IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

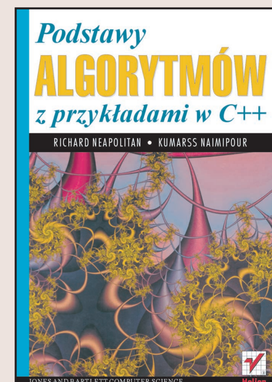
ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

Podstawy algorytmów z przykładami w C++



Autorzy: Richard Neapolitan, Kumarss Naimipour
Tłumaczenie: Bartłomiej Garbacz, Paweł Gonera, Artur Szczepaniak, Mikołaj Szczepaniak
ISBN: 83-7361-429-X
Tytuł oryginału: Foundations of Algorithms. Using C++ Pseudocode
Format: B5, stron: 648

Algorytmy są jednym z fundamentów programowania. Prawidłowo zaprojektowany algorytm jest podstawą efektywnego i niezawodnego programu. Opisanie problemu w postaci algorytmu nie jest prostym zadaniem — wymaga wiedzy z zakresu matematyki, umiejętności oceny złożoności obliczeniowej i znajomości zasad optymalizacji obliczeń. Istnieje wiele metod projektowania algorytmów. Znajomość tych metod znacznie ułatwia analizę zagadnienia i przedstawienie go w postaci zalgorytmizowanej.

Książka „Podstawy algorytmów z przykładami w C++" to kompletny podręcznik poświęcony tym właśnie zagadnieniom. Przedstawia sposoby podejścia do rozwiązywania zagadnień projektowych, udowadnia, że sporo z nich można zrealizować różnymi metodami, a także uczy, jak dobrać właściwą metodę do postawionego problemu. Materiał podzielony jest na wykłady, zilustrowane pseudokodem przypominającym język C++, co bardzo ułatwia zastosowanie poznanej wiedzy w praktyce.

- Wprowadzenie do projektowania algorytmów
- Zastosowanie techniki dziel i zwyciężaj
- Algorytmy programowania dynamicznego
- Analiza złożoności obliczeniowej algorytmów na przykładzie lgorytmów sortowania i przeszukiwania
- Algorytmy z zakresu teorii liczb
- Algorytmy kompresji danych i kryptografii
- Programowanie równoległe

Wykłady poświęcone algorytmom są uzupełnione dodatkami, zawierającymi kompendium niezbędnej wiedzy z dziedziny matematyki, technik rekurencyjnych i algebry zbiorów.

„Podstawy algorytmów z przykładami w C++" to doskonały podręcznik dla uczniów, studentów i wszystkich, którzy chcą poznać tę dziedzinę wiedzy.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

O Autorach **9**

Przedmowa **11**

Rozdział 1. Algorytmy — wydajność, analiza i rząd **17**

1.1. Algorytmy 18
1.2. Znaczenie opracowywania wydajnych algorytmów 25
1.2.1. Wyszukiwanie sekwencyjne a wyszukiwanie binarne 26
1.2.2. Ciąg Fibonacciego 28
1.3. Analiza algorytmów 33
1.3.1. Analiza złożoności 33
1.3.2. Zastosowanie teorii 41
1.3.3. Analiza poprawności 42
1.4. Rząd 42
1.4.1. Intuicyjne wprowadzenie do problematyki rzędu 43
1.4.2. Formalne wprowadzenie do problematyki rzędu 45
⊕ 1.4.3. Wykorzystanie granic do określenia rzędu 56
1.5. Zarys dalszej treści książki 59
Ćwiczenia 60

Rozdział 2. Dziel i zwyciężaj **65**

2.1. Wyszukiwanie binarne 66
2.2. Sortowanie przez scalanie 71
2.3. Podejście typu dziel i zwyciężaj 77
2.4. Sortowanie szybkie (sortowanie przez podstawienie podziałów) 78
2.5. Algorytm Strassena mnożenia macierzy 85
2.6. Arytmetyka wielkich liczb całkowitych 90
2.6.1. Reprezentacja wielkich liczb całkowitych: dodawanie i inne operacje wykonywane w czasie liniowym 91
2.6.2. Mnożenie wielkich liczb całkowitych 91
2.7. Określanie wartości progowych 97
2.8. Przeciwwskazania do używania podejścia dziel i zwyciężaj 101
Ćwiczenia 103

Rozdział 3. Programowanie dynamiczne **111**

3.1. Współczynnik dwumianowy 112
3.2. Algorytm Floyda określania najkrótszej drogi w grafie 117
3.3. Programowanie dynamiczne a problemy optymalizacyjne 125
3.4. Łańcuchowe mnożenie macierzy 127
3.5. Optymalne drzewa wyszukiwania binarnego 136
3.6. Problem komiwojażera 145
Ćwiczenia 152

Rozdział 4. Podejście zachłanne 157
4.1. Minimalne drzewo rozpinające 161
4.1.1. Algorytm Prima 163
4.1.2. Algorytm Kruskala 169
4.1.3. Porównanie algorytmu Prima z algorytmem Kruskala 173
4.1.4. Dyskusja końcowa 173
4.2. Algorytm Dijkstry najkrótszych dróg z jednego źródła 174
4.3. Szeregowanie 177
4.3.1. Minimalizacja całkowitego czasu w systemie 178
4.3.2. Szeregowanie z terminami granicznymi 180
4.4. Kod Huffmana 188
4.4.1. Kody prefiksowe 189
4.4.2. Algorytm Huffmana 190
4.5. Podejście zachłanne a programowanie dynamiczne: problem plecakowy 194
4.5.1. Podejście zachłanne do problemu plecakowego 0-1 195
4.5.2. Podejście zachłanne do ułamkowego problemu plecakowego 196
4.5.3. Podejście programowania dynamicznego do problemu plecakowego 0-1 197
4.5.4. Poprawiona wersja algorytmu programowania dynamicznego dla problemu plecakowego 0-1 197
Ćwiczenia 201

Rozdział 5. Algorytmy z powrotami 207
5.1. Techniki algorytmów z powrotami 208
5.2. Problem n-królowych 215
5.3. Zastosowanie algorytmu Monte Carlo do oszacowania wydajności algorytmu z powrotami 219
5.4. Problem sumy podzbiorów 223
5.5. Kolorowanie grafu 228
5.6. Problem cyklu Hamiltona 232
5.7. Problem plecakowy 0-1 235
5.7.1. Algorytm z powrotami dla problemu plecakowego 0-1 235
5.7.2. Porównanie algorytmu programowania dynamicznego z algorytmem z powrotami do rozwiązywania problemu plecakowego 0-1 244
Ćwiczenia 245

Rozdział 6. Metoda podziału i ograniczeń 251
6.1. Ilustracja metody podziału i ograniczeń dla problemu plecakowego 0-1 253
6.1.1. Przeszukiwanie wszerz z przycinaniem metodą podziału i ograniczeń 253
6.1.2. Przeszukiwanie typu najpierw najlepszy z przycinaniem metodą podziału i ograniczeń 258
6.2. Problem komiwojażera 264
6.3. Wnioskowanie abdukcyjne (diagnozowanie) 272
Ćwiczenia 281

Rozdział 7. Wprowadzenie do złożoności obliczeniowej: problem sortowania 285
7.1. Złożoność obliczeniowa 286
7.2. Sortowanie przez wstawianie i sortowanie przez wybieranie 288
7.3. Dolne ograniczenia dla algorytmów usuwających co najwyżej jedną inwersję dla jednej operacji porównania 294
7.4. Przypomnienie algorytmu sortowania przez scalanie 297
7.5. Przypomnienie algorytmu szybkiego sortowania 303
7.6. Sortowanie stogowe 305
7.6.1. Stogi i podstawowe na nich operacje 305
7.6.2. Implementacja algorytmu sortowania stogowego 309

7.7. Zestawienie algorytmów sortowania przez scalanie, sortowania szybkiego i sortowania stogowego ... 316
7.8. Dolne ograniczenia dla algorytmów sortujących wyłącznie na podstawie operacji porównania kluczy ... 317
7.8.1. Drzewa decyzyjne dla algorytmów sortujących ... 317
7.8.2. Dolne ograniczenia dla najgorszego przypadku ... 320
7.8.3. Dolne ograniczenia dla średniego przypadku ... 323
7.9. Sortowanie przez podział (sortowanie pozycyjne) ... 328
Ćwiczenia ... 332

