równą:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n \cdot (n-1)}{2},$$

a to jest $O(n^2)$. Z drugiej strony średnia złożoność sortowania szybkiego przy równym prawdopodobieństwie wystąpienia wszystkich permutacji wynosi $O(n \cdot \log n)$. Co więcej dla losowych danych quicksort okazuje się być jednym z najszybszych znanych algorytmów.

4.1.2. Mergesort

Innym algorytmem implementującym schemat z algorytmu 4.1 jest algorytm sortowania przez scalanie. Tym razem, inaczej niż w przypadku algorytmu quicksort, sortowaną listę podzielimy na dwie równe części. Potem, zgodnie ze schematem, obie podlisty posortujemy rekurencyjnie tą samą funkcją, a na koniec scalimy obie posortowane podlisty. Scalanie jest w tym przypadku nieco trudniejsze niż w przypadku sortowania szybkiego, ale jest możliwe do wykonania w czasie liniowym. Poniżej opis algorytmu:

Algorytm 4.3.

1. *Jeżeli lista ma nie więcej niż jeden element, to zwracamy ją bez zmian i kończymy działanie algorytmu.*
2. *W przeciwnym wypadku dzielimy listę na dwie podlisty A i B różniące się rozmiarem co najwyżej o jeden (na przykład na pierwszą połowę listy i drugą połowę listy,).*
3. *Sortujemy oddzielnie listę A i listę B.*
4. *Wynikową listę uzyskujemy przez scalenie ze sobą posortowanych list A i B.*

Implementację algorytmu zaczniemy od implementacji funkcji scalającej dwie posortowane listy. Przy użyciu następującego algorytmu można to zrobić w czasie liniowym:

Algorytm 4.4.

1. *Utwórz pustą listę C.*
2. *Dopóki żadna z list A i B nie jest pusta, wykonuj kroki* 3 *i* 4.
3. *Wybierz mniejszy spośród najmniejszych (pierwszych) elementów list A i B (niech to będzie* e*).*
4. *Usuń* e *z listy, na której się znajduje i wstaw na koniec listy C.*
5. *Doklej na koniec listy C tą spośród list A i B, która nie jest pusta.*
6. *Zwróć listę C jako wynik.*

Na przykładzie dwóch krótkich list zobaczymy działanie algorytmu scalającego:

Przykład 4.2. *Będziemy scalali dwie listy $A = (1, 4, 5)$ i $B = (2, 4, 6, 7)$ przenosząc pojedynczo elementy list A i B do nowo utworzonej listy C. Porównujemy ze sobą pierwsze elementy list A i B:*

$$A = (\mathbf{1}, 4, 5),\ B = (\mathbf{2}, 4, 6, 7),\ C = ().$$

Jedynka jest mniejsza od dwójki, więc przenosimy ją do listy C:

$$A = (4, 5),\ B = (2, 4, 6, 7),\ C = (1).$$

W kolejny porównaniu porównujemy 4 *z* 2 *i przenosimy* 2*, jako mniejszą z porównywanych wartości, na koniec listy C:*

Teraz mamy ciekawą sytuację, gdyż pierwsze elementy list A i B są równe. W takiej sytuacji na koniec listy C możemy przenieść którąkolwiek z czwórek. Załóżmy, że przenosimy pierwszy element listy A:

$$A = (5),\ B = (4, 6, 7),\ C = (1, 2, 4).$$

Po kolejnym porównaniu pierwszych elementów list A i B na koniec listy C trafia druga z czwórek:

$$A = (5),\ B = (6, 7),\ C = (1, 2, 4, 4).$$

I znowu mamy ciekawą sytuację, gdyż porównujemy ze sobą jedyny element listy A z pierwszym elementem listy B:

$$A = (\mathbf{5}),\ B = (\mathbf{6}, 7),\ C = (1, 2, 4, 4).$$

W wyniku tego porównania 5 *trafia na koniec listy C i lista A jest już pusta.*

$$A = (),\ B = (6, 7),\ C = (1, 2, 4, 4, 5).$$

W tej sytuacji doklejamy na koniec listy C całą pozostałą część listy B:

$$A = ()\ B = ()\ C = (1, 2, 4, 4, 5, 6, 7)$$

i w ten sposób kończymy scalanie list A i B.

Funkcja scalająca otrzyma jako argumenty dwie posortowane listy i zwróci jako wartość posortowaną listę zawierającą wszystkie elementy z obu list otrzymanych w argumentach:

Listing 4.12. Sortowanie przez scalanie

```
  Lista Merge(Lista ListaA, Lista ListaB){
2
    ...
4
  }
```

W funkcji `Merge` będziemy przebiegać jednocześnie listy `ListaA` i `ListaB` zaczynając od ich początków. Elementy tych list będziemy przenosić do listy `ListaC` wewnątrz pętli `while`, która będzie się kręcić, dopóki któraś z otrzymanych w argumentach list nie będzie pusta.

Listing 4.13. Sortowanie przez scalanie

```
1 Lista Merge(Lista ListaA, Lista ListaB){
    Lista ListaC;
3   while((!ListaA.empty())&&(!ListaB.empty()))

5   ...

7 }
```

W każdym obrocie pętli `while` będziemy przenosić do listy `ListaC` mniejszy z pierwszych elementów list `ListaA` i `ListaB`.

Listing 4.14. Sortowanie przez scalanie

```
1 Lista Merge(Lista ListaA, Lista ListaB){
    Lista ListaC;
3   while((!ListaA.empty())&&(!ListaB.empty()))
      if (ListaA.front()<=ListaB.front()){
5       ListaC.push_back(ListaA.front());
        ListaA.pop_front();
7     }
      else {
9       ListaC.push_back(ListaB.front());
        ListaB.pop_front();
11    }

13  ...

15 }
```

Pozostało nam jeszcze dokleić na koniec listy `ListaC` tą z list `ListaA` i `ListaB`, która nie jest jeszcze pusta. Do sklejania dwu list użyjemy funkcji `sklej` z listingu 4.4.

Listing 4.15. Sortowanie przez scalanie

```
Lista Merge(Lista ListaA, Lista ListaB){
  Lista ListaC;
  while((!ListaA.empty())&&(!ListaB.empty()))
    if (ListaA.front()<=ListaB.front()){
      ListaC.push_back(ListaA.front());
      ListaA.pop_front();
    }
    else {
      ListaC.push_back(ListaB.front());
      ListaB.pop_front();
    }
  if (!ListaA.empty())
    sklej(ListaC, ListaA);
  else
    sklej(ListaC, ListaB);
  return ListaC

}
```

Teraz zaimplementujemy funkcję sortującą, która dostanie jako argument listę i zwróci jako wartość posortowaną listę:

Listing 4.16. Sortowanie przez scalanie

```
Lista MergeSort(Lista Wejscie){

  ...

}
```

W przypadku, gdy otrzymana w argumencie lista będzie co najwyżej jednoelementowa, funkcja zwróci ją od razu jako wartość

Listing 4.17. Sortowanie przez scalanie

```
Lista MergeSort(Lista Wejscie){
  if ((!Wejscie.empty())&&(Wejscie.first!=Wejscie.last())){

    ...

  }
  else
    return Wejscie;
}
```

Na początku musimy podzielić listę `Wejscie` na dwie równe podlisty. Ponieważ nie znamy rozmiaru listy `Wejscie`, to będziemy umieszczali jej elementy na zmianę w listach `ListaA` i `ListaB`, i w ten sposób uzyskamy podział listy na dwie równe części .

Listing 4.18. Sortowanie przez scalanie

```
1  Lista MergeSort(Lista Wejscie){
     if ((!Wejscie.empty())&&(Wejscie.first!=Wejscie.last())){
3      Lista ListaA, ListaB;
       while(!Wejscie.empty()){
5        ListaA.push_back(Wejscie.front());
         Wejscie.pop_front();
7        if (!Wejscie.empty()){
           ListaB.push_back(Wejscie.front());
9          Wejscie.pop_front()
         }
11     }

13     ...

15   }
     else
17     return Wejscie;
   }
```

Zwróćmy uwagę, że w momencie dodawania elementu do listy `ListaA` lista `Wejscie` na pewno nie jest pusta (inaczej nie rozpoczęlibyśmy kolejnego obrotu pętli `while`), natomiast nie mamy takiej pewności w momencie dodawania nowego elementu do listy `ListaB` i dlatego sprawdzamy wcześniej warunek, czy lista `Wejscie` nie jest pusta.

Mając podzieloną listę `Wejscie` na dwie części sortujemy te części rekurencyjnie tą samą metodą i scalamy wynikowe listy. Wynik scalenia posortowanych podlist zwracamy jako wartość funkcji.

Listing 4.19. Sortowanie przez scalanie

```
   Lista MergeSort(Lista Wejscie){
2    if ((!Wejscie.empty())&&(Wejscie.first!=Wejscie.last())){
       Lista ListaA, ListaB;
4      while(!Wejscie.empty()){
         ListaA.push_back(Wejscie.front());
6        Wejscie.pop_front();
         if (!Wejscie.empty()){
8          ListaB.push_back(Wejscie.front());
           Wejscie.pop_front()
10       }
       }
12     ListaA = MergeSort(ListaA);
```

```
    ListaB = MergeSort(ListaB);
14  return Merge(ListaA, ListaB)
  }
16 else
    return Wejscie;
18 }
```

Podobnie jak w przypadku sortowania szybkiego także algorytm sortowania przez scalanie łatwiej jest zaimplementować gdy nie ograniczamy się do podstawowych operacji listy. Poniżej prezentujemy implementację algorytmu mergesort sortującą elementy tablicy. Podobnie jak w przypadku tablicowej implementacji algorytmu quicksort jako argumenty funkcji sortującej, poza samą sortowaną tablicą, podajemy początek i koniec (dokładnie indeks tuż za końcem) sortowanego przedziału.

Listing 4.20. Sortowanie przez scalanie

```
void Merge(Element Wejscie[], unsigned int p,
2                              unsigned int k){
  unsigned int sr,s;
4 sr = s = (p+k)/2;
  Element * pom = new Element[k-p];
6 for(unsigned int i=0;i<k-p;i++)
    if (p == sr){
8     pom[i]=Wejscie[s];
      s++;
10  } else if (s == k) {
      pom[i]=Wejscie[p];
12    p++;
    } else if (Wejscie[p]<=Wejscie[s])
14    pom[i]=Wejscie[p];
      p++;
16  else {
      pom[i] = Wejscie[s];
18    k++
    }
20 for(unsigned int i=0; i<k-p;i++)
    Wejscie[i+p] = pom[i];
22 delete [] pom;
}
24 void MergeSort(Elementy Wejscie[], unsigned int p,
                                    unsigned int k){
26 if (p<k-1){
    unsigned int s= (p+k)/2;
28  mergesort(Wejscie,p,s);
    mergesort(Wejscie,s,k);
30  merge(Wejscie,p,k);
  }
32 }
```

Jak widzimy, w przypadku, gdy listę przechowujemy w tablicy, operacja podziału listy na dwie części jest trywialna. Funkcja `Merge` wykorzystuje tablicę pomocniczą `pom` do zapisania scalonej listy, którą następnie przepisuje z powrotem do tablicy `Wejscie`.

Na koniec oszacujemy złożoność algorytmu sortowania przez scalanie. Zauważmy, że złożoność czasowa funkcji `MergeSort`, jeżeli nie liczyć jej rekurencyjnych wywołań , jest liniowa. Zauważmy również, że na każdym poziomie rekurencji suma długości wszystkich rozważanych list jest co najwyżej taka, jak długość startowej listy (wynika to z faktu, że za każdym razem dzielimy listę na **rozłączne** podlisty). Oznacza to, że złożoność wykonania wszystkich operacji na danym poziomie rekurencji dla listy o długości n wynosi $O(n)$. Ponieważ poziomów rekurencji jest $O(\log n)$ (wynika to z faktu, że na każdym kolejnym poziomie rekurencji sortujemy listy o połowę krótsze niż na poprzednim poziomie), to pesymistyczna czasowa złożoność całego algorytmu jest $O(n \cdot \log n)$. Łatwo pokazać, że średnia złożoność sortowania przez scalanie też wynosi $O(n \cdot \log n)$ (jest to konsekwencja faktu, że podziały na podlisty nie zależą od kolejności elementów listy).

4.2. Zmniejsz i zwyciężaj

Szczególnym przypadkiem metody dziel i zwyciężaj jest przypadek, kiedy rozwiązywany problem udaje nam się sprowadzić do analogicznego problemu, ale dla mniejszych danych. Przypadek ten nazywany jest „zmniejsz i zwyciężaj". Przyjrzymy się dwóm przypadkom, w których taka strategia działa.

4.2.1. Wyszukiwanie binarne

Wyszukiwanie binarne to algorytm wyszukiwania danych w posortowanej liście przechowywanej w tablicy. Algorytm dla danej tablicy i poszukiwanego elementu zwróci indeks w tablicy szukanego elementu lub informację, że poszukiwanego elementu nie ma w tablicy. Algorytm wyszukiwania binarnego można opisać w następujący sposób:

Algorytm 4.5. *Szukamy elementu* e *w liście* L

1. *Jeżeli lista* L *jest pusta lub jednoelementowa, zwracamy informację czy element* e *należy do* L *i kończymy działanie algorytmu.*
2. *W przeciwnym wypadku porównujemy element* e *z elementem* s *znajdującym się w połowie listy* L*.*
3. *Jeżeli element* e *jest równy elementowi* s*, to kończymy działanie algorytmu.*

4. *Jeżeli element* e *jest mniejszy od elementu* s*, to rekurencyjnie przeszukujemy pierwszą połowę listy L. W przeciwnym wypadku przeszukujemy jej drugą połowę.*

Zanim przejdziemy do implementacji powyższego algorytmu przeanalizujemy jego działanie dla przykładowej listy.

Przykład 4.3. *Będziemy wyszukiwali wartość* 8 *w liście przechowywanej w tablicy*

3	4	6	7	9	11	15	23	30	31

Najpierw porównujemy 8 *z wartością środkową listy. Ponieważ lista ma parzystą liczbę elementów (a więc nie ma elementu znajdującego się dokładnie w środku), bierzemy element znajdujący się tuż za środkiem czyli* 11*:*

3	4	6	7	9	**11**	15	23	30	31

Ponieważ 8 *jest mniejsze od* 11 *to w dalszej części algorytmu będziemy przeszukiwać tylko pierwszą połowę tablicy:*

3	4	6	7	9

Teraz 8 *porównujemy z* 6 *i ponieważ* 8 *jest większe, to ograniczamy poszukiwania do listy:*

7	9

Po porównaniu 9 *i* 8 *pozostaje nam do przetworzenia jednoelementowa lista*

7

i ponieważ nie zawiera on 8*, to algorytm zwraca jako wynik informację, że przeszukiwana lista nie zawiera* 8*.*

Algorytm wyszukiwania binarnego implementujemy tylko dla tablicowej implementacji listy, gdyż tylko wtedy mamy możliwość szybkiego dostępu do elementu za pomocą jego indeksu (numeru miejsca, na którym występuje).

Napiszemy funkcję, która otrzymuje jako argumenty szukany element `e`, posortowaną tablicę `Wejscie`, w której szukamy elementu `e` oraz `p` i `k` początek i koniec przedziału, w którym mamy szukać elementu `e`. Tradycyjnie `p` oznacza indeks pierwszego elementu przedziału, zaś `k` indeks pierwszego elementu tuż za przeszukiwanym przedziałem. Indeks poszukiwanego elementu zwrócimy jako wartość funkcji. W przypadku, gdy `e` nie jest przechowywane w tablicy, nasza funkcja zwróci wartość `-1`.

Listing 4.21. Wyszukiwanie binarne

```
1 int BinSearch(Element e, Element Wejscie[], unsigned int p,
                                               unsigned int k){
3
    ...
5
  }
```

Implementowana przez nas funkcja będzie składała się z serii instrukcji warunkowych rozważających różne przypadki. Pierwszy przypadek, który rozważamy to sytuacja, gdy `p` jest większe lub równe `k`. Jest to przypadek, gdy mamy przeszukać pusty przedział i wynikiem w takiej sytuacji jest `-1`.

Listing 4.22. Wyszukiwanie binarne

```
  int BinSearch(Element e, Element Wejscie[], unsigned int p,
2                                              unsigned int k){
    if (p>=k)
4     return -1;

6   ...

8 }
```

Drugi przypadek, który rozważymy, to taki, gdy element tablicy znajdujący się w środku przeszukiwanego przedziału jest poszukiwanym przez nas elementem. W takim wypadku nasza funkcja powinna zwrócić indeks środka przedziału.

Listing 4.23. Wyszukiwanie binarne

```
   int BinSearch(Element e, Element Wejscie[], unsigned int p,
2                                              unsigned int k){
     if (p>=k)
4      return -1;
     unsigned int s = (p+k)/2;
6    if (Wejscie[s]==e)
       return s;
8
     ...
10
   }
```

Przedostatni przypadek to taki, w którym w środku przedziału leży element większy od poszukiwanego. W takiej sytuacji przeszukujemy pierwszą połowę przedziału

Listing 4.24. Wyszukiwanie binarne

```
1 int BinSearch(Element e, Element Wejscie[], unsigned int p,
                                                unsigned int k){
3   if (p>=k)
      return -1;
5   unsigned int s = (p+k)/2;
    if (Wejscie[s]==e)
7     return s;
    if (Wejscie[s]<e)
9     return BinSearch(Wejscie, p, s);

11  ...

13 }
```

W przeciwnym przypadku przeszukujemy drugą połowę przedziału.

Listing 4.25. Wyszukiwanie binarne

```
1 int BinSearch(Element e, Element Wejscie[], unsigned int p,
                                                unsigned int k){
3   if (p>=k)
      return -1;
5   unsigned int s = (p+k)/2;
    if (Wejscie[s]==e)
7     return s;
    if (Wejscie[s]>e)
9     return BinSearch(Wejscie, p, s);
    return BinSearch(Wejscie, s+1,k)
11 }
```

Algorytmy zmniejsz i zwyciężaj zazwyczaj dają się łatwo zaimplementować w sposób iteracyjny (nierekurencyjny). W takiej implementacji, kolejnym poziomom rekurencji z implementacji rekurencyjnej odpowiadają zazwyczaj kolejne obroty pętli. Poniżej iteracyjna wersja funkcji `BinSearch`.