Rozdział 8. Więcej o złożoności obliczeniowej: problem przeszukiwania ... 339

8.1. Dolne ograniczenia dla przeszukiwania wyłącznie na podstawie porównywania wartości kluczy ... 340
8.1.1. Dolne ograniczenia dla najgorszego przypadku ... 343
8.1.2. Dolne ograniczenia dla średniego przypadku ... 345
8.2. Przeszukiwanie przez interpolację ... 351
8.3. Przeszukiwanie w drzewach ... 354
8.3.1. Drzewa wyszukiwania binarnego ... 355
8.3.2. B-drzewa ... 360
8.4. Mieszanie ... 361
8.5. Problem wyboru: wprowadzenie metody dyskusji z adwersarzem ... 367
8.5.1. Znajdowanie największego klucza ... 367
8.5.2. Znajdowanie zarówno najmniejszego, jak i największego klucza ... 369
8.5.3. Znajdowanie drugiego największego klucza ... 376
8.5.4. Znajdowanie k-tego najmniejszego klucza ... 381
8.5.5. Algorytm probabilistyczny dla problemu wyboru ... 390
Ćwiczenia ... 395

Rozdział 9. Złożoność obliczeniowa i trudność problemów: wprowadzenie do teorii o zbiorze NP ... 401

9.1. Trudność ... 402
9.2. Ponowne omówienie zagadnienia rozmiaru danych wejściowych ... 404
9.3. Trzy główne kategorie problemów ... 409
9.3.1. Problemy, dla których wynaleziono algorytmy wielomianowe ... 409
9.3.2. Problemy, których trudność została udowodniona ... 410
9.3.3. Problemy, których trudność nie została udowodniona, jednak nie udało się znaleźć rozwiązujących je algorytmów wielomianowych ... 411
9.4. Teoria o zbiorze NP ... 411
9.4.1. Zbiory P i NP ... 414
9.4.2. Problemy NP-zupełne ... 419
9.4.3. Problemy NP-trudne, NP-łatwe i NP-równoważne ... 431
9.5. Sposoby rozwiązywania problemów NP-trudnych ... 435
9.5.1. Algorytm przybliżony dla problemu komiwojażera ... 436
9.5.2. Przybliżony algorytm dla problemu pakowania ... 441
Ćwiczenia ... 446

Rozdział 10. Algorytmy teorii liczb ... 449

10.1. Przegląd teorii liczb ... 450
10.1.1. Liczby złożone i liczby pierwsze ... 450
10.1.2. Największy wspólny dzielnik ... 451
10.1.3. Rozkładanie liczb całkowitych na czynniki pierwsze ... 455
10.1.4. Najmniejsza wspólna wielokrotność ... 457
10.2. Wyznaczanie największego wspólnego dzielnika ... 457
10.2.1. Algorytm Euklidesa ... 458
10.2.2. Rozszerzenie algorytmu Euklidesa ... 462

10.3. Przegląd arytmetyki modularnej ... 465
10.3.1. Teoria grup ... 465
10.3.2. Kongruencja modulo n ... 467
10.3.3. Podgrupy ... 473
10.4. Rozwiązywanie modularnych równań liniowych ... 479
10.5. Obliczanie modularnych potęg ... 485
10.6. Znajdowanie wielkich liczb pierwszych ... 488
10.6.1. Szukanie liczby pierwszej ... 488
10.6.2. Sprawdzanie, czy dana liczba całkowita jest liczbą pierwszą ... 489
10.7. System szyfrowania RSA z publicznym kluczem ... 508
10.7.1. Systemy szyfrowania z kluczem publicznym ... 508
10.7.2. System szyfrowania RSA ... 509
Ćwiczenia ... 512

Rozdział 11. Wprowadzenie do algorytmów równoległych ... 517

11.1. Architektury równoległe ... 521
11.1.1. Mechanizm sterowania ... 521
11.1.2. Organizacja przestrzeni adresowej ... 522
11.1.3. Sieci łączące ... 524
11.2. Model PRAM ... 527
11.2.1. Projektowanie algorytmów dla modelu CREW PRAM ... 531
11.2.2. Projektowanie algorytmów dla modelu CRCW PRAM ... 538
Ćwiczenia ... 541

Dodatek A Przegląd niezbędnej wiedzy matematycznej ... 543

A.1. Notacja ... 543
A.2. Funkcje ... 545
A.3. Indukcja matematyczna ... 546
A.4. Twierdzenia i lematy ... 553
A.5. Logarytmy ... 554
A.5.1. Definicja i właściwości logarytmów ... 554
A.5.2. Logarytm naturalny ... 557
A.6. Zbiory ... 559
A.7. Permutacje i kombinacje ... 560
A.8. Prawdopodobieństwo ... 563
A.8.1. Losowość ... 569
A.8.2. Wartość oczekiwana ... 573
Ćwiczenia ... 575

Dodatek B Rozwiązywanie równań rekurencyjnych na potrzeby analizy algorytmów rekurencyjnych ... 581

B.1. Rozwiązywanie rekurencji za pomocą indukcji ... 581
B.2. Rozwiązywanie rekurencji z zastosowaniem równań charakterystycznych ... 585
B.2.1. Homogeniczna rekurencja liniowa ... 585
B.2.2. Niehomogeniczna rekurencja liniowa ... 594
B.2.3. Zamiana zmiennej (przekształcenie dziedziny) ... 600
B.3. Rozwiązywanie rekurencji przez podstawianie ... 603
B.4. Rozszerzanie wyników dla n będącego potęgą dodatniej stałej b do wyników dla dowolnego n ... 605
B.5. Dowody twierdzeń ... 611
Ćwiczenia ... 614

Dodatek C Struktury danych dla zbiorów rozłącznych ... 621

Dodatek D Bibliografia ... 631

Rozdział 3.
Programowanie dynamiczne

Jak zapewne Czytelnik pamięta, liczba składników obliczanych przez algorytm typu *dziel i zwyciężaj* w celu określenia *n*-tego wyrazu ciągu Fibonacciego (algorytm 1.6) jest wykładnicza w stosunku do *n*. Wynika to z faktu, że podejście typu *dziel i zwyciężaj* rozwiązuje realizację problemu poprzez jej podział na mniejsze realizacje, a następnie bezpośrednie rozwiązywanie owych mniejszych realizacji. Jak stwierdzono w rozdziale 2., jest to podejście zstępujące. Sprawdza się ono w przypadku problemów w rodzaju sortowania przez scalanie, gdzie mniejsze instancje nie są ze sobą powiązane. Dzieje się tak, ponieważ każda z nich składa się z tablicy kluczy, które muszą zostać posortowane niezależnie. Jednakże w przypadku problemów takich, jak *n*-ty wyraz ciągu Fibonacciego, mniejsze instancje są ze sobą powiązane. Przykładowo, zgodnie z tym, co przedstawiono w podrozdziale 1.2, obliczenie piątego wyrazu ciągu Fibonacciego wymaga obliczenia wyrazu trzeciego i czwartego. Jednak procesy określania czwartego i trzeciego wyrazu są ze sobą związane o tyle, że oba wymagają znajomości wyrazu drugiego. Ze względu na fakt, że algorytm typu *dziel i zwyciężaj* wykonuje oba procesy niezależnie, drugi wyraz ciągu Fibonacciego jest obliczany wielokrotnie. W przypadku problemów,

w których mniejsze realizacje są ze sobą powiązane, często okazuje się, że algorytm typu *dziel i zwyciężaj* wielokrotnie rozwiązuje te same realizacje i w efekcie jest on bardzo niewydajny.

Programowanie dynamiczne (ang. *dynamic programming*), technika omawiana w niniejszym rozdziale, opiera się na przyjęciu odwrotnego podejścia. Programowanie dynamiczne jest podobne do podejścia typu *dziel i zwyciężaj* o tyle, że realizacja problemu zostaje podzielona na mniejsze realizacje. Jednak tym razem najpierw są rozwiązywane małe realizacje, a ich wyniki zostają przechowane; później, kiedy zajdzie taka potrzeba, algorytm może się do nich bezpośrednio odwoływać, zamiast obliczać je ponownie. Pojęcie *programowanie dynamiczne* wywodzi się z teorii sterowania i w tym kontekście *programowanie* oznacza użycie tablicy (tabeli), w ramach której jest konstruowane rozwiązanie. Jak wspomniano w rozdziale 1., wydajny algorytm (algorytm 1.7) obliczania n-tego wyrazu ciągu Fibonacciego jest przykładem programowania dynamicznego. Algorytm ten określa n-ty wyraz ciągu Fibonacciego poprzez konstruowanie po kolei pierwszych $n+1$ wyrazów w tablicy f indeksowanej od 0 do n. W przypadku algorytmu programowania dynamicznego konstruujemy rozwiązanie „od dołu” tablicy (lub ciągu tablic). Programowanie dynamiczne stanowi więc ***podejście wstępujące*** (ang. *bottom-up*). Niekiedy, jak w przypadku algorytmu 1.7, po opracowaniu algorytmu wykorzystującego tablicę (lub ciąg tablic) istnieje możliwość ulepszenia go, tak aby zwolnić wiele niepotrzebnie przydzielonej przestrzeni pamięciowej.

Opracowanie algorytmu programowania dynamicznego polega na wykonaniu następujących działań:

1. *Określamy* właściwość rekurencyjną, która pozwala znaleźć rozwiązanie realizacji problemu.
2. Rozwiązujemy realizację problemu zgodnie z podejściem *wstępującym*, najpierw rozwiązując mniejsze realizacje.

W celu zilustrowania tych działań w podrozdziale 3.1 przedstawiono kolejny prosty przykład programowania dynamicznego. W pozostałych podrozdziałach przedstawiono bardziej zaawansowane przykłady wykorzystania tego typu programowania.

3.1. Współczynnik dwumianowy

Współczynnik dwumianowy (ang. *binomial coefficient*), który omówiono w podrozdziale A.7 w dodatku A, jest pokreślony zależnością:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{dla } 0 \le k \le n$$

Dla wartości n i k, które nie są małe, nie możemy obliczyć współczynnika dwumianowego bezpośrednio z tej definicji, ponieważ wartość $n!$ jest bardzo duża nawet dla średnich wartości n. W ćwiczeniach zostanie wykazane, że:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ lub } k = n \end{cases} \tag{3.1}$$

Możemy wyeliminować konieczność obliczania wyrażenia $n!$ lub $k!$, wykorzystując właściwość rekurencyjną. Sugeruje to zdefiniowanie poniższego algorytmu typu *dziel i zwyciężaj*.

Algorytm 3.1. Obliczanie współczynnika dwumianowego przy użyciu podejścia typu dziel i zwyciężaj

Problem: oblicz współczynnik dwumianowy.

Dane wejściowe: nieujemne liczby całkowite n i k, gdzie $k \leq n$.

Dane wyjściowe: *bin* — wartość współczynnika dwumianowego $\binom{n}{k}$.

```
int bin (int n, int k)
{
   if (k == 0 || n == k)
      return 1;
   else
      return bin(n - 1, k - 1)+bin(n-1, k);
}
```

Podobnie jak w przypadku algorytmu 1.6 (n-ty wyraz ciągu Fibonacciego) algorytm ten jest bardzo niewydajny. W ćwiczeniach Czytelnik zostanie poproszony o wykazanie, że algorytm oblicza

$$2\binom{n}{k} - 1$$

wyrazów w celu obliczenia wartości $\binom{n}{k}$. Problem polega na tym, że te same realizacje są rozwiązywane w każdym wywołaniu rekurencyjnym. Przykładowo wywołania $bin(n–1, k–1)$ oraz $bin(n–1, k)$ wiążą się z obliczeniem wartości $bin(n–2, k–1)$ i realizacja ta jest rozwiązywana niezależnie w każdym wywołaniu. Jak wspomniano w podrozdziale 2.8, podejście typu *dziel i zwyciężaj* nigdy nie jest wydajne, kiedy realizacja jest dzielona na dwie mniejsze realizacje, których rozmiar jest zbliżony do rozmiaru oryginalnej realizacji.

Poniżej opracujemy bardziej wydajny algorytm, wykorzystujący programowanie dynamiczne. W równaniu 3.1 określiliśmy już właściwość rekurencyjną. Wykorzystamy ją do skonstruowania rozwiązania wykorzystującego tablicę B, gdzie $B[i][j]$ będzie zawierać wartość $\binom{i}{j}$. Poniżej opisano kolejne etapy tworzenia algorytmu, opartego na programowaniu dynamicznym.

1. *Określamy* właściwość rekurencyjną. Dokonaliśmy już tego w równaniu 3.1. Zapisując je w kontekście użycia tablicy B otrzymujemy:

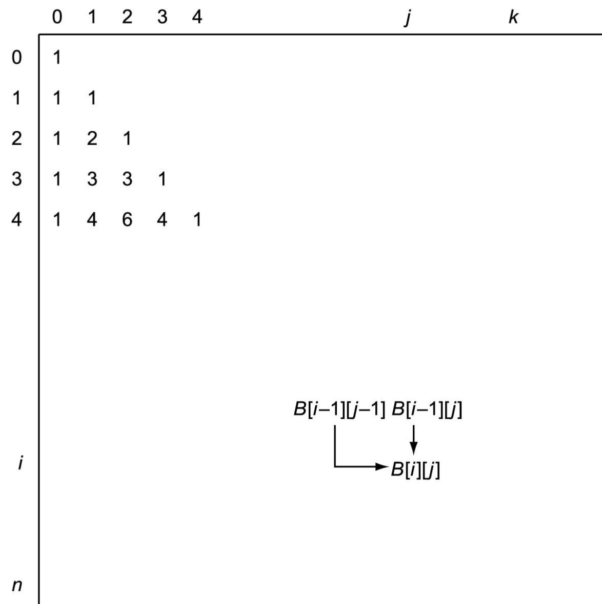
$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \text{ lub } \mathrm{j} = i \end{cases}$$

2. Rozwiązujemy realizację problemu w porządku *wstępującym*, rozpoczynając od pierwszego wiersza i wyliczając po kolei wartości w wierszach tablicy B.

Na rysunku 3.1 zilustrowano przebieg etapu 2. (Czytelnik powinien rozpoznać w przedstawionej tablicy trójkąt Pascala). Każdy następny wiersz jest wyliczany na podstawie wartości wiersza poprzedzającego przy użyciu właściwości rekurencyjnej, określonej w etapie 1. Ostatnia obliczana wartość, $B[n][k]$ to $\binom{n}{k}$.

Przykład 3.1 stanowi odzwierciedlenie tych działań. Należy zauważyć, że są w nim obliczane tylko dwie pierwsze kolumny. Wynika to z faktu, że $k = 2$ i, ogólnie rzecz biorąc, musimy obliczyć wartości w każdym wierszu tylko do k-tej kolumny. W przykładzie 3.1 obliczono wartość $B[0][0]$, ponieważ współczynnik dwumianowy jest zdefiniowany dla $n = k = 0$. Stąd algorytm wykonuje ten etap, nawet jeśli wartość nie jest wykorzystywana w dalszych obliczeniach.

Rysunek 3.1. *Tablica B, używana do obliczenia współczynnika dwumianowego*



Przykład 3.1. Obliczamy wartość $B[4][2] = \binom{4}{2}$.

Obliczamy wiersz 0.: {Etap ten jest wykonywany wyłącznie w celu dokładnego odwzorowania algorytmu}.

{Wartość $B[0][0]$ nie jest potrzebna w dalszych obliczeniach}.

$B[0][0] = 1$

Obliczamy wiersz 1.:

$B[1][0] = 1$
$B[1][1] = 1$

Obliczamy wiersz 2.:

$B[2][0] = 1$
$B[2][1] = B[1][0]+B[1][1] = 1+1 = 2$
$B[2][2] = 1$

Obliczamy wiersz 3.:

$B[3][0] = 1$
$B[3][1] = B[2][0]+B[2][1] = 1+2 = 3$
$B[3][2] = B[2][1]+B[2][2] = 2+1 = 3$

Obliczamy wiersz 4.:

$B[4][0] = 1$
$B[4][1] = B[3][0]+B[3][1] = 1+3 = 4$
$B[4][2] = B[3][1]+B[3][2] = 3+3 = 6$

W przykładzie 3.1 obliczaliśmy po kolei coraz większe wartości współczynnika dwumianowego. W każdym przebiegu wartości potrzebne do wykonania bieżących działań były już obliczone i zachowane. Procedura ta ma fundamentalne znaczenie dla programowania dynamicznego. Poniższy algorytm stanowi implementację opisanego podejścia do obliczania współczynnika dwumianowego.