Listing 4.26. Wyszukiwanie binarne

```
1 int BinSearch(Element e, Element Wejscie[],
                        unsigned int rozmiar){
3   unsigned int p,k,s;
    p=0;
5   k=rozmiar;
    while (p<k) {
7     s = (p+k)/2
      if (Wejscie[s] == e)
9       return s;
      if (Wejscie[s] < e)
11      k = s;
```

```
        else
13          p = s+1;
      }
15    return -1;
    }
```

Iteracyjne wersje algorytmów są zazwyczaj szybsze, wymagają mniej pamięci i łatwiej znaleźć w nich ewentualne błędy. Stąd, gdy poziom skomplikowania wersji rekurencyjnej i iteracyjnej jest porównywalny, warto stosować wersję iteracyjną.

Na koniec oszacujemy złożoność czasową algorytmu wyszukiwania binarnego. Zauważmy, że na kolejnych poziomach rekurencji (lub w kolejnych obrotach pętli w wersji iteracyjnej) rozmiar przeszukiwanego przedziału zmniejsza się o połowę. Stąd wniosek, że dla przedziału zawierającego n lementów funkcja `BinSearch` wywoła się rekurencyjnie $O(\log n)$ razy (pętla `while` wykona się $O(\log n)$ razy). Ponieważ na każdym poziomie rekurencji (w każdym obrocie pętli `while`) wykonujemy stałą liczbę operacji, to pesymistyczna złożoność czasowa algorytmu wyszukiwania binarnego wynosi $O(\log n)$.

4.2.2. Algorytm Euklidesa

Algorytm Euklidesa znajduje największy wspólny dzielnik dwóch liczb naturalnych. W algorytmie wykorzystamy następujące proste własności liczb naturalnych:

— $NWD(a, b) = NWD(b, a)$,
— jeżeli $a > b$, to $NWD(a, b) = NWD(b, a\%b)$,
— jeżeli $a\%b = 0$, to $NWD(a, b) = b$.

Napiszemy funkcję, która dla różnych przypadków zastosuje którąś z powyższych własności. Nasza funkcja dla dwóch dodatnich liczb całkowitych a i b zwróci jako wartość $NWD(a, b)$

Listing 4.27. Algorytm Euklidesa

```
  unsigned int Euklid(unsigned int a, unigned int b){
2
      ...
4
  }
```

Aby uprościć dalszą część funkcji chcielibyśmy móc założyć, że drugi argument funkcji jest mniejszy lub równy pierwszemu

Listing 4.28. Algorytm Euklidesa

```
1 unsigned int Euklid(unsigned int a, unigned int b){
    if (a<b)
3     return Euklid(b,a);

5   ...

7 }
```

Ponadto chcemy sprawdzić, czy mniejszy z argumentów przypadkiem nie dzieli większego.

Listing 4.29. Algorytm Euklidesa

```
1 unsigned int Euklid(unsigned int a, unigned int b){
    if (a<b)
3     return Euklid(b,a);
    if (a%b == 0)
5     return b;

7   ...

9 }
```

W pozostałych przypadkach korzystamy z drugiej spośród wymienionych na początku własności NWD.

Listing 4.30. Algorytm Euklidesa

```
1 unsigned int Euklid(unsigned int a, unigned int b)
    if (a<b)
3     return Euklid(b,a);
    if (a%b == 0)
5     return b;
    return Euklid(b,a%b);
7 }
```

Do oszacowania pozostała nam jeszcze złożoność obliczeniowa tego algorytmu. Za operacje dominujące w tym algorytmie uznajemy operacje arytmetyczne. Zauważmy, że jeżeli $a > b$, to $a\%b < a/2$. Z tego wynika, że co dwa poziomy rekurencji większa z pary liczb zmniejsza się co najmniej o połowę. To z kolei oznacza, że maksymalna głębokość rekurencji jest $O(\log n)$. Ponieważ na każdym poziomie rekurencji wykonujemy $O(1)$ operacji, to złożoność całego algorytmu to $O(\log max(m, n))$, gdzie m i n są liczbami podanymi na wejściu. Oznacza to, że algorytm ma **złożoność liniową** względem rozmiaru wejścia (rozmiar wejścia to w tym przypadku suma logarytmów obu liczb, bo tyle potrzebujemy miejsca do ich zapisu).

Zadanie 21. Zaimplementuj iteracyjna wersję algorytmu Euklidesa.

Rozdział 5

Programowanie dynamiczne

5.1. O metodzie **98**
5.1.1. Problem plecakowy 100
5.1.2. Najdłuższy wspólny podciąg 109
5.2. Metoda spamiętywania **116**
5.2.1. Problem plecakowy 117

5.1. O metodzie

Programowanie dynamiczne jest metodą konstruowania algorytmów, w której wynik dla danego problemu otrzymujemy poprzez rozwiązywanie podproblemów od przypadków najprostszych do bardziej złożonych, aż po rozwiązanie całego problemu. Uzyskane wyniki zapisujemy w tablicy. Programowanie dynamiczne stosujemy zazwyczaj do rozwiązywania problemów optymalizacyjnych, czyli takich, w których szukamy najlepszego rozwiązania ze zbioru rozwiązań dopuszczalnych. Aby móc zastosować programowanie dynamiczne problem musi mieć własność optymalnej podstruktury, czyli optymalne rozwiązanie problemu musi być funkcją optymalnych rozwiązań podproblemów. Inaczej mówiąc posiadając optymalne rozwiązania podproblemów powinniśmy być w stanie na ich podstawie uzyskać optymalne rozwiązanie całego problemu.

W przypadku konstruowania algorytmu wykorzystującego programowanie dynamiczne kluczowe są dwie rzeczy: wymyślenie podproblemów, na które możemy podzielić nasz problem i odkrycie rekurencyjnej zależności pomiędzy problemami i ich podproblemami. Mając te dwie rzeczy implementacja algorytmu jest zwykle bardzo prosta. Dobrze jeżeli podproblemy są sparametryzowane nieujemnymi liczbami całkowitymi. Wtedy łatwo jest zapisywać w tablicy rozwiązania poszczególnych podproblemów, gdyż możemy użyć parametrów poszczególnych podproblemów jako indeksów komórek tablicy przechowujących ich rozwiązania.

W przypadku gdy mamy dane podział na podproblemy i rekurencyjną zależność, możemy się zastanowić dlaczego nie rozwiązać problemu za pomocą metody „dziel i zwyciężaj". Programowanie dynamiczne stosujemy wtedy, gdy podproblemy na siebie nachodzą i w trakcie obliczeń wielokrotnie potrzebujemy tych samych wyników. W takiej sytuacji klasyczna metoda „dziel i zwyciężaj" jest nieefektywna gdyż wielokrotnie rozwiązuje te same podproblemy. Prosty przypadek, w którym rozwiązanie „dziel i zwyciężaj" jest nieefektywne ze względu na wielokrotne liczenie tych samych wartości, analizowaliśmy w rozdziale 1.2 na przykładzie listingu 1.1 zawierającego funkcję liczącą elementy ciągu Fibonacciego. Raz jeszcze przeanalizujemy ten przypadek i przy okazji zobaczymy, jak można rozwiązać ten problem w sposób wykorzystujący programowanie dynamiczne. Poniżej prezentujemy raz jeszcze prostą funkcję, która dla otrzymanej w argumencie liczby n liczy wprost ze wzoru rekurencyjnego n-ty element ciągu Fibonacciego.

Listing 5.1. Funkcja licząca n-ty wyraz ciągu Fibonacciego – wersja 1

```
int f1(unsigned int n){
  if (n==0)||(n==1) return n;
  return f1(n-1)+f1(n-2);
}
```

Funkcja z listingu 5.1 działa w czasie wykładniczym względem wartości n, gdyż wielokrotnie liczymy w niej te same wartości funkcji (przykładowo `f1(1)` liczymy wykładniczą liczbę razy).

W programowaniu dynamicznym wyniki dla podproblemów liczymy iteracyjnie od najprostszych do bardziej skomplikowanych zapisując wyniki w tablicy:

Listing 5.2. Funkcja licząca n-ty wyraz ciągu Fibonacciego – wersja 2

```
int f2(unsigned int n){
  if (n<2)
    return n;
  unsigned int *t= new unsigned int[n];
  t[0]=0;
  t[1]=1;
  for(unsigned int i=2;i<=n;i++)
    t[i]=t[i-1]+t[i-2];
  unsigned int pom = t[n];
  delete [] t;
  return pom;
}
```

Widzimy, że w powyższym rozwiązaniu każdą wartość liczymy dokładnie raz, a złożoność powyższego programu to $O(n)$.

Słabością programowania dynamicznego jest duże zapotrzebowanie na pamięć. Aby zmniejszyć choć trochę zapotrzebowanie na pamięć możemy zmniejszyć do stałego rozmiaru jeden z wymiarów tablicy z wynikami. To, do jakiego rozmiaru możemy zmniejszyć dany wymiar tablicy, zależy od tego jak głęboko do tyłu sięgamy we wzorze rekurencyjnym. W naszym przypadku potrzebujemy tablicy trójelementowej (dwie komórki na przechowywanie już obliczonych przypadków i jedna komórki na właśnie wyliczaną wartość). Zmianę wersji programu pamiętającej wszystkie rozwiązania na taką, z ograniczonym rozmiarem jednego wymiaru można zrobić zupełnie mechanicznie. Wystarczy zamienić we wszystkich odwołaniach do tablicy wyników indeks redukowanego wymiaru na jego resztę z dzielenia przez rozmiar tego wymiaru. My dodatkowo zrezygnujemy ze sprawdzania czy $n < 2$. W listingu 5.2 potrzebowaliśmy tego warunku, gdyż rozmiar tablicy `t` zależał od n.

Listing 5.3. Funkcja licząca n-ty wyraz ciągu Fibonacciego – wersja 3

```
int f3(unsigned int n){
  int i,t[3];
  t[0]=0;
  t[1]=1;
  for(i=2;i<n;i++){
    t[i%3]=t[(i-1)%3]+t[(i-2)%3];
  }
  return t[(n-1)%3];
}
```

W przypadku wielowymiarowej tablicy zapisującej rozwiązania podproblemów i kilku zagnieżdżonych pętli, przebiegających indeksy tej tablicy, możemy skrócić tylko ten wymiar, po którym przebiega najbardziej zewnętrzna pętla (w przypadku, gdy pamiętamy rozwiązania wszystkich podproblemów, kolejność pętli przebiegających indeksy tablicy rozwiązań jest zwykle obojętna).

5.1.1. Problem plecakowy

Problem plecakowy definiujemy w następujący sposób:

Problem 5.1.

Wejście *Na wejściu otrzymujemy listę przedmiotów (każdy z przedmiotów ma swoją cenę, wagę oraz dostępną ilość) oraz maksymalną wagę W przedmiotów, jakie możemy schować do plecaka. Zakładamy, że wagi i ceny przedmiotów są liczbami dodatnimi.*

Wyjście *Lista przedmiotów o maksymalnej możliwej sumie wartości, których suma wag nie przekracza W. Każdy przedmiot możemy wybrać w ilości nieprzekraczającej jego dostępną ilość.*

My rozwiążemy przy pomocy programowania dynamicznego dyskretny problem plecakowy – wersję problemu plecakowego, w której przedmioty występują pojedynczo (jest dostępna jedna sztuka każdego przedmiotu z listy).

Tabela 5.1. Przykładowe dane dla dyskretnego problemu plecakowego

numer przedmiotu	0	1	2	3	4
cena przedmiotu	70zł	40zł	5zł	50zł	40zł
waga przedmiotu	3kg	2kg	1kg	4kg	2kg

Przykład 5.1. *Dla przykładowych danych wejściowych dyskretnego problemu plecakowego znajdujących się w tabeli 5.1 i ograniczeniu na wagę rów-*

nemu 4kg, optymalnym rozwiązaniem jest wybór przedmiotów numer 1 i 4, które w sumie ważą 4kg i mają wartość 8zł.

Na początku rozwiążemy problem, w którym pytamy się tylko o maksymalną możliwą do uzyskania sumę wartości przedmiotów, a potem zobaczymy jak zmienić funkcję tak, żeby generowała także optymalny wybór przedmiotów.

Najpierw musimy zdefiniować podproblemy i rekurencyjną zależność pomiędzy nimi. W programowaniu dynamicznym często definiujemy podproblemy, w których bierze się pod uwagę tylko część danych wejściowych głównego problemu. Tak postąpimy w tym przypadku. Naszymi podproblemami będą dyskretne problemy plecakowe, które na wejściu dostają pierwszych k przedmiotów z pełnego problemu. Drugim parametrem podproblemów będzie ograniczenie na pojemność plecaka, które będzie przyjmowało wartości od 0 do W (znowu jest to typowe rozwiązanie w przypadku liczbowych ograniczeń występujących w treści zadania). Podsumowując, podproblemem o parametrach i, j będzie dyskretny problem plecakowy, w którym bierzemy pod uwagę i pierwszych spośród otrzymanych na wejściu przedmiotów, zaś ograniczenie na sumaryczną wagę wybranych przedmiotów wynosi j. Wyniki będziemy zapisywali w dwuwymiarowej tablicy `A`. W komórce `A[i][j]` będziemy zapisywali optymalną sumę wartości przedmiotów w podproblemie o parametrach i, j.

Teraz poszukamy rekurencyjnej zależności, którą wykorzystamy do rozwiązania problemu. Zauważmy, że `A[0][j]=0` (przypadek kiedy mamy do wyboru 0 przedmiotów) i `A[i][0]=0` (przypadek, w którym plecak ma pojemność 0). Zastanówmy się, ile może być równe `A[i][j]` dla $i, j > 0$. Są dwie możliwości: albo przedmiot i-ty przedmiot (czyli przedmiot o numerze $i - 1$) należy do optymalnego wyboru, albo do niego nie należy. W sytuacji gdy i-ty przedmiot nie należy do optymalnego wyboru, zachodzi `A[i][j]=A[i-1][j]`, natomiast w przeciwnym wypadku mamy `A[i][j]=A[i-1][j-w[i-1]]+c[i-1]`, gdzie `w[i-1]` jest wagą, zaś `c[i-1]` ceną i-tego przedmiotu (jeżeli wybierzemy i-ty przedmiot, to zostanie nam jeszcze `j-w[i-1]` miejsca w plecaku, które możemy wykorzystać do schowania jednego z `i-1` pierwszych przedmiotów). Oczywiście nie wiemy z góry, czy dany przedmiot należy do optymalnego wyboru czy nie, dlatego rozważamy obie opcje i wybieramy tą, która daje lepszy wynik (wyższą sumę cen). Przy powyższych rozważaniach musimy dodatkowo sprawdzić, czy i-ty przedmiot mieści się w plecaku (czy `w[i-1]<=j`), gdyż jeżeli się nie mieści, to `A[i][j]=A[i-1][j]`.

Napiszemy funkcję `plecak`, która otrzymuje jako argumenty tablice `c` i `w` zawierające odpowiednio ceny i wagi poszczególnych przedmiotów, zmienną `liczba` przechowującą liczbę przedmiotów oraz ograniczenie na sumę wag

Tabela 5.2. Wypełnienie tablicy A dla danych z Przykładu 5.1.

	0	**1**	**2**	**3**	**4**
0	0	0	0	0	0
1	0	0	0	70	70
2	0	0	40	70	70
3	0	5	40	70	75
4	0	5	40	70	75
5	0	5	40	70	80

przedmiotów w plecaku `W` i wypisuje na standardowym wyjściu maksymalną możliwą wartość przedmiotów w plecaku.

Listing 5.4. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){

  ...

}
```

W funkcji `plecak` użyjemy dwuwymiarowej tablicy `A`, w której będziemy zapisywali rozwiązania dla wszystkich podproblemów jak również rozwiązanie całego problemu. Ponieważ standard języka C++ nie wspiera automatycznych tablic o rozmiarach podanych przez zmienne, `A` będzie dynamiczną dwuwymiarową tablicą tablic, którą na koniec działania funkcji usuniemy.

Listing 5.5. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){
  //tworzenie tablicy A
  unsigned int ** A = new unsigned int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    A[i] = new unsigned int[W+1];

  ...

  cout<<A[liczba][W];

  //usuwanie z pamieci tablicy A
  for(unsigned int i=0;i<=liczba;i++)
    delete [] A[i];
  delete [] A;
}
```

Pierwszy wiersz i pierwszą kolumnę tablicy `A` trzeba na początku wyzerować (wynika to z faktu, że przy braku przedmiotów do wyboru albo zerowej pojemności plecaka nie da się wybrać żadnego przedmiotu):

Listing 5.6. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){
  //tworzenie tablicy A
  unsigned int ** A = new unsigned int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    A[i] = new unsigned int[W+1];
  //inicjowanie tablicy A
  for(unsigned int i = 0; i<=liczba;i++)
    A[i][0] = 0;
  for(unsigned int i = 0; i<=W;i++)
    A[0][i] = 0;

  ...

  cout<<A[liczba][W];

  //usuwanie z pamieci tablicy A
  for(unsigned int i=0;i<=liczba;i++)
    delete [] A[i];
  delete [] A;
}
```

Następnie wypełniamy tablicę `A`, używając poznanej rekurencyjnej zależności. Tablicę `A` wypełniamy od najmniejszych indeksów do największych (wynika to z faktu, że we wzorze rekurencyjnym sięgamy do komórek o mniejszych indeksach niż ta, której wartość wyliczamy).

Wszystkie możliwe pary parametrów podproblemów będziemy generowali w dwóch zagnieżdżonych pętlach `for`, których zmienne sterujące będą przebiegały wartości od 1 w górę (pierwszy wiersz i pierwszą kolumnę tablicy `A` wcześniej wypełniliśmy).

Listing 5.7. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){
  //tworzenie tablicy A
  unsigned int ** A = new unsigned int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    A[i] = new unsigned int[W+1];
  //inicjowanie tablicy A
  for(unsigned int i = 0; i<=liczba;i++)
    A[i][0] = 0;
  for(unsigned int i = 0; i<=W;i++)
```

```
11      A[0][i] = 0;

13    //wlasciwa czesc algorytmu
      for(unsigned int i=1;i<=liczba;i++)
15      for(unsigned int j=1;j<=W;j++)

17    ...

19    cout<<A[liczba][W];

21    //usuwanie z pamieci tablicy A
      for(unsigned int i=0;i<=liczba;i++)
23      delete [] A[i];
      delete [] A;
25  }
```

Dla kolejnych par parametrów podproblemów zastosujemy wzór rekurencyjny.

Listing 5.8. Problem plecakowy – programowanie dynamiczne

```
1  void plecak(unsigned int c[], unsigned int w[],
               unsigned int liczba, unsigned int W){
3    //tworzenie tablicy A
     unsigned int ** A = new unsigned int[liczba+1];
5    for(unsigned int i=0;i<=liczba;i++)
       A[i] = new unsigned int[W+1];
7    //inicjowanie tablicy A
     for(unsigned int i = 0; i<=liczba;i++)
9      A[i][0] = 0;
     for(unsigned int i = 0; i<=W;i++)
11     A[0][i] = 0;

13   //wlasciwa czesc algorytmu
     for(unsigned int i=1;i<=liczba;i++)
15     for(unsigned int j=1;j<=W;j++)

17   ...

19         A[i][j]= max(A[i-1][j],A[i-1][j-w[i-1]]+c[i-1]);
     cout<<A[liczba][W];
21
     //usuwanie z pamieci tablicy A
23   for(unsigned int i=0;i<=liczba;i++)
       delete [] A[i];
25   delete [] A;
   }
```

Wcześniej musimy jednak sprawdzić, czy `j>=w[i]`

Listing 5.9. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){
  //tworzenie tablicy A
  unsigned int ** A = new unsigned int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    A[i] = new unsigned int[W+1];
  //inicjowanie tablicy A
  for(unsigned int i = 0; i<=liczba;i++)
    A[i][0] = 0;
  for(unsigned int i = 0; i<=W;i++)
    A[0][i] = 0;

  //wlasciwa czesc algorytmu
  for(unsigned int i=1;i<=liczba;i++)
    for(unsigned int j=1;j<=W;j++)
      if (j<w[i-1])
        A[i][j]=A[i-1][j];
      else
        A[i][j]= max(A[i-1][j],A[i-1][j-w[i-1]]+c[i-1]);
  cout<<A[liczba][W];

  //usuwanie z pamieci tablicy A
  for(unsigned int i=0;i<=liczba;i++)
    delete [] A[i];
  delete [] A;
}
```

Złożoność czasowa powyższej funkcji to $O($`liczba`$\cdot$`W`$)$, co wynika z faktu, że najbardziej pracochłonną częścią programu są zagnieżdżone pętle `for` przebiegające jedna `liczba`, a druga `W` wartości. Wbrew pozorom nie jest to złożoność wielomianowa, gdyż długość zapisu liczby W jest $O(\log W)$.

Jeżeli chcemy wypisać optymalny wybór, a nie tylko jego wartość, to będziemy potrzebowali do tego dodatkowej tablicy `B`. Komórka `B[i][j]` będzie przechowywała informację o przedmiocie o największym numerze należącym do optymalnego wyboru w podproblemie o parametrach i, j. W przypadku gdy dla jakiegoś podproblemu rozwiązanie nie zawiera żadnego przedmiotu odpowiadająca mu komórka tablicy `B` będzie miała wartość -1. Zauważmy, że gdy `A[i][j]=A[i-1][j]`, to `B[i][j]=B[i-1][j]` (bo w obu przypadkach możemy wziąć tą samą listę przedmiotów), natomiast, gdy `A[i][j]=A[i-1][j-w[i-1]]+c[i-1]`, to `B[i][j]=i-1` (bo wtedy przedmiot o numerze `i-1` należy do optymalnego wyboru i skoro rozważamy tylko pierwszych `i` przedmiotów, to ma w tym wyborze najwyższy numer).

Wersja funkcji `plecak` wypisująca optymalny wybór przedmiotów przy każdej zmianie wartości komórki tablicy `A` będzie zmieniać wartość odpowiedniej komórki tablicy `B`. Ponieważ od tego, czy w 19 linii listingu 5.9

Tabela 5.3. Wypełnienie tablicy B dla danych z Przykładu 5.1.

	0	**1**	**2**	**3**	**4**
0	−1	−1	−1	−1	−1
1	−1	−1	−1	0	0
2	−1	−1	1	0	0
3	−1	2	1	0	2
4	−1	2	1	0	2
5	−1	2	1	0	4

komórce `A[i][j]` nadamy wartość `A[i-1][j]` czy `A[i-1][j-w[i-1]]+c[i-1]`, zależy wartość `B[i][j]`, to najpierw zmodyfikujemy funkcję z listingu 5.9 w taki sposób, żeby używała operacji warunkowych zamiast funkcji `max`.

Listing 5.10. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){
  //tworzenie tablicy A
  unsigned int ** A = new unsigned int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    A[i] = new unsigned int[W+1];
  //inicjowanie tablicy A
  for(unsigned int i = 0; i<=liczba;i++)
    A[i][0] = 0;
  for(unsigned int i = 0; i<=W;i++)
    A[0][i] = 0;

  //wlasciwa czesc algorytmu
  for(unsigned int i=1;i<=liczba;i++)
    for(unsigned int j=1;j<=W;j++)
      if ((j<w[i-1]) ||
          ((A[i-1][j]>=A[i-1][j-w[i-1]]+c[i-1]))
        A[i][j]=A[i-1][j];
      else
        A[i][j]= A[i-1][j-w[i-1]]+c[i-1];
  cout<<A[liczba][W];

  //usuwanie z pamieci tablicy A
  for(unsigned int i=0;i<=liczba;i++)
    delete [] A[i];
  delete [] A;
}
```

Zwróćmy uwagę, że w powyższym listingu do drugiej części alternatywy z warunku instrukcji `if` wejdziemy wyłącznie wtedy, gdy `j>=w[i-1]` (tak

działa alternatywa w języku C++). Jest to ważne, gdyż w przeciwnym wypadku `j-w[i-1]` jest liczbą ujemną i co za tym idzie nie jest poprawnym indeksem tablicy.

Teraz już łatwo przekształcić funkcję `plecak` w taki sposób, żeby wypisywała optymalny wybór przedmiotów. Samo wypisywanie wyników przenieśliśmy do oddzielnej funkcji, żeby nie komplikować niepotrzebnie funkcji `plecak`:

Listing 5.11. Problem plecakowy – programowanie dynamiczne

```
void plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W){
  //tworzenie tablic
  unsigned int ** A = new unsigned int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    A[i] = new unsigned int[W+1];
  int ** B = new int[liczba+1];
  for(unsigned int i=0;i<=liczba;i++)
    B[i] = new int[W+1];
  //inicjowanie tablic
  for(unsigned int i = 0; i<=liczba;i++)
    A[i][0] = B[i][0] = 0;
  for(unsigned int i = 0; i<=W; i++)
    A[0][i] = B[0][i] = 0;

  //wlasciwa czesc algorytmu
  for(unsigned int i=1;i<=liczba;i++)
    for(unsigned int j=1;j<=W;j++)
      if ((j<w[i-1]) ||
          ((A[i-1][j]>=A[i-1][j-w[i-1]]+c[i-1]))){
        A[i][j]=A[i-1][j];
        B[i][j]=B[i-1][j];
      }
      else {
        A[i][j]= A[i-1][j-w[i-1]]+c[i-1];
        B[i][j]=i-1;
      }
  Wypisz(B,w,liczba,W);

  //zwalnianie pamieci po tablicach
  for(unsigned int i=0;i<=liczba;i++)
    delete [] A[i];
  delete [] A;
  for(unsigned int i=0;i<=liczba;i++)
    delete [] B[i];
  delete [] B;
}
```

Chcąc wypisać optymalny wybór przedmiotów zaczynamy od tyłu

– `B[liczba][W]` zawiera numer ostatniego przedmiotu z listy wybranych. Aby wypisać przedostatni przedmiot musimy wziąć `liczba` równe numerowi ostatniego przedmiotu (czyli `liczba=B[liczba][W]`), ograniczenie na wagę pomniejszone o wagę ostatnio wypisanego przedmiotu (`W=W-w[liczba]`) i wypisać wartość `B[liczba][W]`. Postępujemy w ten sposób tak długo, dopóki `B[liczba][W]` jest większe od -1.

Listing 5.12. Problem plecakowy – programowanie dynamiczne

```
1 void Wypisz(int ** B, unsigned int w[],
      unsigned int liczba, unsigned int W){
3   while(B[liczba][W]!=-1){
      cout<<B[liczba][W]<<endl;
5     liczba = B[liczba][W];
      W = W - w[liczba];
7   }
  }
```

Na koniec zobaczymy na przykładzie programu rozwiązującego problem plecakowy, jak można zaoszczędzić pamięć poprzez ograniczenie jednego z wymiarów. Zrobimy to na przykładzie funkcji z listingu 5.9, gdyż w przypadku wersji algorytmu wypisującej optymalny wybór przedmiotów i tak nie ma możliwości zaoszczędzenia pamięci na tablicy `B`. W przypadku rozważanej funkcji tylko pierwszy wymiar możemy ograniczyć. Wynika to z tego, że w pierwszym wymiarze wzór rekurencyjny cofa się tylko o jeden, zaś w drugim wymiarze wzór rekurencyjny może sięgać dowolnie daleko do tyłu (we wzorze odejmujemy `w[i-1]`). O ile we wcześniejszych wersjach algorytmu kolejność pętli przebiegających po pierwszej i drugiej współrzędnej tablicy `A` była obojętna, to teraz zmienna sterująca najbardziej zewnętrznej pętli musi przebiegać po wymiarze, który skracamy.

Listing 5.13. Problem plecakowy – programowanie dynamiczne

```
  void plecak(unsigned int c[], unsigned int w[],
2             unsigned int liczba, unsigned int W){
    //tworzenie i inicjowanie tablicy A
4   unsigned int ** A = new unsigned int[2];
    A[0] = new unsigned int[W+1];
6   A[1] = new unsigned int[W+1];
    for(unsigned int i = 0; i<=W; i++)
8     A[0][i] = 0;
    A[1][0] = 0;
10
    //wlasciwa czesc algorytmu
12  for(unsigned int i=1;i<=liczba;i++)
      for(unsigned int j=1;j<=W;j++)
14      if (j<w[i-1])
```

```
        A[i%2][j]=A[(i-1)%2][j];
16     else
        A[i%2][j]= max(A[(i-1)%2][j],
18                     A[(i-1)%2][j-w[i-1]]+c[i-1]);
   cout<<A[liczba%2][W];
20
   //usuwanie z pamieci tablicy A
22 delete [] A[0];
   delete [] A[1];
24 delete [] A;
  }
```

5.1.2. Najdłuższy wspólny podciąg

Przyjrzyjmy się jeszcze jednemu przykładowi wykorzystania programowania dynamicznego. Tym razem będzie to rozwiązanie problemu najdłuższego wspólnego podciągu dwóch słów. Zaczniemy od wprowadzenia pojęć potrzebnych do zdefiniowania problemu. Słowem `a` nad alfabetem Σ nazywamy dowolny skończony ciąg elementów ze zbioru Σ. Dla uproszenia będziemy traktowali słowa jako tablice o elementach typu `char`. Niech `a` będzie słowem, przez `a[i..j]` będziemy oznaczać słowo `a[i]a[i+1]...a[j-1]a[j]` (słowo złożone ze wszystkich znaków słowa `a` od znaku o indeksie `i` do znaku o indeksie `j`). Podciągiem danego słowa `a` nazywamy dowolne słowo `b`, takie że istnieje rosnąca funkcja `f`, taka że `b[i]=a[f(i)]` dla `i` od `0` do `|b|-1` (przez `|b|` oznaczamy liczbę znaków słowa `b`). Mówiąc mniej formalnie podciąg słowa `a` jest dowolnym ciągiem znaków ze słowa `a` zachowującym ich oryginalną kolejność. Przykładowo „cce" jest podciągiem słowa „cdce", zaś „cec" nie jest podciągiem tego słowa, gdyż w oryginalnym słowie „e" występuje po obu wystąpieniach „c".

W tym podrozdziale pokażemy rozwiązanie następującego problemu.

Problem 5.2.
Wejście *Dwa słowa* `a` *i* `b`.
Wyjście *Najdłuższy wspólny podciąg słów* `a` *i* `b`*(najdłuższy skończony ciąg, który jest jednocześnie podciągiem słowa* `a` *i słowa* `b`*)*

Zanim spróbujemy stworzyć algorytm rozwiązujący powyższy problem przeanalizujmy następujący przykład:

Przykład 5.2. *Dla danych wejściowych* `a`*=„efcde" i* `b`*=„cedfef" istnieją trzy wspólne podciągi o tej samej maksymalnej długości: „efe", „ede" i „cde".*

Najpierw rozwiążemy uproszczoną wersję problemu 5.2, w której pytamy się tylko o długość najdłuższego wspólnego podciągu dwóch słów. Pierwsza

rzecz, jaką musimy zrobić, to zdefiniować podproblemy, jakie będziemy rozwiązywać. Pamiętamy, że najwygodniej jest sparametryzować podproblemy liczbami całkowitymi. W naszym przypadku dla danych dwóch słów `a` i `b` podproblemem o parametrach `i,j` będzie problem najdłuższego wspólnego podciągu słów `a[0..i-1]` i `b[0..j-1]`. Rozwiązania poszczególnych podproblemów będziemy zapisywali w dwuwymiarowej tablicy `A` (komórka `A[i][j]` będzie zawierała rozwiązanie problemu o parametrach `i,j`). Oczywiście `A[i][0]=A[0][i]=0`, zaś `A[|a|][|b|]` zawiera rozwiązanie całego problemu.

Zastanówmy się teraz nad rekurencyjną zależnością, jaka zachodzi pomiędzy zdefiniowanymi podproblemami. Rozważmy, ile może być równe `A[i][j]` dla `i,j>0`. Jeżeli `a[i-1]=b[j-1]`, to możemy założyć, że `a[i]` należy do najdłuższego wspólnego podciągu słów `a[0..i-1]` i `b[0..j-1]` i co za tym idzie `A[i][j]=A[i-1][j-1]+1`. Jeżeli `a[i-1]` i `b[j-1]` są różne, to wiadomo, że jeden z tych znaków nie należy do najdłuższego wspólnego podciągu, co implikuje, że `A[i][j]=max(A[i-1][j],A[i][j-1])`.

Tabela 5.4. Wypełnienie tablicy A dla danych z Przykładu 5.2.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	1	2	2	2
3	0	1	1	1	2	2	2
4	0	1	1	2	2	2	2
5	0	1	2	2	2	3	3

Teraz zaimplementujemy omówione powyżej rozwiązanie. Napiszemy funkcję, która jako argumenty dostanie tablice zawierające słowa, których najdłuższego wspólnego podciągu szukamy oraz rozmiary tych tablic.

Listing 5.14. Najdłuższy wspólny podciąg

```
1 void podciag(char a[], unsigned int rozmiara,
               char b[], unsigned int rozmiarb){
3
    ...
5
  }
```

Wyniki będziemy zapisywali w dwuwymiarowej tablicy `A` o wymiarach `rozmiara+1` na `rozmiarb+1`, której komórka `A[rozmiara][rozmiarb]` będzie na koniec zawierała rozwiązanie

Listing 5.15. Najdłuższy wspólny podciąg

```
  void podciag(char a[], unsigned int rozmiara,
2              char b[], unsigned int rozmiarb){
    \\tworzenie tablicy A
4   unsigned int ** A = new unsigned int*[rozmiara+1];
    for(unsigned int i=0;i<=rozmiara;i++)
6     A[i] = new unsigned int[rozmiarb+1];

8   ...

10  cout<<A[rozmiara][rozmiarb]<<endl;

12  //usuwanie z pamieci tablicy A
    for(unsigned int i=0;i<=rozmiara;i++)
14    delete [] A[i];
    delete [] A;
16 }
```

Jak wcześniej zauważyliśmy `A[i][0]=A[0][i]=0`.

Listing 5.16. Najdłuższy wspólny podciąg

```
  void podciag(char a[], unsigned int rozmiara,
2              char b[], unsigned int rozmiarb){
    \\tworzenie tablicy A
4   unsigned int ** A = new unsigned int*[rozmiara+1];
    for(unsigned int i=0;i<=rozmiara;i++)
6     A[i] = new unsigned int[rozmiarb+1];
    \\inicjowanie tablicy A
8   for(unsigned int i=0;i<=rozmiara;i++)
      A[i][0] = 0;
10  for(unsigned int i=0;i<=rozmiarb;i++)
      A[0][i] = 0;
12
    ...
14
    cout<<A[rozmiara][rozmiarb]<<endl;
16
    \\usuwanie z pamieci tablicy A
18  for(unsigned int i=0;i<=rozmiara;i++)
      delete [] A[i];
20  delete [] A;
  }
```

Tablicę `A` będziemy wypełniali w zagnieżdżonych pętlach `for`, z których jedna będzie przebiegać po pierwszym, a druga po drugim wymiarze tablicy `A`. Kolejność pętli jest w tym przypadku obojętna:

Listing 5.17. Najdłuższy wspólny podciąg

```
1 void podciag(char a[], unsigned int rozmiara,
              char b[], unsigned int rozmiarb){
3   \\tworzenie tablicy A
    unsigned int ** A = new unsigned int*[rozmiara+1];
5   for(unsigned int i=0;i<=rozmiara;i++)
      A[i] = new unsigned int[rozmiarb+1];
7   \\inicjowanie tablicy A
    for(unsigned int i=0;i<=rozmiara;i++)
9     A[i][0] = 0;
    for(unsigned int i=0;i<=rozmiarb;i++)
11    A[0][i] = 0;

13  \\wlasciwa czesc algorytmu
    for(unsigned int i=1;i<=rozmiara;i++)
15    for(unsigned int j=1;j<=rozmiarb;j++)

17  ...

19  cout<<A[rozmiara][rozmiarb]<<endl;

21  \\usuwanie z pamieci tablicy A
    for(unsigned int i=0;i<=rozmiara;i++)
23    delete [] A[i];
    delete [] A;
25 }
```

Wewnątrz pętli `for` użyjemy naszego wzoru rekurencyjnego do wypełnienia tabeli `A`.