Algorytm 3.2. Obliczanie współczynnika dwumianowego przy użyciu programowania dynamicznego

Problem: oblicz współczynnik dwumianowy.

Dane wejściowe: nieujemne liczby całkowite n i k, gdzie $k \leq n$.

Dane wyjściowe: *bin2* — wartość współczynnika dwumianowego $\binom{n}{k}$.

```
int bin2 (int n, int k)
{
   index i, j;
   int B[0..n][0..k];

   for (i = 0; i <= n; i++)
      for (j = 0; j <= minimum(i, k); j++)
         if (j == 0 || j == i)
            B[i][j] = 1;
         else
            B[i][j] = B[i-1][j-1]+B[i-1][j];
   return B[n][k];
}
```

Parametry n i k nie są rozmiarem danych wejściowych w przypadku tego algorytmu. Stanowią one dane wejściowe, natomiast rozmiarem danych wejściowych jest liczba symboli użytych do ich zakodowania. Z podobną sytuacją mieliśmy do czynienia w podrozdziale 1.3, gdzie omawialiśmy algorytm obliczający n-ty wyraz ciągu Fibonacciego. Jednakże wciąż możemy zyskać wgląd w wydajność działania algorytmu poprzez określenie ilości wykonywanych działań w funkcji zmiennych n i k. Dla danego n i k obliczymy liczbę przebiegów pętli `for`-j. W poniższej tabeli 3.1 zawarto liczby przebiegów dla każdej wartości i.

Tabela 3.1. *Liczba przebiegów pętli for-j w zależności od wartości zmiennej i*

i	0	1	2	3	…	k	k+1	…	n
Liczba przebiegów	1	2	3	4	…	k+1	k+1	…	k+1

Całkowitą liczbę przebiegów określa więc zależność:

$$1+2+3+4+\ldots+k+\underbrace{(k+1)+(k+1)+\ldots+(k+1)}_{n-k+1 \text{ razy}}$$

Wykorzystując wynik otrzymany w przykładzie A.1 w dodatku A otrzymujemy następujące wyrażenie:

$$\frac{k(k+1)}{2}+(n-k+1)(k+1)=\frac{(2n-k+2)(k+1)}{2}\in\Theta(nk)$$

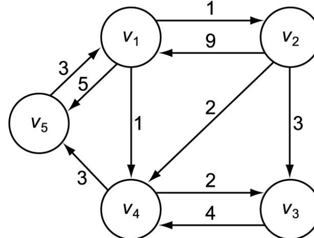
Wykorzystując programowanie dynamiczne zamiast podejścia typu *dziel i zwyciężaj* udało nam się opracować znacznie wydajniejszy algorytm. Jak wcześniej wspomniano, programowanie dynamiczne jest podobne do metody *dziel i zwyciężaj* o tyle, że szukamy właściwości rekurencyjnej, która dzieli realizację problemu na mniejsze realizacje. Różnica polega na tym, że w przypadku programowania dynamicznego wykorzystujemy właściwość rekurencyjną w celu iteracyjnego rozwiązania realizacji w kolejnych krokach, rozpoczynając od najmniejszej realizacji, zamiast nieuzasadnionego nadużywania rekurencji. W ten sposób każdą mniejszą realizację rozwiązujemy tylko raz. Programowanie dynamiczne jest dobrym rozwiązaniem do wypróbowania w sytuacji, gdy metoda *dziel i zwyciężaj* daje w wyniku bardzo mało wydajny algorytm.

Najprostszym sposobem przedstawienia algorytmu 3.2 jest utworzenie dwuwymiarowej tablicy B. Jednak kiedy obliczy się wartości w danym wierszu, nie są już potrzebne wartości obliczone w wierszu poprzednim. Stąd algorytm można zapisać przy użyciu tablicy jednowymiarowej, indeksowanej od 0 do k. Tego rodzaju modyfikację zawarto w ćwiczeniach. Kolejnym ulepszeniem algorytmu jest wykorzystanie faktu, że $\binom{n}{k}=\binom{n}{n-k}$.

3.2. Algorytm Floyda określania najkrótszej drogi w grafie

Częstym problemem podróżnych latających samolotami jest znalezienie najkrótszej drogi z jednego miasta do drugiego w przypadku, gdy nie istnieje połączenie bezpośrednie. Poniżej opracujemy algorytm rozwiązywania tego typu problemów. Najpierw jednak przedstawimy nieformalne wprowadzenie do teorii grafów. Na rysunku 3.2 przedstawiono ważony graf skierowany. W graficznej reprezentacji grafu kółka oznaczają ***wierzchołki*** (ang. *vertices*), natomiast linie łączące jeden wierzchołek z drugim to ***krawędzie*** (ang. *edges*). Jeżeli do wszystkich krawędzi zostanie przypisany odpowiedni kierunek, graf jest określany mianem ***grafu skierowanego*** (ang. *directed graph*) lub ***digrafu***. Rysując krawędź w takim grafie używamy strzałki w celu przedstawienia kierunku. W digrafie mogą istnieć dwie krawędzie między dwoma wierzchołkami, każda biegnąca w innym kierunku. Na przykład na rysunku 3.2 mamy krawędź z wierzchołka v_1 do wierzchołka v_2 oraz z wierzchołka v_2 do v_1. Jeżeli krawędzie posiadają związane ze sobą wartości, są one nazywane ***wagami*** (ang. *weights*), zaś graf określa się mianem ***grafu ważonego*** (ang. *weighted graph*). Zakładamy, że wagi są nieujemne. Choć określa się je zwykle mianem wag, w wielu zastosowaniach reprezentują one odległości. Stąd mówimy o drodze z jednego wierzchołka do innego. W grafie skierowanym *droga* (ang. *path*) to sekwencja wierzchołków, taka że istnieje krawędź z każdego wierzchołka do jego następnika. Przykładowo, na rysunku 3.2 sekwencja $[v_1, v_4, v_3]$ jest drogą, gdyż istnieje krawędź z wierzchołka v_1 do v_4 oraz z v_4 do v_3. Sekwencja $[v_3, v_4, v_1]$ nie jest drogą, gdyż nie istnieje krawędź z wierzchołka v_4 do v_1. Droga z wierzchołka do niego samego nosi nazwę ***cyklu*** (ang. *cycle*). Droga $[v_1, v_4, v_5, v_1]$ na rysunku 3.2 jest cyklem. Jeżeli graf zawiera cykl, jest grafem ***cyklicznym*** (ang. *cyclic*). W przeciwnym wypadku nosi nazwę ***acyklicznego*** (ang. *acyclic*). Droga jest nazywana drogą ***prostą*** (ang. *simple*), jeżeli nie przechodzi dwa razy przez ten sam wierzchołek. Na rysunku 3.2 droga $[v_1, v_2, v_3]$ jest prosta, ale droga $[v_1, v_4, v_5, v_1, v_2]$ nie jest prosta. Należy zauważyć, że droga prosta nigdy nie zawiera poddrogi, która byłaby cykliczna. ***Długością*** (ang. *length*) drogi w grafie skierowanym jest suma wag krawędzi należących do drogi. W przypadku grafu nieważonego jest to po prostu liczba krawędzi należących do drogi.

Rysunek 3.2.
Graf skierowany z podanymi wagami



Często spotykanym problemem jest znalezienie najkrótszych dróg z każdego wierzchołka do wszystkich innych wierzchołków. Oczywiście, najkrótsza droga musi być drogą prostą. Na rysunku 3.2 istnieją trzy drogi proste z wierzchołka v_1 do v_3: $[v_1, v_2, v_3]$, $[v_1, v_4, v_3]$ oraz $[v_1, v_2, v_4, v_3]$. Z uwagi na fakt, że:

$$długość\,[v_1, v_2, v_3] = 1 + 3 = 4$$
$$długość\,[v_1, v_4, v_3] = 1 + 2 = 3$$
$$długość\,[v_1, v_2, v_4, v_3] = 1 + 2 + 2 = 5$$

najkrótszą drogą z v_1 do v_3 jest $[v_1, v_4, v_3]$. Jak wcześniej wspomniano, jednym z często wykorzystywanych zastosowań problemu najkrótszej drogi jest określanie najkrótszych tras między miastami.

Problem najkrótszej drogi to ***problem optymalizacji*** (ang. *optimization problem*). Może istnieć więcej niż jedno rozwiązanie kandydujące do miana najlepszego dla problemu optymalizacji. Każde rozwiązanie kandydujące posiada związaną ze sobą wartość i rozwiązaniem realizacji jest rozwiązanie kandydujące o optymalnej wartości. W zależności od problemu wartością optymalną może być minimum lub maksimum. W przypadku problemu najkrótszej drogi *rozwiązaniem kandydującym* (ang. *candidate solution*) jest droga z jednego wierzchołka do drugiego, *wartością* jest długość tej drogi, zaś *wartością optymalną* jest minimum tych długości.

Ze względu na fakt, że może istnieć wiele najkrótszych dróg z jednego wierzchołka do drugiego, rozwiązanie problemu polega na znalezieniu dowolnej z tych najkrótszych dróg. Oczywistym algorytmem rozwiązania problemu byłoby określenie dla każdego wierzchołka długości wszystkich dróg z niego do innych wierzchołków i znalezienie minimum wśród tych długości. Jednak algorytm taki byłby wykonywany w czasie gorszym od wykładniczego. Na przykład załóżmy, że istnieje krawędź z jednego wierzchołka do wszystkich innych wierzchołków. Ponadto podzbiór wszystkich dróg z tego wierzchołka do innych wierzchołków jest zbiorem tych wszystkich dróg, które rozpoczynają się od pierwszego wierzchołka i kończą na innych wierzchołkach, przechodząc przez wszystkie pozostałe. Z uwagi na fakt, że drugim wierzchołkiem w takiej drodze może być dowolny spośród $n–2$ wierzchołków, trzecim wierzchołkiem w takiej drodze może być dowolny spośród $n–3$ wierzchołków itd., przedostatnim wierzchołkiem w takiej drodze może być tylko jeden wierzchołek, całkowita liczba dróg z jednego wierzchołka do innych wierzchołków, które przechodzą przez wszystkie inne wierzchołki, wynosi:

$$(n-2)(n-3)\ldots 1 = (n-2)!$$

Wynik ten jest gorszy od wykładniczego. Z tą samą sytuacją mamy do czynienia w przypadku wielu problemów optymalizacji, to znaczy oczywisty algorytm rozpatrujący wszystkie możliwości jest wykładniczy lub gorszy. Naszym celem jest znalezienie bardziej wydajnego algorytmu.

Wykorzystując programowanie dynamiczne opracujemy algorytm wykonywany w czasie sześciennym, stanowiący rozwiązanie problemu najkrótszej drogi. Najpierw opracujemy algorytm określający tylko długości najkrótszych dróg. Później zmodyfikujemy go tak, aby dawał na wyjściu również same najkrótsze drogi. Ważony graf skierowany, zawierający n wierzchołków, reprezentujemy za pomocą tablicy W, gdzie:

$$W[i][j] = \begin{cases} \text{waga krawędzi,} & \text{jeżeli istnieje krawędź z } v_i \text{ do } v_j \\ \infty, & \text{jeżeli nie istnieje krawędź z } v_i \text{ do } v_j \\ 0, & \text{jeżeli } i = j \end{cases}$$

O wierzchołku v_j mówimy, że jest ***przyległy*** (ang. *adjacent*) do wierzchołka v_i wówczas, gdy istnieje krawędź z v_i do v_j. Taką tablicę określa się mianem reprezentacji grafu w postaci ***macierzy przyległości*** (ang. *adjacency matrix*). Graf z rysunku 3.2 przedstawiono w tej postaci na rysunku 3.3. Tablica *D* na rysunku 3.3 zawiera długości najkrótszych dróg w grafie. Przykładowo, *D*[3][5] wynosi 7, ponieważ 7 jest długością najkrótszej drogi z v_3 do v_5. Jeżeli będziemy w stanie znaleźć sposób obliczania wartości w *D* na podstawie wartości w *W*, otrzymamy algorytm rozwiązujący problem najkrótszej drogi. Osiągniemy to, tworząc sekwencję $n+1$ tablic $D^{(k)}$, gdzie $0 \le k \le n$ oraz

$D^{(k)}[i][j]$ = długość najkrótszej drogi z v_i do v_j, zawierającej jako wierzchołki pośrednie tylko wierzchołki należące do zbioru $\{v_1, v_2, \ldots v_k\}$.

Rysunek 3.3. *Tablica W reprezentuje graf z rysunku 3.2, zaś tablica D zawiera długości najkrótszych dróg. Opracowywany algorytm rozwiązania problemu najkrótszej drogi oblicza wartości w D na podstawie wartości w W*

W	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

D	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

Zanim wykażemy, że pozwala nam to na obliczenie wartości w *D* na podstawie wartości w *W*, zilustrujemy znaczenie poszczególnych elementów w tych tablicach.

Przykład 3.2. Obliczmy pewne przykładowe wartości w tablicy $D^{(k)}[i][j]$ dla grafu z rysunku 3.2.

$D^{(0)}[2][5] = length\ [v_2, v_5] = \infty$

$D^{(1)}[2][5] = minimum\ (length\ [v_2, v_5], length\ [v_2, v_1, v_5])$
$minimum\ (\infty, 14) = 14$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14$ {Dla dowolnego grafu są one równe, ponieważ}
{najkrótsza droga rozpoczynająca się w v_2}
{nie może przechodzić przez v_2.}

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14$ {Dla tego grafu są one równe, ponieważ}
{uwzględnienie wierzchołka v_3 nie daje żadnej}
{nowej drogi z v_2 do v_5.}

$D^{(4)}[2][5] = minimum\ (length\ [v_2, v_1, v_5], length\ [v_2, v_4, v_5])$
$length\ [v_2, v_1, v_4, v_5], length\ [v_2, v_3, v_4, v_5])$
$minimum\ (14, 5, 13, 10)+5$

$D^{(5)}[2][5] = D^{(4)}[2][5] = 5$ {Dla tego grafu są one równe, ponieważ}
{najkrótsza droga kończąca się w v_5 nie może}
{przechodzić przez v_5.}

Ostatnia obliczona wartość, $D^{(5)}[2][5]$, jest długością najkrótszej drogi z v_2 do v_5, która może przechodzić przez dowolny inny wierzchołek. Oznacza to, że jest ona długością najkrótszej drogi.

$D^{(n)}[i][j]$ jest długością najkrótszej drogi z v_i do v_j, która może przechodzić przez dowolne wierzchołki, więc jest długością najkrótszej drogi z v_i do v_j. $D^{(0)}[i][j]$jest długością najkrótszej drogi, która nie może przechodzić przez inne wierzchołki, więc jest wagą przypisaną do krawędzi, wiodącej od v_i do v_j. Określiliśmy, że

$$D^{(0)} = W \quad \text{oraz} \quad D^{(n)} = D$$

Stąd w celu określenia D na podstawie W musimy jedynie znaleźć sposób otrzymywania wartości $D^{(n)}$ na podstawie $D^{(0)}$. Poniżej opisano poszczególne etapy zmierzających do tego działań, w których wykorzystamy programowanie dynamiczne.

1. *Określamy* właściwość rekurencyjną (proces), dzięki której możemy obliczyć $D^{(k)}$ na podstawie $D^{(k-1)}$.
2. Rozwiązujemy realizację problemu w porządku *wstępującym* poprzez powtarzanie procesu (określonego w etapie 1.) dla $k = 1$ do n. Daje to sekwencję:

$$\begin{array}{ll} D^0, D^1, D^2, \ldots, & Dn \\ \uparrow & \uparrow \\ W & D \end{array} \tag{3.2}$$

Etap 1. wykonujemy, rozpatrując dwa przypadki.

Przypadek 1. Przynajmniej jedna najkrótsza droga z v_i do v_j, zawierająca jako wierzchołki pośrednie tylko wierzchołki ze zbioru $\{v_1, v_2, \ldots, v_k\}$, nie zawiera wierzchołka v_k. Wówczas

$$D^{(k)}[i][j] = D^{(k-1)}[i][j] \tag{3.3}$$

Przykładem tej sytuacji na rysunku 3.2 jest

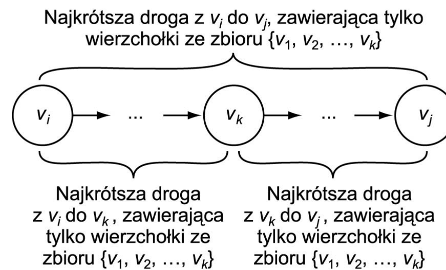
$$D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

ponieważ kiedy uwzględnimy wierzchołek v_5, najkrótsza droga z v_1 do v_3 to wciąż $[v_1, v_4, v_3]$.