Listing 5.18. Najdłuższy wspólny podciąg

```
1 void podciag(char a[], unsigned int rozmiara,
              char b[], unsigned int rozmiarb){
3   \\tworzenie tablicy A
    unsigned int ** A = new unsigned int*[rozmiara+1];
5   for(unsigned int i=0;i<=rozmiara;i++)
      A[i] = new unsigned int[rozmiarb+1];
7   \\inicjowanie tablicy A
    for(unsigned int i=0;i<=rozmiara;i++)
9     A[i][0] = 0;
    for(unsigned int i=0;i<=rozmiarb;i++)
11    A[0][i] = 0;

13  \\wlasciwa czesc algorytmu
    for(unsigned int i=1;i<=rozmiara;i++)
15    for(unsigned int j=1;j<=rozmiarb;j++)
        if (a[i-1]==b[j-1])
17        A[i][j] = A[i-1][j-1] + 1;
        else if (A[i-1][j]<A[i][j-1])
```

```
19            A[i][j]=A[i][j-1];
          else
21            A[i][j]=A[i-1][j];
      cout<<A[rozmiara][rozmiarb]<<endl;
23
      \\usuwanie z pamieci tablicy A
25    for(unsigned int i=0;i<=rozmiara;i++)
        delete [] A[i];
27    delete [] A;
    }
```

Aby móc wypisać najdłuższy wspólny podciąg dwóch słów, a nie tylko jego długość, użyjemy dodatkowej tablicy `B`, której komórka `B[i][j]` będzie przechowywała indeks w słowie `a[0..i-1]` ostatniego elementu najdłuższego wspólnego ciągu słów `a[0..i-1]` i `b[0..j-1]`. To, jaką wartość wstawimy do komórki `B[i][j]`, zależy od tego, który z przypadków wybraliśmy przy wypełnianiu komórki `A[i][j]`. Jeżeli `a[i-1]==b[j-1]`, to `B[i][j]=i-1`. W przeciwnym razie w zależności od tego, czy `A[i-1][j]<A[i][j-1]` do `B[i][j]`, przepisujemy `B[i][j-1]` lub `B[i-1][j]`. W przypadku gdy `A[i][j]` jest równy `0` (a więc najdłuższy wspólny podciąg jest pusty), element `B[i][j]` będzie miał wartość `-1`.

Tabela 5.5. Wypełnienie tablicy B dla danych z Przykładu 5.2.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	0	0	0	0	0
2	-1	-1	0	0	1	1	1
3	-1	2	0	0	1	1	1
4	-1	2	0	3	1	1	1
5	-1	2	4	3	1	4	4

Listing 5.19. Najdłuższy wspólny podciąg – wypisywanie podciągu

```
    void podciag(char a[], unsigned int rozmiara,
2                char b[], unsigned int rozmiarb){
      \\tworzenenie tablic A i B
4     unsigned int ** A = new unsigned int*[rozmiara+1];
      unsigned int ** B = new unsigned int*[rozmiara+1];
6     for(unsigned int i=0;i<=rozmiara;i++){
        A[i] = new unsigned int[rozmiarb+1];
8       B[i] = new unsigned int[rozmiarb+1];
      }
10    //inicjowanie tablic A i B
      for(unsigned int i=0;i<=rozmiara;i++){
12      A[i][0] = 0;
```

```
    B[i][0] = -1;
14  }
    for(unsigned int i=0;i<=rozmiarb;i++){
16    A[0][i] = 0;
      B[0][i] = -1;
18  }

20  //wlasciwa czesc algorytmu
    for(unsigned int i=1;i<=rozmiara;i++)
22    for(unsigned int j=1;j<=rozmiarb;j++)
        if (a[i-1]==b[j-1]){
24        A[i][j] = A[i-1][j-1] + 1;
          B[i][j] = i-1;
26      }
        else if (A[i-1][j]<A[i][j-1]){
28        A[i][j]=A[i][j-1];
          B[i][j]=B[i][j-1];
30      }
        else{
32        A[i][j]=A[i-1][j];
          B[i][j]=B[i-1][j];
34      }
    wypisz(B,a,rozmiara, b, rozmiarb);
36
    //usuwanie z pamieci tablic A i B
38  for(unsigned int i=0;i<=rozmiara;i++){
      delete [] A[i];
40    delete [] B[i];
    }
42  delete [] A;
    delete [] B;
44 }
```

Wypisywanie wyniku ze względu na czytelność przenieśliśmy do funkcji `wypisz`. Idea algorytmu wypisywania jest taka, że najpierw szukamy ostatniego znaku największego wspólnego podciągu słów `a` i `b` oraz indeksów, pod którymi występuje w słowach `a` i `b` (niech będą to miejsca o indeksach `i` oraz `j`). Następnie szukamy ostatniego elementu największego wspólnego podciągu słów `a[0..i-1]` i `b[0..j-1]` itd.

Bardziej dokładnie – aby poznać ostatni znak szukanego podciągu – wystarczy sprawdzić `a[B[rozmiara][rozmiarb]]`. Żeby poznać przedostatni znak tego podciągu trzeba wziąć `rozmiara=B[rozmiara][rozmiarb]`, największe `rozmiarb` mniejsze od dotychczasowej wartości takie, że `b[rozmiarb]=a[rozmiara]` i wtedy znowu wziąć `a[B[rozmiara][rozmiarb]]` itd. W ten sposób uzyskamy poszukiwany ciąg, ale w odwróconej kolejności. Aby poprawnie go wypisać użyjemy stosu do odwrócenia kolejności znaków.

Przeanalizujmy działanie powyższego algorytmu na przykładzie danych z Przykładu 5.2 i wygenerowanej dla nich tablicy `B` prezentowanej w Tabeli 5.5.

Przykład 5.3. *Ostatnim znakiem znalezionego najdłuższego wspólnego podciągu jest znak, który w tablicy przechowującej słowo* `a` *ma indeks* `B[5][6]`*. Jest to znak o indeksie* 4 *czyli „e”. Teraz szukamy ostatniego wystąpienia „e” w słowie* `b` *o indeksie mniejszym niż* 6*. To wystąpienie ma indeks* 4*.*

Zgodnie z algorytmem kolejnym znakiem znalezionego podciągu jest znak, który w słowie `a` *ma indeks* `B[4][4]=1`*. Znakiem tym jest znak „f”. Indeks ostatniego wystąpienia „f” w słowie* `b` *o indeksie mniejszym niż* 4 *jest równy* 3 *(zwróćmy uwagę, że nie jest to ostatnie wystąpienie „f” w słowie b).*

W kolejnym kroku algorytmu wyznaczamy następny element najdłuższego wspólnego podciągu słów `a` *i* `b`*. Analogicznie jak poprzednio robimy to sprawdzając znak słowa* `a` *o indeksie* `B[1][3]=0`*. Ten znak to „e”, zaś indeks ostatniego wystąpienia „e” w słowie* `b` *o indeksie mniejszym niż* 3 *to* 1*.*

Na koniec dostajemy, że `B[0][1]=-1`*, co oznacza, że wygenerowaliśmy już wszystkie znaki najdłuższego wspólnego podciągu słów* `a` *i* `b`*. Teraz wystarczy odwrócić kolejność wygenerowanych znaków (w naszym przypadku ze względu na symetryczność otrzymanego podciągu niczego to nie zmienia) i otrzymujemy najdłuższy wspólny podciąg: „efe”.*

Kod implementujący powyższy algorytm wygląda następująco:

Listing 5.20. Najdłuższy wspólny podciąg

```
void wypisz(unsigned int ** B, char a[],
                         unsigned int rozmiara,
                  char b[], unsigned int rozmiarb){
   Stos s;
   while(B[rozmiara][rozmiarb]>=0){
      rozmiara=B[rozmiara][rozmiarb];
      s.push(a[rozmiara]);
      do
         rozmiarb--;
      while(b[rozmiarb]!=a[rozmiara]);

   }
   while(!s.empty()){
      cout<<s.top();
      s.pop();
   }
```

Na koniec zobaczymy modyfikację programu 5.18, w której ograniczymy rozmiar pierwszego wymiaru tablicy `A`. Ponieważ we wzorze rekurencyjnym

sięgamy co najwyżej o jedną komórkę do tyłu, wystarczy, aby pierwszy wymiar miał rozmiar 2.

Listing 5.21. Najdłuższy wspólny podciąg

```
  void podciag(char a[], unsigned int rozmiara,
2               char b[], unsigned int rozmiarb){
   \\tworzenie tablicy A
4  unsigned int ** A = new unsigned int*[2];
   A[0] = new unsigned int[]rozmiarb+1
6  A[i] = new unsigned int[rozmiarb+1];
   \\inicjowanie tablicy A
8  for(unsigned int i=0;i<=rozmiarb;i++)
     A[0][i] = 0;
10 A[1][0] = 0;

12 \\wlasciwa czesc algorytmu
   for(unsigned int i=1;i<=rozmiara;i++)
14   for(unsigned int j=1;j<=rozmiarb;j++)
       if (a[i-1]==b[j-1])
16       A[i%2][j] = A[(i-1)%2][j-1] + 1;
       else if (A[(i-1)%2][j]<A[i%2][j-1])
18       A[i%2][j]=A[i%2][j-1];
       else
20       A[i%2][j]=A[(i-1)%2][j];
   cout<<A[rozmiara%2][rozmiarb]<<endl;
22
   \\usuwanie z pamieci tablicy A
24 for(unsigned int i=0;i<=rozmiara;i++)
     delete [] A[i];
26 delete [] A;
  }
```

Zadanie 22. Napisz program rozwiązujący przy pomocy programowania dynamicznego problem wydawania reszty zdefiniowany w rozdziale 6 jako problem 6.1.

Zadanie 23. Napisz funkcję, która dla otrzymanych w argumencie tablicy dwuwymiarowej `tab` o wymiarach 3×3 oraz liczb n i a, szuka takiego ciągu liczb `x(1)`, `x(2)`, ..., `x(n)`, żeby spełniony był następujący warunek:
`tab[x(1)][tab[x(2)][...tab[x(n-1)][x(n)]...]]=a`.
W funkcji wykorzystaj programowanie dynamiczne.

5.2. Metoda spamiętywania

Metoda spamiętywania jest odmianą programowania dynamicznego, w której odwracamy kolejność rozwiązywania problemów. Zamiast zaczynać

od dołu i rozwiązywać kolejne podproblemy od tych najmniejszych do coraz większych, w metodzie spamiętywania zaczynamy od całego problemu, żeby w kolejnych rekurencyjnych wywołaniach funkcji zejść do najmniejszych podproblemów. Aby nie rozwiązywać wielokrotnie tych samych podproblemów wszystkie otrzymane wyniki zapisujemy do globalnej tablicy. Asymptotyczna pesymistyczna złożoność algorytmów stosujących metodę spamiętywania jest taka sama jak analogicznych algorytmów stosujących klasyczne programowanie dynamiczne, choć ze względu na narzut czasowy wynikający z zastosowania rekurencji, implementacje metody spamiętywania mogą być w praktyce wolniejsze od ich dynamicznych odpowiedników. Metoda spamiętywania ma za to przewagę, gdy do obliczenia końcowego wyniku nie trzeba rozwiązywać wszystkich podproblemów. W takiej sytuacji w metodzie spamiętywania rozwiązujemy tylko te podproblemy, których rozwiązanie jest konieczne, podczas gdy algorytmy programowania dynamicznego rozwiązują wszystkie podproblemy po kolei.

5.2.1. Problem plecakowy

Przeanalizujemy użycie metody spamiętywania na przykładzie dyskretnego problemu plecakowego, który rozwiązywaliśmy w poprzednim podrozdziale przy pomocy programowania dynamicznego. Dla uproszczenia będziemy rozwiązywali wersję problemu, w której interesuje nas tylko wartość przedmiotów w optymalnym upakowaniu (a nie lista przedmiotów w tym upakowaniu).

Nasza funkcja dostanie jako argumenty instancję problemu (w takiej samej formie jak w przypadku dynamicznego rozwiązania problemu) oraz dwuwymiarową tablicę `A` do zapisywania wyników. Zakładamy, że elementy tablicy `A` mają początkowo wartość `-1`. Dzięki temu będziemy mogli rozpoznać, które podproblemy już zostały rozwiązane.

Listing 5.22. Dyskretny problem plecakowy – metoda spamiętywania

```
1 void Plecak(unsigned int c[], unsigned int w[],
              unsigned int liczba, unsigned int W,
3             int ** A){

5   ...

7 }
```

W pierwszej kolejności sprawdzimy, czy dany podproblem nie został jeszcze rozwiązany.

Listing 5.23. Dyskretny problem plecakowy – metoda spamiętywania

```
void Plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W,
            int ** A){
  if (A[liczba][W]>=0)
    return A[liczba][W];

  ...

}
```

Następnie sprawdzimy warunki brzegowe (czyli przypadki gdy `liczba` lub `W` jest równe zero).

Listing 5.24. Dyskretny problem plecakowy – metoda spamiętywania

```
void Plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W,
            int ** A){
  if (A[liczba][W]>=0)
    return A[liczba][W];
  if ((liczba==0)||(W==0))
    return 0;

  ...

}
```

W kolejnym kroku sprawdzimy, czy ostatni przedmiot zmieści się sam w plecaku.

Listing 5.25. Dyskretny problem plecakowy – metoda spamiętywania

```
void Plecak(unsigned int c[], unsigned int w[],
            unsigned int liczba, unsigned int W,
            int ** A){
  if (A[liczba][W]>=0)
    return A[liczba][W];
  if ((liczba==0)||(W==0))
    return 0;
  if (W < w[liczba-1]){
    A[liczba][W]=Plecak(c,w,liczba-1,W,A);
    return A[liczba][W]
  }

  ...

}
```

Na koniec policzymy maksimum z dwóch przypadków: gdy ostatni przedmiot należy i nie należy do optymalnego wyboru.

Listing 5.26. Dyskretny problem plecakowy – metoda spamiętywania

```
1 void Plecak(unsigned int c[], unsigned int w[],
              unsigned int liczba, unsigned int W,
3             int ** A){
    if (A[liczba][W]>=0)
5     return A[liczba][W];
    if ((liczba==0)||(W==0))
7     return 0;
    if (W < w[liczba-1]){
9     A[liczba][W]=Plecak(c,w,liczba-1,W,A);
      return A[liczba][W]
11  }
    unsigned int pom1, pom2;
13  pom1 = Plecak(c,w,liczba-1,W,A);
    pom2 = Plecak(c,w,liczba-1,W- w[liczba-1],A)+c[liczba-1];
15  A[liczba][W] = max(pom1,pom2);
    return A[liczba][W];
17 }
```

Zadanie 24. Rozwiąż zadania z programowania dynamicznego przy pomocy metody spamiętywania.

Rozdział 6

Algorytmy zachłanne

6.1. O metodzie **122**
6.2. Przykłady **122**
6.2.1. Problem wydawania reszty 122
6.2.2. Problem plecakowy 126

6.1. O metodzie

Algorytmem zachłannym nazywamy algorytm, który sprowadza rozwiązanie problemu do dokonania serii wyborów i każdego z tych wyborów dokonuje, wybierając – w jakimś sensie – lokalnie najlepsze rozwiązanie. Algorytmy zachłanne często stosujemy w problemach optymalizacyjnych, w których szukamy najlepszego ze zbioru dopuszczalnych rozwiązań. Nie dla wszystkich takich problemów algorytmy zachłanne dają optymalne rozwiązanie, ale często dla trudnych obliczeniowo problemów dają akceptowalne rozwiązania w rozsądnym czasie. Zazwyczaj dodatkową zaletą algorytmów zachłannych jest ich prostota.

Często nieświadomie stosujemy algorytmy zachłanne. Przykładowo:

- Wydając resztę dajemy zwykle najwyższy nominał mieszczący się w reszcie, następnie najwyższy nominał mieszczący się w pozostałej części reszty itd. W ten sposób próbujemy minimalizować liczbę wydanych banknotów i monet.
- Chcąc dojść w nieznanym terenie do widocznego w oddali obiektu, staramy się wybierać drogi o kierunku jak najbardziej zbliżonym do kierunku, w którym znajduje się obiekt, do którego zmierzamy. Takie rozwiązanie nie zawsze daje optymalne rozwiązanie, gdyż wybrana droga może po pewnym czasie skręcić w niepożądanym kierunku.
- Ludzie kupując w sklepach często kierują się niższą ceną za opakowanie lub większymi rozmiarami opakowania. W ten sposób w ciągu kilku lat typowa kostka masła zmalała z 250g do 200g.

6.2. Przykłady

6.2.1. Problem wydawania reszty

Przyjrzyjmy się teraz problemowi wydawania reszty. Możemy go zdefiniować w następujący sposób:

Problem 6.1.
Wejście *Dostępne nominały oraz kwota do wydania.*
Wyjście *Sposób wydania zadanej kwoty przy pomocy minimalnej możliwej liczby monet i banknotów.*

Przeanalizujmy rozwiązanie problemu wydawania reszty dla przykładowej instancji:

Przykład 6.1.
Załóżmy, że chcemy wydać 6zł i mamy dostępne wszystkie nominały monet.

Podaną resztę możemy wydać na wiele sposobów. Przykładowo możemy wydać 3 monety po 2zł lub 6 monet jednozłotowych. Optymalnym rozwiązaniem zadania jest w tym przypadku wydanie jednej monety 5zł i jednej monety 1zł. Wydajemy w ten sposób resztę przy pomocy dwóch monet. Ponieważ nie istnieje moneta szcześciozłotowa, to nie da się wydać 6zł przy pomocy mniejszej liczby monet.

Najpopularniejszy algorytm zachłanny rozwiązujący problem wydawania reszty działa następująco:

Algorytm 6.1.

1. *Wczytaj dostępne nominały i wydawaną kwotę w.*
2. *Dopóki $w > 0$ powtarzaj kroki od 3 do 5.*
3. *Wybierz n największy spośród dostępnych nominałów, który jest mniejszy od w*
4. *Wypisz n.*
5. *$w = w - n$.*

Zobaczmy jak powyższy algorytm będzie działał dla danych z przykładu 6.1:

Przykład 6.2.
Na początku w jest równe 6. Największym nominałem mniejszym lub równym 6 jest 5, więc po pierwszym obrocie pętli sytuacja wygląda następująco:

$$n = 5, \; w = w - n = 6 - 5 = 1.$$

Po drugim obrocie pętli mamy:

$$n = 1, \; w = 1 - 1 = 0.$$

Po dwóch obrotach pętli algorytm kończy swoje działanie, gdyż nie jest spełniony warunek z punktu 2 (w nie jest większe od 0).

Zaimplementujemy teraz algorytm 6.1. Napiszemy funkcję, która otrzyma jako argumenty tablicę z listą nominałów (zakładamy, że tablica jest posortowana rosnąco), jej rozmiar oraz kwotę do wydania i wypisze na standardowym wyjściu listę wydawanych nominałów.

Listing 6.1. Wydawanie reszty – algorytm zachłanny

```
1 void Reszta(unsigned int nominaly, unsigned int rozmiar,
                                    unsigned int kwota){
3     ...
  }
```

Główną częścią programu będzie pętla, która będzie się kręcić tak długo jak długo pozostanie jeszcze jakaś kwota do wydania.

Listing 6.2. Wydawanie reszty – algorytm zachłanny

```
void Reszta(unsigned int tab_nomin[], unsigned int rozmiar,
                                      unsigned int kwota){
  ...
  while(kwota>0){
    ...
  }
}
```

Wewnątrz pętli `while` będziemy robić trzy rzeczy. Najpierw będziemy szukali największego nominału mniejszego równego od wydawanej kwoty (skorzystamy tutaj z faktu, że kolejne wydawane nominały nie będą większe od poprzednich).

Listing 6.3. Wydawanie reszty – algorytm zachłanny

```
void Reszta(unsigned int tab_nomin[], unsigned int rozmiar,
                                      unsigned int kwota){
  unsigned int nominal=rozmiar-1;
  while(kwota>0){
    while((nominal>0)&&(tab_nomin[nominal]>kwota))
      nominal--;
    ...
  }
}
```

Następnie będziemy wypisywali znaleziony nominał i odejmowali go od wydawanej kwoty.

Listing 6.4. Wydawanie reszty – algorytm zachłanny

```
void Reszta(unsigned int tab_nomin[], unsigned int rozmiar,
                                      unsigned int kwota){
  unsigned int nominal=rozmiar-1;
  while(kwota>0){
    while((nominal>=0)&&(tab_nomin[nominal]>kwota))
      nominal--;
    if (nominal>=0){
      cout<<tab_nomin[nominal]<<endl;
      kwota = kwota - tab_nomin[nominal];
    }

    ...

  }
}
```

Na koniec rozważymy przypadek, gdy danej reszty nie da się wydać algorytmem zachłannym. W takiej sytuacji, w pewnym momencie wszystkie posiadane nominały będą większe od wydawanej kwoty i zmienna `nominal` po wyjściu z wewnętrznej pętli `while` będzie miała wartość `-1`.

Listing 6.5. Wydawanie reszty – algorytm zachłanny

```
void Reszta(unsigned int tab_nomin[], unsigned int rozmiar,
                                       unsigned int kwota){
  unsigned int nominal=rozmiar-1;
  while((kwota>0)&&(nominal>=0)){
    while((nominal>=0)&&(tab_nomin[nominal]>kwota))
      nominal--;
    if (nominal>=0){
      cout<<tab_nomin[nominal]<<endl;
      kwota = kwota - tab_nomin[nominal];
    }
    else
      cout<<"Nie moge wydac reszty"<<endl;
  }
}
```

Nie dla każdego zestawu danych przedstawiony algorytm zachłanny zwraca optymalne rozwiązanie:

Przykład 6.3.

- *Załóżmy, że mamy do wydania kwotę 6zł przy pomocy nominałów 1gr, 2zł i 5zł.*
- *Optymalnym rozwiązaniem jest wydanie reszty przy pomocy trzech monet dwuzłotowych.*
- *Algorytm zachłanny w tym przypadku wyda resztę przy pomocy pięciozłotówki oraz 100 monet po 1gr (101 monet wobec trzech monet w optymalnym rozwiązaniu).*

Systemy monetarne w większości państw w tym w Polsce są tak skonstruowane, żeby przy dostępnych wszystkich nominałach algorytm zachłanny wydawania reszty dawał optymalne rozwiązanie. Przykładowo jednak jeszcze do roku 1971 w brytyjskim systemie monetarnym nie działał zachłanny algorytm wydawania reszty.

Aby rozwiązać problem wydawania reszty w ogólnym przypadku trzeba użyć algorytmu innego niż zachłanny. Najpopularniejszy taki algorytm wykorzystuje programowanie dynamiczne.

6.2.2. Problem plecakowy

W rozdziale 5 poznaliśmy rozwiązanie dyskretnego problemu plecakowego za pomocą programowania dynamicznego. Poznany algorytm zawsze zwraca optymalne rozwiązanie, ale nie działa w czasie wielomianowym. W tym rozdziale spróbujemy rozwiązywać problem plecakowy algorytmem zachłannym.

Możemy sobie wyobrazić wiele różnych algorytmów zachłannych rozwiązujących problem plecakowy. Przykładowo możemy w pierwszej kolejności wybierać najlżejsze przedmioty albo przedmioty o najwyższej wartości. My rozważymy algorytm, w którym w pierwszej kolejności wybieramy przedmioty o najkorzystniejszym stosunku wartości do wagi.

W algorytmie będziemy wykorzystywali dwa typy struktur: jedną do przechowywania danych wejściowych oraz drugą przechowującą stosunek wartości do wagi dla poszczególnych przedmiotów.

Listing 6.6. Zachłanne rozwiązanie problemu plecakowego – typy

```
struct Przedmiot{
 unsigned int cena, waga;
};

struct Stosunek{
  unsigned int przedmiot;
  double stos;
};
```

Będziemy potrzebowali także funkcji sortującej tablicę struktur `Stosunek` ze względu na wartość pola `stos`.

Listing 6.7. Zachłanne rozwiązanie problemu plecakowego – sortowanie

```
void Sort(Stosunek tab[], unsigned int n){
  unsigned int i,j,k;
  Stosunek pom;
  for(i=1;i<n;i++)
    if (tab[i].stos>tab[i-1].stos){
      pom=tab[i];
      for(j=i-1;(j>=0)&&(tab[j].stos<pom.stos);j--)
        tab[j+1]=tab[j];
      j++;
      tab[j]=pom;
    }
}
```

Powyższa funkcja to drobna modyfikacja algorytmu sortowania przez wstawianie z listingu 2.56. Modyfikacja polega na tym, że sortujemy tablicę

o elementach typu `Stosunek` biorąc pod uwagę wyłącznie wartość pola `stos` oraz że sortujemy tablicę malejąco a nie rosnąco.

Teraz zaimplementujemy funkcję `plecak`, która dla otrzymanej w argumentach tablicy przedmiotów, jej rozmiaru oraz ograniczeniu plecaka na wagę wypisuje na standardowym wyjściu sposób upakowania plecaka według algorytmu zachłannego.

Listing 6.8. Dyskretny problem plecakowy

```
  void plecak_dys(Przedmiot tab_przed[], unsigned int rozmiar,
2                                          unsigned int limit){

4   ...

6 }
```

Najpierw stworzymy tablicę, a następnie policzymy stosunki ceny do wagi dla wszystkich przedmiotów i uporządkujemy przedmioty ze względu na ten stosunek:

Listing 6.9. Dyskretny problem plecakowy

```
  void plecak_dys(Przedmiot tab_przed[], unsigned int rozmiar,
2                                          unsigned int limit){
    Stosunek tab_stos[rozmiar];
4   for(unsigned int i=0;i<rozmiar;i++){
      tab_stos[i].przedmiot = i;
6     tab_stos[i].stos = (double)tab_przed[i].cena
                                /tab_przed[i].waga;
8   }
    Sort(tab_stos,rozmiar);
10
    ...
12
  }
```

Teraz w pętli przejrzymy przedmioty posortowane malejąco względem stosunku ceny do wagi, dodając te przedmioty, które zmieszczą się do plecaka (po dodaniu każdego przedmiotu będziemy aktualizować dostępny limit wagi).

Listing 6.10. Dyskretny problem plecakowy

```
1 void plecak_dys(Przedmiot tab_przed[], unsigned int rozmiar,
                                         unsigned int limit){
3   Stosunek tab_stos[rozmiar];
    for(unsigned int i=0;i<rozmiar;i++){
5     tab_stos[i].przedmiot = i;
```

```
      tab_stos[i].stos = (double)tab_przed[i].cena
7                                /tab_przed[i].waga;
    }
9   Sort(tab_stos,rozmiar);
    for(unsigned int i = 0; (i<rozmiar)&&(limit>0);i++)
11    if (tab_przed[tab_stos[i].przedmiot].waga<limit){
        cout<<tab_stos[i].przedmiot<<endl;
13      limit = limit - tab_przed[tab_stos[i].przedmiot].waga;
      }
15
  }
```

Taki algorytm dla dyskretnego algorytmu plecakowego nie zawsze daje optymalne rozwiązanie:

Tabela 6.1. Przykładowe dane dla dyskretnego problemu plecakowego

numer przedmiotu	1	2	3	4
cena przedmiotu	80zł	60zł	30zł	30zł
waga przedmiotu	6kg	5kg	3kg	4kg

Przykład 6.4.
W tablicy 6.1 znajduje się przykład zbioru przedmiotów, dla którego przy ograniczeniu 8kg na wagę przedmiotów w plecaku, algorytm zachłanny da nieoptymalne rozwiązanie dyskretnego problemu plecakowego. Algorytm zachłanny włoży do plecaka tylko przedmiot pierwszy o wartości 80zł, zaś w optymalnym rozwiązaniu zapakowalibyśmy przedmioty 2 i 3 o łącznej wartości 90zł.

Teraz rozważymy ciągły problem plecakowy. Jest to modyfikacja problemu plecakowego, dla której algorytm zachłanny daje optymalne wyniki:

Problem 6.2.
Wejscie *Lista przedmiotów (dla każdego przedmiotu mamy określoną jego wagę jednostkową, cenę jednostkową oraz dostępną ilość), pojemność plecaka.*
Wyjście *Wybór przedmiotów mieszczących się w plecaku o maksymalnej możliwej wartości. Możemy brać ułamkowe ilości przedmiotów, ale nie możemy do plecaka zapakować większej ilości danego towaru niż jego dostępna ilość.*

Rozważymy teraz ciągły problem plecakowy dla przykładowych danych.

Przykład 6.5.
Załóżmy, że dane z Tabeli 6.1 dotyczą ciągłego problemu plecakowego, że ogra-

niczenie na pojemność plecaka to 8kg i że dostępna jest jedna sztuka każdego z przedmiotów wymienionych w tabeli.

Algorytm zachłanny w pierwszej kolejności wybierze przedmiot numer 1, gdyż ma on najkorzystniejszy stosunek masy do wagi. Pozostała w plecaku wolna przestrzeń, zostanie wykorzystana przez algorytm zachłanny do zapakowania 3kg drugiego przedmiotu (jak łatwo policzyć 3kg drugiego przedmiotu jest warte 36zł). W ten sposób algorytm zachłanny wygeneruje sposób zapakowania plecaka przedmiotami o sumarycznej wartości 116zł. To jest optymalne rozwiązanie dla rozważanego przypadku.

Ponieważ tym razem możemy brać różną liczbę poszczególnych przedmiotów, nasza funkcja będzie nieco bardziej skomplikowana. Pierwszą rzeczą, jaka się zmieni, będzie typ `Przedmiot`, który będzie zawierał teraz także dostępną ilość przedmiotu.

Listing 6.11. Ciągły problem plecakowy

```
  struct Przedmiot{
2  unsigned int cena, waga, ilosc;
  };
```

Początkowa część programu, w której porządkujemy przedmioty względem stosunku ceny do wagi, nie ulegnie zmianie w stosunku do rozwiązania dyskretnego problemu plecakowego.

Listing 6.12. Ciągły problem plecakowy

```
1  void plecak_dys(Przedmiot tab_przed[], unsigned int rozmiar,
                                              unsigned int limit){
3    Stosunek tab_stos[rozmiar];
     for(unsigned int i=0;i<rozmiar;i++){
5      tab_stos[i].przedmiot = i;
       tab_stos[i].stos = (double)tab_przed[i].cena
7                                  /tab_przed[i].waga;
     }
9    Sort(tab_stos,rozmiar);

11   ...

13 }
```

Tym razem wkładając jakiś przedmiot do plecaka będziemy wkładali jego maksymalną możliwą ilość.

Listing 6.13. Ciągły problem plecakowy

```
void plecak_dys(Przedmiot tab_przed[], unsigned int rozmiar,
                                        unsigned int limit){
  Stosunek tab_stos[rozmiar];
  for(unsigned int i=0;i<rozmiar;i++){
    tab_stos[i].przedmiot = i;
    tab_stos[i].stos = (double)tab_przed[i].cena
                              /tab_przed[i].waga;
  }
  Sort(tab_stos,rozmiar);
  double pom;
  unsigned int ind;
  for(unsigned int i = 0; (i<rozmiar)&&(limit>0);i++){
    ind = tab_stos[i].przedmiot;
    if (tab_przed[ind].waga*tab_przed[ind].ilosc<limit){
      cout<<tab_przed[tind].ilosc<<"*"<<ind<<endl;
      limit=limit-tab_przed[ind].ilosc*tab_przed[ind].waga;
    }
    else {
      cout<<limit/tab_pred[ind].waga<<"*"<<ind<<endl;
      limit = 0;
    }
  }
}
```

Zadanie 25. Napisz funkcję, która otrzymuje jako argumenty tablicę obustronnie otwartych odcinków (każdy odcinek reprezentowany jest przez parę liczb oznaczających początek i koniec odcinka) oraz rozmiar tablicy i generuje największy pod względem liczności podzbiór otrzymanych odcinków taki, że żadne dwa odcinki na siebie nie nachodzą.

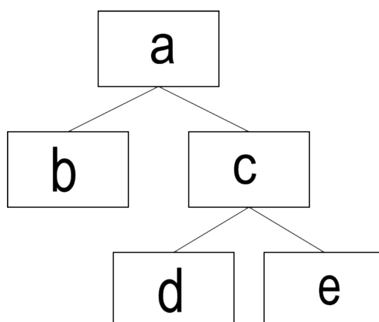
Rozdział 7

Drzewa

7.1. Definicja drzew i sposoby ich reprezentacji **132**
7.1.1. Uogólniona lista wskaźnikowa 133
7.1.2. „Lewy syn, prawy brat” 134
7.2. Przechodzenie drzew **135**
7.2.1. Preorder 135
7.2.2. Postorder 136
7.2.3. Inorder 137
7.3. Kopce, sortowanie przez kopcowanie **138**
7.3.1. Kopce zupełne, sposób ich reprezentacji 138
7.3.2. Implementacja kolejki priorytetowej przy pomocy kopca 140
7.3.3. Sortowanie przez kopcowanie 147

7.1. Definicja drzew i sposoby ich reprezentacji

Drzewo to struktura danych składająca się z wierzchołków. Jeden wierzchołek w drzewie jest wyróżniony i nazywany jest korzeniem. Każdy wierzchołek poza korzeniem jest połączony krawędzią z dokładnie jednym wierzchołkiem nazywanym ojcem. Korzeń nie posiada ojca. Pojedynczy wierzchołek może być ojcem dla wielu innych wierzchołków. Poza pojęciem ojca w odniesieniu do drzew używamy także pozostałej rodzinnej terminologii, mówimy między innymi o: synach, braciach, potomkach, przodkach itd. Wierzchołki, które nie mają dzieci nazywamy liśćmi, wierzchołki, które mają dzieci nazywamy wewnętrznymi. Drzewo, w którym każdy wierzchołek, ma co najwyżej k synów nazywamy k-arnym. Drzewo 2-arne nazywamy drzewem binarnym.



Rysunek 7.1. Przykładowe drzewo

Drzewa przyjęło się rysować w taki sposób, że korzeń znajduje się na samej górze, pod nim w jednym rzędzie jego dzieci, w kolejnej warstwie wnuki itd.. Przykładową graficzną reprezentację drzewa przedstawiono na rysunku 7.1. Drzewo przedstawione na tym rysunku jest drzewem binarnym.

Zazwyczaj przyjmujemy, że dzieci każdego wierzchołka w drzewie są uporządkowane liniowo. W nawiązaniu do graficznej reprezentacji drzewa mówi się, że jakiś wierzchołek leży na lewo lub na prawo od swojego brata. W przypadku drzew binarnym mówimy o lewym i prawym synu, i zazwyczaj rozróżniamy sytuacje, gdy wierzchołek ma tylko lewego lub tylko prawego syna.

W znajdującym się na rysunku 7.1 drzewie
— wierzchołek `a` jest korzeniem,
— wierzchołki `b`, `d` i `e` są liśćmi,
— wierzchołki `a` i `c` są wierzchołkami wewnętrznymi,
— wierzchołki `d` i `e` są dziećmi wierzchołka `c` (wierzchołek `d` jest lewym, zaś wierzchołek `e` prawym synem),
— wierzchołki `b` i `c` są braćmi.