Przypadek 2. Wszystkie najkrótsze drogi z v_i do v_j, zawierające jako wierzchołki pośrednie tylko wierzchołki ze zbioru $\{v_1, v_2, \ldots, v_k\}$, zawierają wierzchołek v_k. W takim przypadku dowolna najkrótsza droga ma postać podobną do przedstawionej na rysunku 3.4. v_k nie może być wierzchołkiem pośrednim na poddrodze z v_i do v_k, więc taka poddroga zawiera jako wierzchołki pośrednie jedynie wierzchołki ze zbioru $\{v_1, v_2, \ldots, v_{k-1}\}$. To implikuje, że długość poddrogi musi być równa $D^{(k-1)}[i][k]$ z poniżej podanego powodu. Po pierwsze, długość poddrogi nie może być mniejsza, gdyż $D^{(k-1)}[i][k]$ jest długością najkrótszej drogi z v_i do v_k, zawierającej tylko wierzchołki ze zbioru $\{v_1, v_2, \ldots, v_{k-1}\}$. Po drugie, długość poddrogi nie może być większa, ponieważ gdyby tak było, moglibyśmy ją zastąpić na rysunku 3.4 najkrótszą drogą, co stałoby w sprzeczności z faktem, że cała droga z rysunku 3.4 jest najkrótszą drogą. Podobnie długość poddrogi z v_k do v_j na rysunku 3.4 musi być równa $D^{(k-1)}[k][j]$. Stąd w drugim przypadku:

$$D^{(k)}[i][j] = D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \tag{3.4}$$

Rysunek 3.4. *Najkrótsza droga, zawierająca wierzchołek v_k*



Przykładem drugiego przypadku na rysunku 3.2 jest

$$D^{(2)}[5][3] = 7 = 4 + 3 = D^{(1)}[5][2] + D^{(1)}[2][3]$$

Ponieważ przypadek 1. lub 2. musi występować, wartością $D^{(k)}[i][j]$ jest minimum wartości, znajdujących się po prawej stronie równań 3.3 oraz 3.4. Oznacza to, że możemy określić $D^{(k)}$ na podstawie $D^{(k-1)}$ w następujący sposób:

$$D^{(k)}[i][j] = minimum(\underbrace{D^{(k-1)}[i][j]}_{\text{przypadek 1.}}, \underbrace{D^{(k-1)}[i][k] + D^{(k-1)}[k][j]}_{\text{przypadek 2.}})$$

W ten sposób wykonaliśmy etap 1. opracowywania algorytmu, wykorzystującego programowanie dynamiczne. W celu wykonania etapu 2. wykorzystamy właściwość rekurencyjną z etapu 1. w celu utworzenia sekwencji tablic, przedstawionej w wyrażeniu 3.2. Poniżej przyjrzymy się przykładowi sposobu obliczania wartości każdej z tych tablic na podstawie poprzedniej tablicy.

Przykład 3.3. Jeśli mamy dany graf z rysunku 3.2, reprezentowany przez macierz przyległości W na rysunku 3.3, niektóre obliczenia wykonuje się w następujący sposób (należy pamiętać, że $D^{(0)} = W$):

$$\begin{aligned}
D^{(1)}[2][4] &= minimum(D^{(0)}[2][4], D^{(0)}[2][1] + D^{(0)}[1][4]) \\
&= minimum(2, 9+1) = 2 \\
D^{(1)}[5][2] &= minimum(D^{(0)}[5][2], D^{(0)}[5][1] + D^{(0)}[1][2]) \\
&= minimum(\infty, 3+1) = 4 \\
D^{(1)}[5][4] &= minimum(D^{(0)}[5][4], D^{(0)}[5][1] + D^{(0)}[1][4]) \\
&= minimum(\infty, 3+1) = 4
\end{aligned}$$

Kiedy cała tablica $D^{(1)}$ zostanie wyliczona, obliczamy tablicę $D^{(2)}$. Przykładowe obliczenie ma postać:

$$\begin{aligned}
D^{(2)}[5][4] &= minimum(D^{(1)}[5][4], D^{(1)}[5][2] + D^{(1)}[2][4]) \\
&= minimum(4, 4+2) = 4
\end{aligned}$$

Po obliczeniu wszystkich wyrazów w tablicy $D^{(2)}$ kontynuujemy działania po kolei do momentu wyliczenia $D^{(5)}$. Końcową tablicą jest D, która zawiera długości najkrótszych dróg. Na rysunku 3.3 umieszczono ją po prawej stronie.

Poniżej przedstawimy algorytm opracowany przez Floyda (1962), znany jako *algorytm Floyda* (ang. *Floyd's algorithm*). Później wyjaśnimy, dlaczego korzysta on tylko z jednej tablicy D oprócz tablicy wejściowej W.

Algorytm 3.3. Algorytm Floyda określania najkrótszej drogi

Problem: oblicz najkrótsze drogi z każdego wierzchołka w grafie ważonym do wszystkich innych wierzchołków. Wagi są liczbami nieujemnymi.

Dane wejściowe: ważony graf skierowany oraz n, liczba wierzchołków w grafie. Graf jest reprezentowany przez tablicę dwuwymiarową W, której wiersze i kolumny są indeksowane od 1 do n, gdzie $W[i][j]$ jest wagą krawędzi, prowadzącej od i-tego do j-tego wierzchołka.

Dane wyjściowe: dwuwymiarowa tablica D, której wiersze i kolumny są indeksowane od 1 do n, gdzie $D[i][j]$ jest długością najkrótszej drogi, prowadzącej od i-tego do j-tego wierzchołka.

```
void floyd (int n,
            const number W[][],
                  number D[][])
{
   index i, j, k;

   D = W;
   for (k = 1; k <= n; k++)
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            D[i][j] = minimum(D[i][j], D[i][k]+D[k][j]);
}
```

Możemy wykonać nasze obliczenia przy użyciu tylko jednej tablicy D, ponieważ wartości w k-tym wierszu oraz k-tej kolumnie nie są wymieniane w czasie k-tego przebiegu pętli. Oznacza to, że w k-tym przebiegu algorytm dokonuje przypisania:

$$D[i][k] = minimum(D[i][k], D[i][k] + D[k][k])$$

co jest oczywiście równe $D[i][k]$ oraz

$$D[k][j] = minimum(D[k][j], D[k][k] + D[k][j])$$

co jest oczywiście równe $D[k][j]$. W czasie k-tego przebiegu element $D[i][j]$ jest obliczany tylko na podstawie własnej wartości oraz wartości w k-tym wierszu i k-tej kolumnie. Wartości te zostały przypisane w (k–1). przebiegu, więc są poszukiwanymi przez nas wartościami. Jak wspomniano wcześniej, czasem po opracowaniu algorytmu programowania dynamicznego istnieje możliwość jego poprawienia, tak aby był bardziej wydajny pod względem zajętości pamięci.

Poniżej zostanie przedstawiona analiza algorytmu Floyda.

Analiza algorytmu 3.3.

Złożoność czasowa w każdym przypadku (algorytm Floyda na najkrótszą drogę)

Operacja podstawowa: instrukcja w pętli `for`-j.

Rozmiar danych wejściowych: n, liczba wierzchołków w grafie.

Mamy do czynienia z pętlą w pętli w pętli, z których każda jest związana z wykonaniem n przebiegów, tak więc:

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3).$$

Poniższa modyfikacja algorytmu 3.3 pozwala na obliczenie najkrótszej drogi.

Algorytm 3.4. Algorytm Floyda określania najkrótszej drogi 2

Problem: podobnie jak w przypadku algorytmu 3.3. Oprócz tego tworzone są również najkrótsze drogi.

Dodatkowe dane wyjściowe: tablica P, której wiersze i kolumny są indeksowane od 1 do n, gdzie

$$P(i, j) = \begin{cases} \text{najwyższy indeks wierzchołka pośredniego w najkrótszej drodze} \\ \text{od } v_i \text{ do } v_j\text{, jeżeli istnieje co najmniej jeden wierzchołek pośredni} \\ 0\text{, jeżeli nie istnieje żaden wierzchołek pośredni} \end{cases}$$

```
void floyd2 (int n,
            const number W[][],
                  number D[][],
                  index  P[][])
```

```
{
   index i, j, k;

   for (i = 1; i <= n; i++)
      for (i = 1; i <= n; i++)
         P[i][j] = 0;
   D = W;
   for (k = 1; k <= n; k++)
      for (i = 1; i <= n; i++)
         for (j = 1; j <= n; j++)
            if (D[i][k]+D[k][j] < D[i][j]) {
               P[i][j] = k;
               D[i][j]+D[i][k]+D[k][j]);
            }
}
```

Na rysunku 3.5 przedstawiono tablicę *P*, która jest tworzona w przypadku zastosowania algorytmu dla grafu z rysunku 3.2.