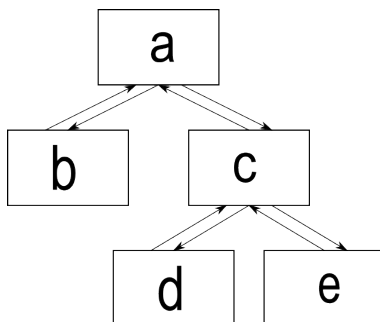
7.1.1. Uogólniona lista wskaźnikowa

Najpopularniejszym sposobem reprezentacji drzew jest swego rodzaju uogólnienie dwukierunkowej listy wskaźnikowej. Uogólnienie, w którym wszystkie wierzchołki poza pierwszym (korzeniem) mają poprzednik (ojca) i w którym wierzchołki mogą mieć więcej niż jeden następnik (syna). Podobnie jak to miało miejsce w przypadku listy wskaźnikowej brak syna lub ojca będziemy zaznaczać nadając wartość `NULL` odpowiedniemu wskaźnikowi. W przypadku drzew binarnych typ wierzchołka jest następujący:

Listing 7.1. Typ do przechowywania wierzchołka drzewa binarnego

```
1 struct wierzcholek{
    Element el;
3   wierzcholek *ojciec, *l_syn, *p_syn;
  };
```

Jeżeli drzewo miałoby większą arność, struktura miałaby więcej wskaźników na dzieci (przy dużej arności wskaźniki do dzieci mogłyby być umieszczone w tablicy). Na rysunku 7.2 znajduje się ilustracja reprezentacji drzewa binarnego przez strukturę taką, jak w listingu 7.1 (strzałka z wierzchołka a do wierzchołka b oznacza, że w wierzchołku a przechowujemy wskaźnik na b itd.). Tego typu reprezentację drzew możemy stosować nie tylko dla drzew



Rysunek 7.2. Reprezentacja przy pomocy uogólnionej listy wskaźnikowej

o ograniczonej arności, jednak przy drzewach bez takiego ograniczenia lista dzieci musi być potencjalnie nieograniczona. Można to zaimplementować choćby przy pomocy tablic dynamicznych. Poza tablicą dzieci będziemy potrzebowali wtedy także pola przechowującego liczbę dzieci danego wierzchołka.

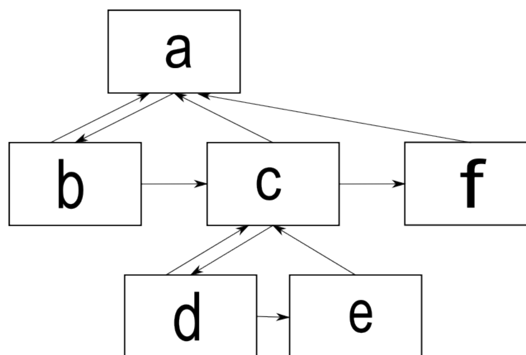
Listing 7.2. Typ do przechowywania wierzchołka drzewa o dowolnej arności

```
struct wierzcholek{
  Element el;
  wierzcholek *ojciec, *tab_dzieci;
  unsigned int liczba_dzieci;
};
```

Tak jak w przypadku list wskaźnikowych dostęp do listy uzyskiwaliśmy poprzez wskaźnik do pierwszego elementu listy, tak w przypadku powyższej reprezentacji drzewa dostęp do drzewa uzyskujemy poprzez wskaźnik do korzenia drzewa. Stąd przechowując drzewo pamiętamy wskaźnik na korzeń.

7.1.2. „Lewy syn, prawy brat”

Innym sposobem reprezentacji drzew o dowolnej arności jest reprezentacja „lewy syn, prawy brat” polegająca na tym, że każdy wierzchołek poza wskaźnikiem na swojego ojca pamięta wskaźnik na skrajnie lewego syna i brata z prawej. Takie informacje wystarczą do tego, żeby zapewnić sobie dostęp do dowolnego wierzchołka w drzewie. Na rysunku 7.3 mamy schemat połączeń pomiędzy wierzchołkami w tej reprezentacji drzewa. Podobnie jak poprzednio także przy reprezentacji „lewy syn, prawy brat”, jeżeli któryś z wierzchołków nie ma ojca, syna czy prawego brata, to odpowiedni wskaźnik otrzymuje wartość `NULL`.



Rysunek 7.3. Reprezentacja „lewy syn, prawy brat”

Podstawową zaletą omawianej implementacji jest identyczny sposób przechowywania drzew o dowolnej arności. Do reprezentowania wierzchołka dowolnego drzewa wystarczy nam struktura przechowująca tylko trzy wskaźniki.

Listing 7.3. Typ wierzchołka w reprezentacji „lewy syn, prawy brat”

```
1 struct wierzcholek_lspb{
    Element el;
3   wierzcholek *ojciec, * lewy_syn, * prawy_brat;
  };
```

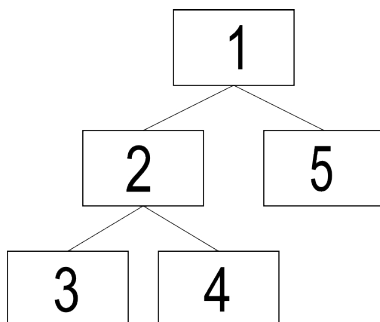
Podobnie jak w poprzednio omawianej reprezentacji drzewa, także w reprezentacji „lewy syn, prawy brat”, aby mieć dostęp do drzewa, przechowujemy wskaźnik na jego korzeń.

7.2. Przechodzenie drzew

W wielu algorytmach na drzewach występuje potrzeba odwiedzenia („zrobienia czegoś z”) wszystkich wierzchołków drzewa. Często też nie jest obojętne, w jakiej kolejności będziemy odwiedzali wierzchołki drzewa. Poniżej prezentujemy trzy najpopularniejsze porządki odwiedzania wierzchołków danego drzewa.

7.2.1. Preorder

Preorder to porządek, w którym zanim odwiedzimy jakiś wierzchołek, to wcześniej musimy odwiedzić wszystkich jego przodków. Na rysunku numery wierzchołków oznaczają kolejność ich odwiedzania w porządku preorder:



Rysunek 7.4. Przechodzenie drzew w kolejności preorder

Poniższa funkcja otrzymuje jako argument drzewo binarne (czyli wskaźnik do korzenia drzewa) przechowywane w strukturach typu `wierzcholek` z listingu 7.1 i odwiedza wszystkie wierzchołki tego drzew, tj. uruchamia na nich funkcję `odwiedz`, w kolejności preorder. Z definicji kolejności preorder najpierw odwiedzamy korzeń drzewa/poddrzewa, a następnie rekurencyjnie wszystkich jego potomków.

Listing 7.4. Preorder w drzewie binarnym

```
  void peorder(struct wierzcholek * korzen){
2    odwiedz(korzen)
     if (korzen->l_syn!=NULL)
4      prorder(korzen->l_syn);
      if (korzen->r_syn!=NULL)
6      prorder(korzen->r_syn);
  }
```

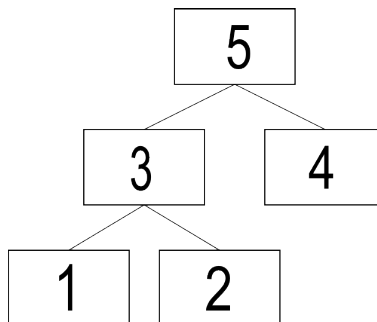
Otrzymując drzewo w reprezentacji „lewy syn, prawy brat” postępujemy podobnie, z tym że po odwiedzeniu wszystkich potomków musimy jeszcze zadbać o odwiedzenie braci i potomków braci rozpatrywanego wierzchołka.

Listing 7.5. Preorder w reprezentacji „lewy syn, prawy brat”

```
1 void peorder(struct wierzcholek_lspb * korzen){
    odwiedz(korzen);
3   if (korzen->lewy_syn!=NULL)
      prorder(korzen->lewy_syn);
5    if (korzen->prawy_brat!=NULL)
      prorder(korzen->prawy_brat);
7 }
```

7.2.2. Postorder

Drugi z omawianych porządków odwiedzania drzewa jest w pewnym sensie odwrotnością kolejności preorder, choć porównując rysunki 7.4 i 7.5 widzimy, że nie w sensie odwrotnej kolejności odwiedzania wierzchołków. W kolejności postorder, zanim odwiedzimy jakiś wierzchołek, musimy odwiedzić wszystkich jego potomków. Na rysunku 7.5 numery wierzchołków odpowiadają kolejności ich występowania w porządku postorder.



Rysunek 7.5. Przechodzenie drzew w kolejności postorder

Implementacja przechodzenia drzewa binarnego w kolejności postorder jest bardzo podobna do implementacji przechodzenia drzewa w kolejności preorder. Jedyną różnicą jest to, że teraz funkcję `odwiedz` wywołujemy na końcu funkcji (czyli po odwiedzeniu potomków danego wierzchołka).

Listing 7.6. Postorder w drzewie binarnym

```
1 void postorder(struct wierzcholek * korzen){
    if (korzen->l_syn!=NULL)
3     prorder(korzen->l_syn);
    if (korzen->r_syn!=NULL)
5     prorder(korzen->r_syn);
    odwiedz(korzen);
7 }
```

Implementacja przechodzenia drzewa w reprezentacji „lewy syn, prawy brat” w kolejności postorder również niewiele się różni od analogicznej implementacji przechodzenia w kolejności preorder.

Listing 7.7. Postorder w reprezentacji „lewy syn, prawy brat”

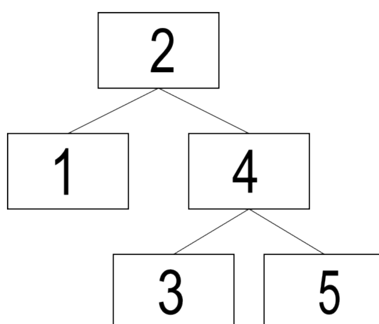
```
1 void postorder(struct wierzcholek_lspb * korzen){
    if (korzen->lewy_syn!=NULL)
3     prorder(korzen->lewy_syn);
     odwiedz(korzen);
5   if (korzen->prawy_brat!=NULL)
      prorder(korzen->prawy_brat);
7 }
```

Zwróćmy uwagę, że nie przenieśliśmy wywołania funkcji `odwiedz` na koniec funkcji `postorder`. Powodem tego jest specyfika tego sposobu reprezentacji drzew. Funkcję `odwiedz` wywołujemy po odwiedzeniu potomków danego wierzchołka, ale przed odwiedzeniem jego rodzeństwa z prawej (w przypadku reprezentacji drzewa przy pomocy uogólnionych list wskaźnikowych, funkcję `postorder` dla całego rodzeństwa wywołujemy z poziomu ich ojca, stąd wywołanie funkcji `odwiedz` jest wtedy na końcu funkcji `postorder`).

7.2.3. Inorder

Porządek inorder definiujemy tylko dla drzew binarnych (niektórzy autorzy uogólniają go na drzewa o dowolnej arności, ale taki uogólniony porządek jest nienaturalny i nie będziemy go rozważać). W porządku tym dowolny wierzchołek odwiedzamy po odwiedzeniu wszystkich wierzchołków poddrzewa o korzeniu w lewym synu rozważanego wierzchołka, ale przed odwiedzeniem wierzchołków drzewa o korzeniu w jego prawym synu. Numery

wierzchołków w drzewie na rysunku 7.6 odpowiadają kolejności ich odwiedzania w porządku inorder.



Rysunek 7.6. Przechodzenie drzew w kolejności inorder

Implementacja przechodzenia drzewa binarnego w kolejności `inorder` jest analogiczna do implementacji przechodzenia drzew binarnych w kolejności preorder i postorder:

Listing 7.8. Inorder w drzewie binarnym

```
1 void postorder(struct wierzcholek * korzen){
    if (korzen->l_syn!=NULL)
3     prorder(korzen->l_syn);
    odwiedz(korzen);
5    if (korzen->r_syn!=NULL)
      prorder(korzen->r_syn);
7 }
```

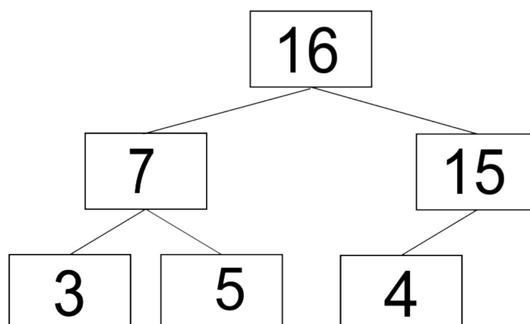
Nie będziemy rozważali implementacji przechodzenia inorder drzew w reprezentacji „lewy syn, prawy brat”, gdyż jest to reprezentacja przeznaczona do przechowywania drzew o dowolnej arności, a porządek inorder definiujemy wyłącznie dla drzew binarnych. Ponadto w tej reprezentacji nie istnieje pojęcia lewego czy prawego syna (dzieci są uporządkowane od lewej do prawej, ale w przypadku gdy wierzchołek ma jednego syna, to nie ma możliwości oznaczenia go jako lewego lub prawego).

7.3. Kopce, sortowanie przez kopcowanie

7.3.1. Kopce zupełne, sposób ich reprezentacji

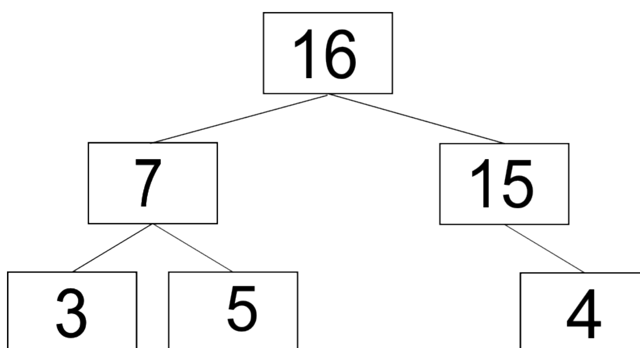
Na koniec rozdziału o drzewach poznamy jedno z ich ważnych zastosowań. Kopcem nazywamy drzewo, w którym wartość przechowywana w ojcu

jest większa od wartości przechowywanych w jego synach. Na rysunku 7.7 przedstawiono przykładowy kopiec.



Rysunek 7.7. Kopiec

Drzewo k-arne nazywamy zupełnym, jeżeli wszystkie jego poziomy, poza ewentualnie ostatnim, są w pełni wypełnione, zaś ostatni poziom jest spójnie wypełniony od lewej. Kopiec jest zupełny, jeżeli jest drzewem zupełnym. Kopiec z rysunku 7.7 jest przykładem zupełnego kopca binarnego, gdyż ma w pełni wypełnione pierwsze dwa poziomy (ma na nich maksymalną liczbę wierzchołków, jaką na danym poziomie może mieć drzewo binarne), zaś ostatni poziom (trzeci) ma wypełniony spójnie od lewej (wszystkie wierzchołki na tym poziomie zostały wstawione „bez dziur" od lewej strony).



Rysunek 7.8. Przykłada kopca binarnego, który nie jest zupełny

Rysunek 7.8 przedstawia przykład kopca, który nie jest zupełny. Co prawda wszystkie poziomy poza ostatnim są wypełnione, ale na ostatnim poziomie wierzchołki nie są ułożone spójnie od lewej (pomiędzy wierzchołkami przechowującymi piątkę i czwórkę jest puste miejsce na lewego syna wierzchołka przechowującego wartość 15).

Drzewa zupełne, a więc także kopce zupełne, możemy przechowywać w tablicy. W komórce o indeksie `0` znajduje się wtedy korzeń, a kolejne wierzchołki umieszczamy w tablicy warstwami. W tabeli 7.1 znajduje się tablica zawierająca zapisany w ten sposób kopiec z rysunku 7.7 (w pierwszym wierszu są indeksy komórek tablicy, a w drugim zawartość komórek tablicy).

Tabela 7.1. Tablicowa reprezentacja kopca z Rysunku 7.7

indeksy komórek	0	1	2	3	4	5
wartości komórek	16	7	15	3	5	4

Łatwo policzyć, że w przedstawionej reprezentacji k-arnego drzewa zupełnego ojciec wierzchołka o indeksie `i` ma indeks $(i-1)/k$, zaś jego synowie $k \cdot i+1$, $k \cdot i+2$, $\ldots$, $k \cdot i+k$.

7.3.2. Implementacja kolejki priorytetowej przy pomocy kopca

Kopce są jednym z najpopularniejszych sposobów implementacji kolejki priorytetowej. Poniżej znajduje się nagłówek klasy, która będzie implementowała kolejkę priorytetową za pomocą zupełnych kopców binarnych:

Listing 7.9. Nagłówek klasy PriorityQueue

```
 1 class PriorityQueue {
   private:
 3   struct Element_kol{
       Element el;
 5     unsigned int prior;
       Element_kol(Element e, unsigned int p):el(e),
 7                                        priorytet(p){}
     };
 9   Element_kol *kopiec;
     unsigned int rozmiar_tab, liczba_el;
11 public:
     PriorityQueue();
13   ~PriorityQueue();
     bool empty();
15   void push(Element e, unsigned int p);
     Element top();
17   void pop();
   };
```

Tablica `kopiec` będzie przechowywała elementy kopca, zmienna `rozmiar_tab` bieżący rozmiar tablicy, zaś zmienna `liczba_el` liczbę elementów kopca.

Konstruktor klasy musi zaalokować tablicę `kopiec` oraz zainicjować zmienne `rozmiar_tab` i `liczba_el`. Początkowy rozmiar tablicy nie ma wpły-

wu na asymptotyczną złożoność poszczególnych operacji (chyba, że znamy ograniczenie na maksymalny rozmiar kopca). Destruktor musi tylko zwolnić pamięć po tablicy `kopiec`.

Listing 7.10. Konstruktor i destruktor klasy PriorityQueue

```
PriorityQueue::PriorityQueue(){
  kopiec = new Element_kol[100];
  rozmiar_tab = 100;
  liczba_el = 0;
}

PriorityQueue::~PriorityQueue(){
  delete [] kopiec;
}
```

Jak zwykle najprostsza do zaimplementowania jest funkcja `empty`

Listing 7.11. Operacja empty

```
bool PriorityQueue::empty(){
  return (liczba_el==0);
}
```

Równie prosta jest w tej implementacji funkcja `top`, gdyż element o największym priorytecie będzie zawsze w korzeniu kopca, a więc w elemencie tablicy `kopiec` o indeksie `0`.

Listing 7.12. Operacja top

```
Element PriorityQueue::top(){
  return kopiec[0].el;
}
```

Usuwając element kolejki o najwyższym priorytecie będziemy usuwali korzeń kopca. Musimy to zrobić w taki sposób, aby otrzymane po przekształceniu drzewo było nadal kopcem zupełnym. Aby to uzyskać na miejsce usuniętego korzenia kopca przeniesiemy wierzchołek o największym indeksie w tablicy. Dzięki temu otrzymane drzewo będzie drzewem zupełnym, w którym, w co najwyżej w jednym miejscu (w nowym korzeniu), naruszony został porządek kopca. Jeżeli nowy korzeń ma priorytet mniejszy od któregoś ze swoich synów, to zamieniamy go miejscami z tym spośród synów, który ma większy priorytet (w ten sposób korzeniem znowu jest wierzchołek o największym priorytecie). Przeniesiony w dół kopca wierzchołek nadal może mieć mniejszy priorytet niż któryś z jego synów. W takim przypadku ponownie zamieniamy go z synem o większym priorytecie. Czynimy tak do-

póki przesuwany w dół wierzchołek nie będzie liściem albo nie będzie miał priorytetu większego od obu swoich synów. Takie przesuwanie wierzchołka w dół kopca w celu przywrócenia porządku kopca nazywamy *przesiewaniem w dół*. Poniżej algorytm usuwania korzenia kopca opisany w punktach:

Algorytm 7.1.