Rysunek 3.5.
Tablica P, utworzona w przypadku zastosowania algorytmu 3.4 dla grafu z rysunku 3.2

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

Poniższy algorytm tworzy najkrótszą drogę od wierzchołka v_q do v_r na podstawie tablicy *P*.

Algorytm 3.5. Wyświetlenie najkrótszej drogi

Problem: wyświetl wierzchołki pośrednie w najkrótszej drodze od jednego wierzchołka do innego w grafie ważonym.

Dane wejściowe: tablica *P*, utworzona przez algorytm 3.4 oraz dwa indeksy (q i r) wierzchołków w grafie, który stanowi dane wejściowe algorytmu 3.4.

$$P(i,j) = \begin{cases} \text{najwyższy indeks wierzchołka pośredniego w najkrótszej drodze} \\ \text{od } v_i \text{ do } v_j \text{, jeżeli istnieje co najmniej jeden wierzchołek pośredni} \\ 0\text{, jeżeli nie istnieje żaden wierzchołek pośredni} \end{cases}$$

Dane wyjściowe: wierzchołki pośrednie w najkrótszej drodze z v_q do v_r.

```
void path (index q, r)
{
   if (P[q][r] != 0) {
      path(q, P[q][r]);
      cout << "v" << P[q][r];
```

```
      path(P[q][r], r);
   }
}
```

Warto przypomnieć konwencję określoną w rozdziale 2., zgodnie z którą danymi wejściowymi podprogramów rekurencyjnych mogą być tylko takie zmienne, których wartość może ulegać zmianie w wywołaniach rekurencyjnych. Stąd tablica *P* nie stanowi danych wejściowych procedury *path*. Gdyby algorytm zaimplementowano definiując *P* globalnie i chcielibyśmy określić najkrótszą drogę z wierzchołka v_q do wierzchołka v_r, to wywołanie procedury *path* na najwyższym poziomie miałoby postać:

```
path(q, r);
```

Dla danej wartości *P* z rysunku 3.5, jeżeli wartości *q* i *r* wynosiłyby, odpowiednio, 5 i 3, dane wyjściowe miałyby postać:

$$v1 \quad v4$$

Są to wierzchołki pośrednie w najkrótszej drodze z wierzchołka v_5 do wierzchołka v_3.

W ćwiczeniach zostanie określone, że $W(n) \in \Theta(n)$ w przypadku algorytmu 3.5.

3.3. Programowanie dynamiczne a problemy optymalizacyjne

Należy przypomnieć, że algorytm 3.4 nie tylko pozwala określić długości najkrótszych dróg, ale również konstruuje najkrótsze drogi. Konstrukcja optymalnego rozwiązania jest trzecim etapem w procesie opracowywania algorytmu programowania dynamicznego dla problemu optymalizacji. Oznacza to, że w procesie opracowywania takiego algorytmu można wyróżnić następujące etapy:

1. *Określenie* właściwości rekurencyjnej, która daje rozwiązanie optymalne dla realizacji problemu.
2. Obliczenie wartości rozwiązania optymalnego w porządku *wstępującym*.
3. Skonstruowanie rozwiązania optymalnego w porządku *wstępującym*.

Etapy 2. oraz 3. są zwykle wykonywane mniej więcej w tym samym miejscu algorytmu. Ze względu na fakt, że algorytm 3.2 nie jest problemem optymalizacji, nie występuje w nim trzeci etap.

Choć może się wydawać, że problem optymalizacji może zawsze zostać rozwiązany przy użyciu programowania dynamicznego, nie jest to prawdą. Aby tak było, w przypadku danego problemu musi mieć zastosowanie zasada optymalności. Zasadę tę można wyrazić następująco:

Definicja

O ***zasadzie optymalności*** (ang. *principle of optimality*) mówi się, że ma zastosowanie w problemie wówczas, gdy rozwiązanie optymalne realizacji problemu zawsze zawiera rozwiązania optymalne dla wszystkich podrealizacji.

Zasada optymalności jest trudna do jednoznacznego zdefiniowania i łatwiej jest ją zrozumieć poprzez analizę konkretnego przykładu. W przypadku problemu najkrótszej drogi pokazaliśmy, że jeżeli v_k jest wierzchołkiem należącym do drogi optymalnej z v_i do v_j, to poddrogi z v_i do v_k oraz z v_k do v_j również muszą być optymalne. Stąd optymalne rozwiązanie realizacji zawiera rozwiązania optymalne wszystkich podrealizacji i ma tu zastosowanie zasada optymalności.

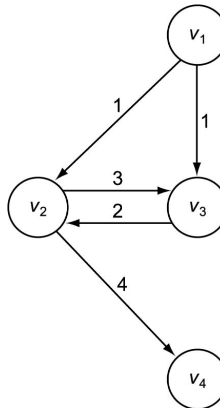
Jeżeli zasada optymalności ma zastosowanie w przypadku danego problemu, możemy określić właściwość rekurencyjną, która będzie dawać optymalne rozwiązanie realizacji w kontekście optymalnych rozwiązań podrealizacji. Istotnym, choć subtelnym powodem, dla którego możemy wówczas wykorzystać programowanie dynamiczne w celu skonstruowania optymalnego rozwiązania realizacji, jest to, że optymalne rozwiązania podrealizacji mogą być dowolnymi optymalnymi rozwiązaniami. Przykładowo, w przypadku problemu najkrótszej drogi: jeżeli poddrogami są dowolne najkrótsze drogi, połączona droga będzie optymalna. Możemy więc wykorzystać właściwość rekurencyjną w celu skonstruowania rozwiązań optymalnych coraz większych realizacji w porządku wstępującym. Każde tak otrzymane rozwiązanie jest optymalne.

Choć zasada optymalności może wydawać się oczywista, w praktyce konieczne jest wykazanie, że zasada ma zastosowanie, zanim jeszcze założy się, że rozwiązanie optymalne może zostać otrzymane dzięki programowaniu dynamicznemu. Poniższy przykład pokazuje, że nie ma ona zastosowania w przypadku każdego problemu optymalizacji.

Przykład 3.4. Rozważmy problem najdłuższej drogi, polegający na znalezieniu najdłuższych prostych dróg wiodących od każdego wierzchołka do wszystkich innych wierzchołków. Problem ograniczamy do prostych dróg, ponieważ w przypadku cykli zawsze możemy utworzyć dowolnie długą drogę poprzez powtarzalne przechodzenie przez cykl. Na rysunku 3.6 optymalna (najdłuższa) prosta droga z v_1 do v_4 to $[v_1, v_3, v_2, v_4]$. Jednak poddroga $[v_1, v_3]$ nie jest optymalną (najdłuższą) drogą z v_1 do v_3, ponieważ

$$\textit{długość}[v_1, v_3] = 1 \quad \text{oraz} \quad \textit{długość}[v_1, v_2, v_3] = 4$$

Zatem zasada optymalności nie ma zastosowania. Wynika to z faktu, że optymalne drogi z v_1 do v_3 oraz z v_3 do v_4 nie mogą zostać razem powiązane, aby utworzyć optymalną drogę z v_1 do v_4. Spowodowałoby to utworzenie cyklu, a nie optymalnej drogi.



Rysunek 3.6.
Ważony graf skierowany z cyklem

Pozostałą część rozdziału poświęcimy problemom optymalizacji. Opracowując algorytmy nie będziemy wymieniać wcześniej opisanych etapów działania. Powinno być rzeczą oczywistą, że postępujemy zgodnie z nimi.