1. *Nadpisz korzeń elementem* `w`*, który znajduje się w komórce tablicy o największym indeksie (największym indeksie spośród zajętych komórek tablicy).*
2. *Dopóki* `w` *nie jest liściem lub ma mniejszy priorytet od któregoś ze swoich synów powtarzaj krok* 3*:*
3. *Zamień miejscami wierzchołek* `w` *z tym z jego synów, który ma większy priorytet.*

Najprostszy do zaimplementowania jest pierwszy krok algorytmu 7.1

Listing 7.13. Operacja pop

```
1 void PriorityQueue::pop(){
    liczba_el--;
3   kopiec[0] = kopiec[liczba_el];

5   ...

7 }
```

W pętli z punktu drugiego algorytmu 7.1 będziemy potrzebowali pomocniczej zmiennej do zamieniania miejscami elementów tablicy oraz zmiennej przechowującej aktualny indeks przesiewanego wierzchołka (jego początkowa wartość to `0`). Pętla będzie wykonywana tak długo, jak długo przesiewany wierzchołek będzie miał chociaż jednego syna o priorytecie większym od niego.

Listing 7.14. Operacja pop

```
1 void PriorityQueue::pop(){
    liczba_el--;
3   kopiec[0] = kopiec[liczba_el];
    Element_kol pom;
5   unsigned int i = 0;
    while(((2*i+1<liczba_el)&&
7          (kopiec[i].prior<kopiec[2*i+1].prior))||
           ((2*i+2<liczba_el)&&
9          (kopiec[i].prior<kopiec[2*i+2].prior))){
      ...
11  }
  }
```

Wewnątrz pętli `while` należy znaleźć syna o wyższym priorytecie (pamiętając przy tym, że przesiewany wierzchołek może mieć tylko jednego syna).

Listing 7.15. Operacja pop

```
void PriorityQueue::pop(){
  liczba_el--;
  kopiec[0] = kopiec[liczba_el];
  Element_kol pom;
  unsigned int i = 0;
  while(((2*i+1<liczba_el)&&
        (kopiec[i].prior<kopiec[2*i+1].prior))||
        ((2*i+2<liczba_el)&&
        (kopiec[i].prior<kopiec[2*i+2].prior))){
    unsigned int syn = 2*i+1;
    if ((syn+1<liczba_el)&&
        (kopiec[syn].prior<kopiec[syn+1].prior))
      syn++;

    ...

  }
}
```

Na koniec musimy jeszcze zamienić przesiewany wierzchołek miejscami z synem o większym priorytecie (jest pod indeksem `syn`)

Listing 7.16. Operacja pop

```
void PriorityQueue::pop(){
  liczba_el--;
  kopiec[0] = kopiec[liczba_el];
  Element_kol pom;
  unsigned int i = 0;
  while(((2*i+1<liczba_el)&&
        (kopiec[i].prior<kopiec[2*i+1].prior))||
        ((2*i+2<liczba_el)&&
        (kopiec[i].prior<kopiec[2*i+2].prior))){
    unsigned int syn = 2*i+1;
    if ((syn+1<liczba_el)&&
        (kopiec[syn].prior<kopiec[syn+1].prior))
      syn++;
    pom = kopiec[syn];
    kopiec[syn] = kopiec[i];
    kopiec[i] = pom;
    i = syn;
  }
}
```

Można oszacować, że liczba warstw wierzchołków zupełnego drzewa binarnego zawierającego n wierzchołków jest rzędu $\log_2 n$, co implikuje, że pętla `while` w powyższym programie wykona $O(\log_2 n)$ obrotów. Ponieważ w każdym obrocie wykonujemy ograniczoną przez stałą liczbę operacji, to złożoność całej funkcji to $O(\log n)$, gdzie n jest liczbą elementów przechowywanych w kopcu.

Na koniec przedstawimy operację `push`. W jej przypadku nowy element wstawimy do komórki tablicy o indeksie `liczba_el`. Ta komórka odpowiada pierwszemu wolnemu miejscu w ostatniej warstwie wierzchołków drzewa lub jeżeli ostatnia warstwa jest już zapełniona, pierwszemu wierzchołkowi w nowej warstwie. Po wstawieniu nowy wierzchołek *przesiewamy w górę*, czyli zmieniamy z ojcem tak długo, dopóki przesiewany wierzchołek nie stanie się korzeniem albo nie będzie miał mniejszego priorytetu od swojego ojca. Algorytm wstawiania możemy zapisać w następujący sposób:

Algorytm 7.2.

1. *Dopisz wierzchołek* `w` *na koniec kopca.*
2. *Dopóki* `w` *nie jest korzeniem lub ma większy priorytet od swojego ojca, powtarzaj krok* 3.
3. *Zamień miejscami wierzchołek* `w` *z jego ojcem.*

W przypadku operacji `push` musimy uwzględnić przypadek, gdy w tablicy nie ma już miejsca.

Listing 7.17. Operacja push

```
1 void PriorityQueue::push(Element e, unsigned int p)
    if (liczba_el == rozmiar_tab)
3
    ...
5
  }
```

W takiej sytuacji tworzymy nową, dwa razy większą tablicę, przepisujemy do niej zawartość starej, zaś starą tablicę usuwamy z pamięci.

Listing 7.18. Operacja push

```
  void PriorityQueue::push(Element e, unsigned int p)
2   if (liczba_el == rozmiar_tab){
      Element_kol * nowa_tab = new Element_kol[rozmiar_tab*2];
4
      for(int i=0;i<rozmiar_tab;i++)
6       nowa_tab[i] = kopiec[i];

8     delete [] kopiec;
```

```
      kopiec = nowa_tab;
10    rozmiar_tab*=2;
    }
12  ...

14 }
```

Po zapewnieniu sobie posiadania odpowiedniej ilości miejsca w tablicy wstawiamy na jej koniec nowy element.

Listing 7.19. Operacja push

```
  void PriorityQueue::push(Element e, unsigned int p)
2   if (liczba_el == rozmiar_tab){
      Element_kol * nowa_tab = new Element_kol[rozmiar_tab*2];
4
      for(int i=0;i<rozmiar_tab;i++)
6       nowa_tab[i] = kopiec[i];

8     delete [] kopiec;
      kopiec = nowa_tab;
10  }

12  kopiec[liczba_el].el = e;
    kopiec[liczba.el].prior = p;
14  liczba_el++;

16  ...

18 }
```

Następnie będziemy przesiewali wstawiony element w górę. Wszystko będzie się odbywało w pętli, która będzie się kręciła tak długo, jak długo przesiewany wierzchołek będzie miał ojca i ten ojciec będzie miał mniejszy od niego priorytet.

Listing 7.20. Operacja push

```
  void PriorityQueue::push(Element e, unsigned int p)
2   if (liczba_el == rozmiar_tab){
      Element_kol * nowa_tab = new Element_kol[rozmiar_tab*2];
4
      for(int i=0;i<rozmiar_tab;i++)
6       nowa_tab[i] = kopiec[i];

8     delete [] kopiec;
      kopiec = nowa_tab;
10  }
```

```
12    kopiec[liczba_el].el = e;
      kopiec[liczba.el].prior = p;
14    liczba_el++;
      Element_kol pom;
16    unsigned int i = liczba_el-1;
      while ((i>0)&&(kopiec[i].prior>kopiec[(i-1)/2].prior)){
18
        ...
20
      }
22 }
```

Wewnątrz pętli `while` zamieniamy tylko miejscami przesiewany wierzchołek z jego ojcem.

Listing 7.21. Operacja push

```
   void PriorityQueue::push(Element e, unsigned int p)
2    if (liczba_el == rozmiar_tab){
       Element_kol * nowa_tab = new Element_kol[rozmiar_tab*2];
4      for(int i=0;i<rozmiar_tab;i++)
         nowa_tab[i] = kopiec[i];
6      delete [] kopiec;
       kopiec = nowa_tab;
8    }
     kopiec[liczba_el].el = e;
10   kopiec[liczba.el].prior = p;
     liczba_el++;
12   Element_kol pom;
     unsigned int i = liczba_el-1;
14   while (i>0)&&(kopiec[i].prior>kopiec[(i-1)/2].prior){
       pom=kopiec[(i-1)/2];
16     kopiec[(i-1)/2]=kopiec[i];
       kopiec[i]=pom;
18     i=(i-1)/2;
     }
20 }
```

Łatwo sprawdzić, że złożoność czasowa powyższej implementacji operacji `push` w przypadku, gdy nie musimy powiększać tablicy, to $O(\log n)$ dla kolejki przechowującej n elementów. Jeżeli musimy powiększyć tablicę, to złożoność operacji `push` rośnie do $O(n)$. Proste rachunki, analogiczne do tych, które przeprowadziliśmy w przypadku operacji `push_back` dla tablicowej implementacji listy, pokazują, że zamortyzowana złożoność operacji `push` wynosi $O(\log n)$.

7.3.3. Sortowanie przez kopcowanie

W rozdziale 2.4.3 zauważyliśmy, że dowolnej kolejki priorytetowej możemy użyć do sortowania listy. Robimy to w ten sposób, że umieszczamy wszystkie elementy listy w kolejce priorytetowej operacją `push` z priorytetami równymi wartościom elementów, a następnie wyjmujemy elementy z kolejki priorytetowej i umieszczamy je od końca w tablicy. W efekcie tablica, do której przeniesiemy elementy sortowanej listy z kolejki priorytetowej będzie posortowana rosnąco. Gdy taki algorytm zastosujemy używając kolejki priorytetowej zaimplementowanej przy pomocy kopca, dostaniemy algorytm sortujący o złożoności $O(n \cdot \log n)$. Taką złożoność uzyskujemy analizując złożoności i liczbę wykonań poszczególnych operacji. Dla listy o rozmiarze n taki algorytm najpierw tworzy kopiec wykonując n operacji `push` o złożoności $O(\log n)$, co w sumie daje złożoność $O(n \cdot \log n)$ tworzenia kopca, a następnie przenosi elementy kolejki do tablicy wykonując n razy operację `top` o złożoności $O(1)$ i operację `pop` o złożoności $O(\log n)$, co daje złożoność $O(n \cdot \log n)$ drugiego etapu algorytmu.

Teraz przedstawimy algorytm sortujący zapisaną w tablicy listę w miejscu (czyli bez przenoszenia elementów tablicy do tymczasowych struktur) przy pomocy kopców. Taki algorytm nosi nazwę sortowania przez kopcowanie (ang. *heapsort*).

W algorytmie wykorzystamy pomocniczą funkcję, przesiewającą elementy kopca w dół. Funkcja otrzyma jako argumenty kopiec (tablicę, w której priorytety poszczególnych elementów są równe ich wartościom), rozmiar tablicy oraz indeks elementu, który mamy przesiać w dół

Listing 7.22. Sortowanie przez kopcowanie – przesiewanie

```
void Przesiej_w_dol(Element kopiec, unsigned int rozmiar,
                                       unsigned int indeks){
  element pom;
  while(((2*indeks+1<rozmiar)&&
        (kopiec[i]<kopiec[2*indeks+1]))||
        ((2*indeks+2<rozmiar)&&
        (kopiec[indeks]<kopiec[2*indeks+2]))){
    unsigned int syn = 2*indeks+1;
    if ((syn+1<rozmiar)&&
        (kopiec[syn]<kopiec[syn+1]))
      syn++;
    pom = kopiec[syn];
    kopiec[syn] = kopiec[indeks];
    kopiec[indeks] = pom;
    i = syn;
  }
}
```

Teraz napiszemy funkcję sortującą. Funkcja otrzyma jako argumenty tablicę oraz jej rozmiar i posortuje otrzymaną w argumencie tablicę.

Listing 7.23. Sortowanie przez kopcowanie

```
1 void HeapSort(Elementy tablica[], unsigned int rozmiar){

3   ...

5 }
```

Okazuje się, że mając nieposortowaną tablicę o rozmiarze n, możemy zbudować z jej elementów kopiec w czasie $O(n)$, czyli szybciej niż przez użycie n wywołań operacji `push`, do czego potrzebujemy $O(n \cdot \log n)$ porównań. Wystarczy po kolei przesiać w dół elementy tablicy od elementu o indeksie $(n-2)/2$ do elementu o indeksie 0 (elementy z drugiej połowy tablicy, w tablicowej reprezentacji drzewa, są liśćmi, więc nie da się ich przesiać w dół). Szacując złożoność takiego algorytmu trzeba zauważyć, że połowa elementów nie będzie przesiewana, jedna czwarta elementów będzie przesiana o jeden poziom, jedna ósma elementów będzie przesiana o dwa poziomy itd. Otrzymujemy więc, że złożoność algorytmu jest ograniczona przez sumę następującego szeregu:

$$\frac{1}{4} \cdot n \cdot 1 + \frac{1}{8} \cdot n \cdot 2 + \ldots$$

a to jest $O(n)$.

W funkcji sortującej wykorzystamy przedstawiony powyższy algorytm budowania kopca.

Listing 7.24. Sortowanie przez kopcowanie

```
1 void HeapSort(Elementy tablica[], unsigned int rozmiar){
    for(unsigned int i = (rozmiar-2)/2;i>=0;i--)
3     Przesiej_w_dol(tablica, rozmiar, i);

5   ...

7 }
```

Na koniec funkcji usuniemy `rozmiar-1` razy korzeń z kopca. Usuwany element zapiszemy w komórce zwalnianej w tym samym czasie przez kopiec. W ten sposób, kolejne, coraz mniejsze, elementy znajdujące się w korzeniu kopca, będziemy zapisywali po kolei od końca tablicy i w efekcie otrzymamy posortowaną tablicę.

Listing 7.25. Sortowanie przez kopcowanie

```
1 void HeapSort(Elementy tablica[], unsigned int rozmiar){
    for(unsigned int i = (rozmiar-2)/2;i>=0;i--)
3     Przesiej_w_dol(tablica, rozmiar, i);
    Element pom;
5   for(unsigned int i = rozmiar-1; i>1;i--){
     pom = tablica[0];
7     tablica[0] = tablica[i];
      tablica[i] = pom;
9     Przesiej_w_dol(tablica, i,0);
    }
11 }
```

Rozdział 8

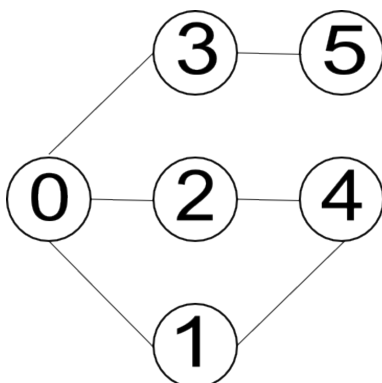
Grafy

8.1. Definicja i sposoby reprezentacji grafu **152**
8.1.1. Listy sąsiedztwa 153
8.1.2. Macierz sąsiedztwa 154
8.1.3. Porównanie 155
8.2. Przechodzenie grafów **156**
8.2.1. Przechodzenie grafu w głąb 156
8.2.2. Przechodzenie grafu wszerz 159

8.1. Definicja i sposoby reprezentacji grafu

Grafy to jeden z najważniejszych abstrakcyjnych typów danych. W niniejszym rozdziale poznamy jedynie podstawowe informacje na ich temat: sposoby przechowywania grafów w pamięci komputera i sposoby ich przechodzenia. Istnieje wiele wartych poznania algorytmów operujących na grafach. Zainteresowany nimi czytelnik może przykładowo sięgnąć po [2] lub [7].

Grafem lub grafem nieskierowanym nazywamy w matematyce parę $G = (V, E)$, w której V to zbiór obiektów nazywanych wierzchołkami, zaś E to zbiór dwuelementowych podzbiorów V nazywanych krawędziami. Graf przedstawiamy graficznie jako zbiór wierzchołków połączonych krawędziami (krawędź $\{a, b\}$ łączy wierzchołki a i b). Wierzchołki połączone krawędziami nazywamy sąsiednimi. Przykładowy graf możemy zobaczyć na rysunku 8.1.



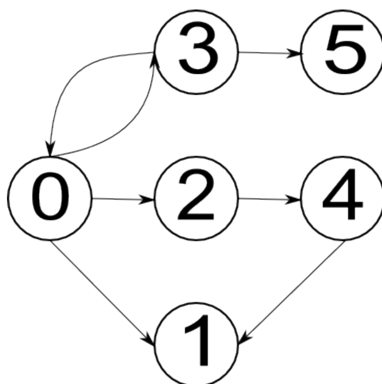
Rysunek 8.1. Przykładowy graf

W grafie z rysunku 8.1 zbiór wierzchołków $V = \{0, 1, 2, 3, 4, 5\}$, zaś zbiór krawędzi $E = \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 4\}, \{2, 4\}, \{3, 5\}\}$. Przy pomocy grafów możemy modelować wiele rzeczywistych obiektów i zjawisk. Jednym z najpopularniejszych szkolnych przykładów jest spojrzenia na grafy jako na modele sieci dróg, w których wierzchołki odpowiadają skrzyżowaniom, zaś krawędzie drogom pomiędzy nimi.

W grafie zdefiniowanym jak powyżej krawędzie nie mają kierunku ($\{0, 1\}$ i $\{1, 0\}$ to ten sam zbiór, a więc i ta sama krawędź w nieskierowanym grafie). Czasami jednak chcemy, żeby krawędź miała kierunek (tak jak droga może być jednokierunkowa). Dlatego definiuje się także grafy skierowane.

Grafem skierowanym nazywamy parę $G = (V, E)$, w której V jest zbiorem obiektów nazywanych wierzchołkami, zaś $E \subseteq V \times V$ jest zbiorem krawędzi. Grafy skierowane, podobnie jak grafy nieskierowane, przedstawiamy graficznie, przy czym krawędzie przedstawiamy jako strzałki (krawędź (a, b)

przedstawiamy jako strzałkę z wierzchołka a do wierzchołka b). Rysunek 8.2 przedstawia przykładowy graf skierowany.



Rysunek 8.2. Przykładowy graf skierowany

W grafie z rysunku 8.2 zbiór wierzchołków $V = \{0, 1, 2, 3, 4, 5\}$, zaś zbiór krawędzi to $E = \{(0, 1), (0, 2), (0, 3), (3, 0)(2, 4), (4, 1), (3, 5)\}$. Zauważmy, że w zbiorze krawędzi jest zarówno krawędź $(0, 3)$, jak i krawędź $(3, 0)$.

Inną modyfikacją grafów są grafy z wagami krawędzi. W takich grafach do każdej krawędzi przypisana jest dodatkowa wartość liczbowa nazywana wagą krawędzi. Wagi krawędzi mogą mieć różne zastosowania. Przykładowo interpretując graf jako model sieci dróg wagę krawędzi możemy interpretować jako długość drogi odpowiadającej danej krawędzi.

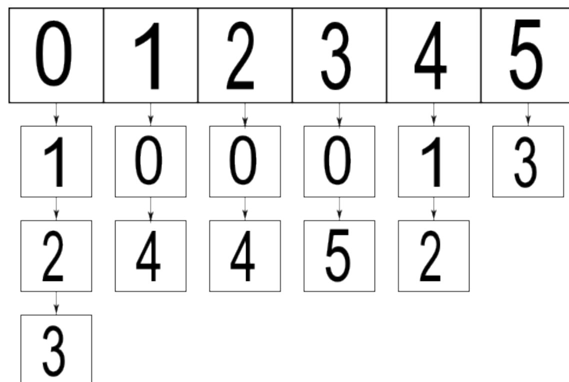
Graf w informatyce to struktura danych implementująca matematyczny graf. Są dwa najpopularniejsze sposoby reprezentacji grafów.

8.1.1. Listy sąsiedztwa

Lista sąsiedztwa to sposób reprezentacji grafu, w której graf przechowywany jest jako tablica list. W tej reprezentacji wierzchołki są ponumerowane liczbami naturalnymi. Lista przechowywana pod indeksem v przechowuje sąsiadów wierzchołka o numerze v (czyli zazwyczaj ich numery). Poszczególne listy sąsiadów mogą być implementowane zarówno za pomocą tablic, jak i list wskaźnikowych.

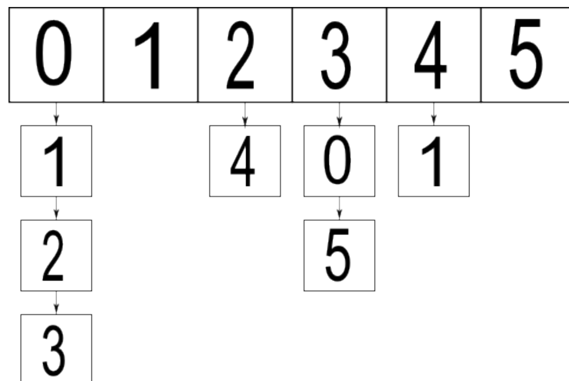
Listy sąsiedztwa nadają się zarówno do przechowywania grafów skierowanych, jak i nieskierowanych. W tym drugim przypadku lista o indeksie v przechowuje zazwyczaj wierzchołki znajdujące się po drugiej stronie krawędzi wychodzących z v.

Na rysunku 8.3 znajduje się lista sąsiedztwa dla przykładowego nieskierowanego grafu z rysunku 8.1. Natomiast na rysunku 8.4 lista sąsiedztwa dla skierowanego grafu 8.2. Widzimy, że o ile w liście sąsiedztwa grafu



Rysunek 8.3. Listy sąsiedztwa dla grafu z rysunku 8.1

nieskierowanego każdej krawędzi odpowiadają dwa wpisy w listach sąsiadów, dla krawędzi $\{0, 1\}$ dodajemy 0 do listy sąsiadów wierzchołka 1 oraz 1 do listy sąsiadów wierzchołka 0, to w grafie skierowanym każdej krawędzi odpowiada pojedynczy wpis w liście sąsiadów początkowego wierzchołka krawędzi.



Rysunek 8.4. Listy sąsiedztwa dla grafu z rysunku 8.2

W przypadku grafów z wagami w listach sąsiedztwa zamiast przechowywać same numery sąsiadów danego wierzchołka przechowujemy pary: numer sąsiada i waga krawędzi prowadzącej do danego sąsiada.

8.1.2. Macierz sąsiedztwa

Macierz sąsiedztwa to sposób reprezentacji grafów, w którym informację o krawędziach umieszczamy w dwuwymiarowej tablicy, nazwijmy ją `M`, o wymiarach $|V| \times |V|$. W tym sposobie reprezentacji grafów komórka `M[i][j]`

zwiera informację o krawędzi pomiędzy wierzchołkiem `i` a wierzchołkiem `j`. W najprostszej wersji `M[i][j]` jest różna od zera wtedy i tylko wtedy, gdy istnieje w danym grafie krawędź z wierzchołka `i` do wierzchołka `j`. W grafach z wagami krawędzi komórka `M[i][j]` zawiera zwykle wagę krawędzi z `i` do `j`. Poniżej macierze sąsiedztwa dla przykładowych grafów z rysunków 8.1 i 8.2

Tabela 8.1. Macierz sąsiedztwa dla grafu z rysunku 8.1

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	1	1	0	0
1	1	0	0	0	1	0
2	1	0	0	0	1	0
3	1	0	0	0	0	1
4	0	1	1	0	0	0
5	0	0	0	1	0	0

Tabela 8.2. Macierz sąsiedztwa dla grafu z rysunku 8.2

	0	**1**	**2**	**3**	**4**	**5**
0	0	1	1	1	0	0
1	0	0	0	0	0	0
2	0	0	0	0	1	0
3	1	0	0	0	0	5
4	0	1	0	0	0	0
5	0	0	0	0	0	0

Jak widać na powyższych przykładach, macierz sąsiedztwa dla grafów nieskierowanych (rys. 8.1.2) jest symetryczna względem diagonalnej (dla dowolnych `i, j` zachodzi `M[i][j]=M[j][i]`), natomiast dla grafów skierowanych (rys. 8.1.2) taka symetria nie zachodzi.

8.1.3. Porównanie

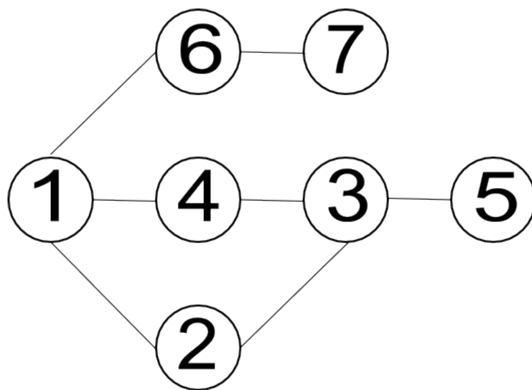
Decyzja o wyborze sposobu reprezentacji grafu zależy od implementowanego algorytmu. Jeżeli w algorytmie przeszukujemy wielokrotnie listy sąsiadów poszczególnych wierzchołków, to reprezentacja grafu za pomocą listy sąsiedztwa będzie bardziej efektywna. Natomiast jeżeli w algorytmie będziemy raczej potrzebowali informacji na temat połączenia pomiędzy parami różnych wierzchołków, to prawdopodobnie lepszym pomysłem będzie zaimplementowanie macierzy sąsiedztwa. W przypadku rzadkich grafów, tzn. grafów o małej liczbie krawędzi, listy sąsiedztwa będą miały mniejszą złożoność pamięciową.

8.2. Przechodzenie grafów

Podobnie jak w przypadku drzew, także w przypadku grafów istotną część wielu algorytmów stanowi odwiedzanie wierzchołków grafu w odpowiedniej kolejności. Istnieją dwie najpopularniejsze kolejności odwiedzania wierzchołków grafu odpowiadające dwóm algorytmom przeszukiwania grafów: w głąb (DFS – deep first search) i wszerz (BFS – breadth-first search). Najpierw przyjrzymy się algorytmowi przeszukiwania grafu w głąb.

8.2.1. Przechodzenie grafu w głąb

Przechodzenie grafu w głąb lub inaczej przeszukiwanie grafu w głąb to algorytm, który startuje od dowolnego wierzchołka i stara się iść w głąb grafu jak najdalej (stąd nazwa algorytmu). W sytuacji, gdy dojdziemy do wierzchołka grafu, którego wszystkich sąsiadów już wcześniej odwiedziliśmy, cofamy się po własnych śladach tak długo, aż wrócimy do wierzchołka, który posiada jeszcze nieodwiedzonego sąsiada. Wtedy idziemy do tego sąsiada i następnie w głąb grafu jak najdalej się da. Powyższe powtarzamy tak długo, aż wrócimy do startowego wierzchołka i nie będzie miał on już nieodwiedzonych sąsiadów. Na rysunku 8.5 przedstawiono przykładową kolejność odwiedzania wierzchołków grafu w algorytmie przechodzenia w głąb, w którym startowy wierzchołek oznaczono jedynką:



Rysunek 8.5. Przeszukiwanie grafu algorytmem DFS

Będziemy przeszukiwali graf reprezentowany przez listę sąsiedztwa zaimplementowaną jako tablicę list wskaźnikowych bez głowy. Elementy listy będą następującego typu:

Listing 8.1. Typ elementu listy sąsiadów wierzchołka

```
1 struct sasiad {
    unsigned int s;
3   sasiad * nastepny;
  };
```

Przechodzenie grafu w głąb w sposób naturalny implementuje się przy pomocy funkcji rekurencyjnej. Będzie to funkcja, która otrzyma jako argumenty listę sąsiedztwa grafu, tablicę odwiedzonych wierzchołków oraz numer wierzchołka, który ma być teraz odwiedzony.

Listing 8.2. Algorytm przeszukiwania grafu w głąb

```
  void DFS_rek(sasiad l_sasiedztwa[], bool odwiedzony[],
2                                  unsigned int wierzcholek){
    ...
4 }
```

Wewnątrz funkcji musimy zaznaczyć rozpatrywany wierzchołek jako odwiedzony i wykonać na nim wszystkie zaplanowane operacje (u nas odpowiada za to funkcja `odwiedz`).

Listing 8.3. Algorytm przeszukiwania grafu w głąb

```
  void DFS_rek(sasiad l_sasiedztwa[], bool odwiedzony[],
2                                  unsigned int wierzcholek){
    odwiedzony[wierzcholek] = true;
4   odwiedz(wierzcholek);

6   ...
  }
```

Na koniec w pętli uruchomimy rekurencyjnie funkcję `DFS` dla wszystkich nieodwiedzonych sąsiadów rozpatrywanego wierzchołka.

Listing 8.4. Algorytm przeszukiwania grafu w głąb

```
1 void DFS_rek(sasiad l_sasiedztwa[], bool odwiedzony[],
                                   unsigned int wierzcholek){
3   odwiedzony[wierzcholek] = true;
    odwiedz(wierzcholek);
5   sasiad wsk = l_sasiedztwa[wierzcholek];
    while(wsk!=NULL){
7     if (!odwiedzony[wsk->s])
        DFS_rek(l_sasiedztwa, odwiedzony, wsk->s)
9     wsk = wsk->nastepny;
    }
11 }
```

Jeżeli graf nie jest spójny, czyli istnieją w nim takie dwa wierzchołki, że z jednego nie da się przejść do drugiego idąc po krawędziach, to powyższa funkcja nie wystarczy do odwiedzenia wszystkich wierzchołków grafu. Napiszemy funkcję, która w przypadku, gdy funkcja `DFS_rek` nie odwiedzi wszystkich wierzchołków, wywołuje funkcję `DFS_rek` na pierwszym jeszcze nieodwiedzonym wierzchołku. Funkcja będzie wywoływała `DFS_rek` tak długo, aż odwiedzone zostaną wszystkie wierzchołki. Funkcja dostanie jako argument tablicę sąsiedztwa oraz liczbę wierzchołków.

Listing 8.5. Algorytm przeszukiwania grafu w głąb

```
1 void DFS(sasiad l_sasiedztwa[], unsigned int liczba_wierzch){

3   ...

5 }
```

Musimy zadeklarować, zainicjować, a na końcu usunąć z pamięci tablicę `odwiedzone`.

Listing 8.6. Algorytm przeszukiwania grafu w głąb

```
1 void DFS(sasiad l_sasiedztwa[], unsigned int liczba_wierzch){
    bool *odwiedzone = new[liczba_wierzch];
3   for(unsigned int i=0;i<liczba_wierzch;i++)
      odwiedzone[i] = false;
5
    ...
7
    delete [] odwiedzone;
9 }
```

Na koniec przeglądamy tablicę `odwiedzone` i wywołujemy funkcję `DFS_rek` dla każdego nieodwiedzonego wierzchołka.

Listing 8.7. Algorytm przeszukiwania grafu w głąb

```
1 void DFS(sasiad l_sasiedztwa[], unsigned int liczba_wierzch){
    bool *odwiedzone = new bool[liczba_wierzch];
3   for(unsigned int i=0;i<liczba_wierzch;i++)
      odwiedzone[i] = false;
5   for(int i=0;i<liczba_wierzch;i++)
      if (!odwiedzone[i])
7       DFS_rek(l_sasiedztwa, odwiedzone, i);
    delete [] odwiedzone;
9 }
```

Złożoność całego powyższego algorytmu to $O(|V|+|E|)$, gdzie $|V|$ to liczba wierzchołków, zaś $|E|$ to liczba krawędzi grafu. Wynika to z faktu, że każdy wierzchołek odwiedzamy dokładnie raz, a każdą krawędź dokładnie dwa razy przeglądając listy sąsiadów wierzchołków połączonych daną krawędzią. Taka złożoność nie byłaby możliwa do osiągnięcia, gdybyśmy graf reprezentowali jako macierz sąsiedztwa, gdyż wtedy przejrzenie listy sąsiadów pojedynczego wierzchołka wymagałoby przejrzenia całego wiersza macierzy sąsiedztwa, czyli $O(|V|)$ operacji. Złożoność całego algorytmu wyniosłaby wtedy $O(|V|^2)$.

Na koniec tego podrozdziału zobaczymy iteracyjną wersję funkcji `DFS_rek`, która zamiast rekurencji wykorzystuje stos:

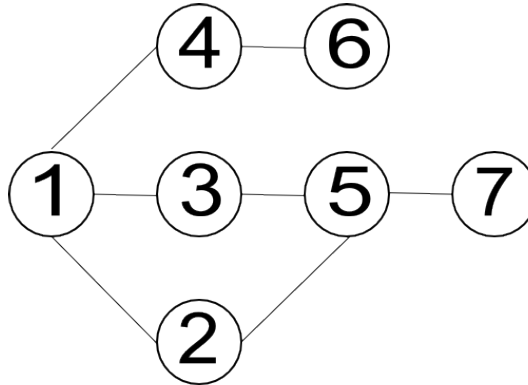
Listing 8.8. Algorytm przeszukiwania grafu w głąb

```
1 void DFS_it(sasiad l_sasiedztwa[], bool odwiedzony[],
                                  unsigned int wierzcholek){
3   Stos S;
    S.push(wierzcholek);
5   while(!S.empty()){
      unsigned int pom=S.top();
7     S.pop();
      if (!odwiedzony[pom]){
9       odwiedzony[pom]=true;
        odwiedz(pom);
11      sasiad * wsk=l_sasiedztwa[pom];
        while(wsk!=NULL){
13        S.push(wsk->s);
          wsk=wsk->nastepny;
15      }\\while
      }\\if
17  }\\while
  }\\dfs
```

8.2.2. Przechodzenie grafu wszerz

Przechodzenie/przeszukiwanie grafu wszerz to algorytm, który przechodzi wierzchołki grafu warstwami. Zaczyna od wierzchołka startowego, potem odwiedza jego sąsiadów, następnie sąsiadów sąsiadów itd. Na rysunku 8.6 numery wierzchołków pokazują kolejność przeglądania wierzchołków przez algorytm BFS rozpoczynający działanie od wierzchołka 1.

Implementacja algorytmu przeszukiwania grafu wszerz od iteracyjnej implementacji przeszukiwania grafu w głąb różni się tylko tym, że w miejsce stosu używamy kolejki. My dodatkowo zmienimy nieco sposób reprezentacji grafu. Nadal będziemy używać list sąsiedztwa, ale tym razem pojedyncze listy sąsiadów zaimplementujemy przy pomocy tablic. Pierwszy element ta-



Rysunek 8.6. Przeszukiwanie grafu algorytmem BFS

kiej tablicy będzie zawierał liczbę sąsiadów danego wierzchołka, zaś kolejne elementy tablicy numery kolejnych sąsiadów.

Listing 8.9. Algorytm przeszukiwania grafu wszerz

```
   void BFS(unsigned int ** l_sasiedztwa, bool odwiedzony[],
 2                                       unsigned int wierzcholek){
     Kolejka K;
 4   K.push(wierzcholek);
     while(!K.empty()){
 6     unsigned int pom=K.front();
       K.pop();
 8     if (!odwiedzony[pom]){
         odwiedzony[pom]=true;
10       odwiedz(pom);
         for(unsigned int i=1;i<=l_sasiedztwa[pom][0];i++{
12         S.push(l_sasiedztwa[pom][i]);
       }\\if
14   }\\while
   }\\dfs
```

Bibliografia

[1] L. Banachowski, K. Diks, W. Rytter, *Algorytmy i struktury danych*, WNT, Warszawa 2006.

[2] T. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[3] D. Harel, *Rzecz o istocie informatyki. Algorytmika*, WNT, Warszawa 2008.

[4] D. Knuth. *Sztuka programowania*, t. 1-3, WNT, Warszawa 2002.

[5] W. Lipski, *Kombinatoryka dla programistów*, WNT, Warszawa 2009.

[6] Ch. Papadimitriou, *Złożoność obliczeniowa*, WNT, Warszawa 2002.

[7] Studia informatyczne – `http://wazniak.mimuw.edu.pl` – Metody programowania, Algorytmy i struktury danych.

[8] `http://pl.wikipedia.org/wiki/Teoria_złożoności`