3.4. Łańcuchowe mnożenie macierzy

Załóżmy, że chcemy pomnożyć macierz o wymiarach 2×3 przez macierz o rozmiarach 3×4 w sposób następujący:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

Macierz wynikowa ma rozmiary 2×4. Jeżeli użyjemy standardowej metody mnożenia macierzy (opartej na definicji mnożenia macierzy), obliczenie każdego elementu iloczynu wymagać będzie wykonania trzech podstawowych operacji mnożenia. Przykładowo, pierwszy element w pierwszej kolumnie to:

$$\underbrace{1 \times 7 + 2 \times 2 + 3 \times 6}_{\text{3 mnożenia}}$$

Ze względu na fakt, że w iloczynie występuje 2×4 = 8 pozycji, całkowita liczba elementarnych operacji mnożenia wynosi

$$2 \times 4 \times 3 = 24$$

Ogólnie rzecz biorąc, w celu pomnożenia macierzy o wymiarach $i \times j$ przez macierz o wymiarach $j \times k$ przy użyciu metody standardowej musimy wykonać

$$i \times j \times k \text{ elementarnych operacji mnożenia}$$

Rozważmy operację mnożenia następujących macierzy:

$$\begin{matrix} A & \times & B & \times & C & \times & D \\ 20\times 2 & & 2\times 30 & & 30\times 12 & & 12\times 8 \end{matrix}$$

Wymiary każdej macierzy zapisano pod nimi. Mnożenie macierzy jest operacją łączną, co oznacza, że kolejność, w jakiej wykonujemy mnożenie, nie ma znaczenia. Przykładowo, operacje $A(B(CD))$ oraz $(AB)(CD)$ dają ten sam wynik. Istnieje pięć różnych kolejności, w jakich możemy pomnożyć cztery macierze i które dadzą zwykle różną liczbę elementarnych operacji mnożenia. W przypadku powyższych macierzy mamy następujące liczby elementarnych operacji mnożenia dla różnych kolejności działań:

$$\begin{array}{ll} A(B(CD)) & 30\times12\times8+2\times30\times8+20\times2\times8 = 3680 \\ (AB)(CD) & 20\times2\times30+30\times12\times8+20\times30\times8 = 8880 \\ A((BC)D) & 2\times30\times12+2\times12\times8+20\times2\times8 = 1232 \\ ((AB)C)D & 20\times2\times30+20\times30\times12+20\times12\times8 = 10320 \\ (A(BC))D & 2\times30\times12+20\times2\times12+20\times12\times8 = 3120 \end{array}$$

Trzecia kolejność jest optymalna w przypadku mnożenia czterech macierzy.

Naszym celem jest opracowanie algorytmu, który będzie określał optymalną kolejność mnożenia n macierzy. Kolejność ta zależy tylko od rozmiarów macierzy. Stąd oprócz n rozmiary te stanowią jedyne dane wejściowe dla algorytmu. Algorytm wykorzystujący metodę siłową polegałby na rozważeniu wszystkich możliwych kolejności i wybraniu minimum — tak jak postąpiliśmy powyżej. Wykażemy, że taki algorytm jest wykonywany co najmniej w czasie wykładniczym. Niech t_n będzie liczbą różnych kolejności, w jakich możemy pomnożyć n macierzy: $A_1, A_2, \ldots, A_n$. Jednym z podzbiorów wszystkich kolejności jest zbiór kolejności, dla których macierz A_1 jest ostatnią mnożoną macierzą. Jak pokazano poniżej, liczba różnych kolejności w tym podzbiorze wynosi t_{n-1}, ponieważ jest to liczba kolejności, w jakich możemy pomnożyć macierze od A_2 do A_n:

$$A_1(\underbrace{A_2 A_3 \ldots A_n}_{\substack{t_{n-1}\text{ różnych} \\ \text{kolejności}}})$$

Drugi podzbiór wszystkich kolejności jest zbiorem kolejności, w przypadku których macierz A_n jest ostatnią mnożoną macierzą. Oczywiście, liczba różnych kolejności w tym podzbiorze również wynosi t_{n-1}. Stąd:

$$t_n \geq t_{n-1} + t_{n-1} = 2t_{n-1}$$

Ze względu na fakt, że istnieje tylko jeden sposób pomnożenia dwóch macierzy, $t_2 = 1$. Wykorzystując techniki opisane w dodatku B możemy rozwiązać tę rekurencję, wykazując, że:

$$t_n \geq 2^{n-2}$$

Nietrudno zauważyć, że zasada optymalności ma zastosowanie w przypadku tego problemu. Oznacza to, że optymalna kolejność mnożenia n macierzy zawiera optymalną kolejność mnożenia dowolnego podzbioru zbioru n macierzy. Przykładowo, jeżeli optymalna kolejność mnożenia sześciu macierzy ma postać

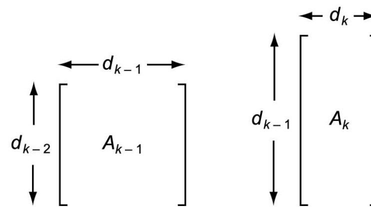
$$A_1((((A_2 A_3)A_4)A_5)A_6)$$

to

$$(A_2 A_3)A_4$$

musi być optymalną kolejnością mnożenia macierzy od A_2 do A_4. Oznacza to, że w celu skonstruowania rozwiązania możemy wykorzystać programowanie dynamiczne.

Ze względu na fakt, że mnożymy $(k-1)$-tą macierz, A_{k-1}, przez k-tą macierz, A_k, liczba kolumn w A_{k-1} musi być równa liczbie wierszy w A_k. Na przykład w przypadku wcześniej omówionego iloczynu pierwsza macierz ma trzy kolumny, zaś druga macierz — trzy wiersze. Jeżeli przyjmiemy, że d_0 jest liczbą wierszy w A_1, zaś d_k jest liczbą kolumn w A_k dla $1 \leq k \leq n$, to wymiary A_k będą wynosić $d_{k-1} \times d_k$. Zilustrowano to na rysunku 3.7.



Rysunek 3.7. *Liczba kolumn w macierzy A_{k-1} jest taka sama, jak liczba wierszy w macierzy A_k*

Tak, jak w poprzednim podrozdziale, w celu skonstruowania rozwiązania użyjemy sekwencji tablic. Dla $1 \leq i \leq j \leq n$ niech

$M[i][j]$ = minimalna liczba mnożeń wymaganych do pomnożenia macierzy od A_i do A_j, jeżeli $i < j$

$M[i][i]$ = 0

Zanim omówimy sposób użycia tych tablic, poniżej zilustrujemy znaczenie zawartych w nich elementów.

Przykład 3.5. Załóżmy, że posiadamy następujące sześć macierzy:

$$\begin{array}{ccccccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\ 5\times 2 & & 2\times 3 & & 3\times 4 & & 4\times 6 & & 6\times 7 & & 7\times 8 \\ d_0\ d_1 & & d_1\ d_2 & & d_2\ d_3 & & d_3\ d_4 & & d_4\ d_5 & & d_5\ d_6 \end{array}$$

W celu pomnożenia macierzy A_4, A_5 i A_6 możemy określić poniższe dwie kolejności oraz liczby elementarnych operacji mnożenia:

$(A_4A_5)A_6$ Liczba operacji mnożenia $= d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6$
$= 4 \times 6 \times 7 + 4 \times 7 \times 8 = 392$

$A_4(A_5A_6)$ Liczba operacji mnożenia $= d_4 \times d_5 \times d_6 + d_3 \times d_4 \times d_6$
$= 6 \times 7 \times 8 + 4 \times 6 \times 8 = 528$

Stąd

$$M[4][6] = minimum(392, 528) = 392$$

Optymalna kolejność mnożenia sześciu macierzy musi mieć jeden z następujących rozkładów:

1. $A_1(A_2A_3A_4A_5A_6)$
2. $(A_1A_2)(A_3A_4A_5A_6)$
3. $(A_1A_2A_3)(A_4A_5A_6)$
4. $(A_1A_2A_3A_4)(A_5A_6)$
5. $(A_1A_2A_3A_4A_5)A_6$

gdzie w każdym nawiasie iloczyn jest otrzymywany zgodnie z optymalną kolejnością dla liczby macierzy, znajdujących się w tym nawiasie. Spośród rozkładów ten, który daje minimalną liczbę operacji mnożenia, musi być optymalny. Liczba operacji mnożenia dla k-tego rozkładu jest minimalną liczbą potrzebną do otrzymania każdego czynnika, powiększoną o liczbę potrzebną do pomnożenia dwóch czynników. Oznacza to, że wynosi ona

$$M[1][k] + M[k+1][6] + d_0 d_k d_6$$

Określiliśmy, że

$$M[1][6] = \underbrace{minimum}_{1 \le k \le 5}(M[1][k] + M[k+1][6] + d_0 d_k d_6)$$

W powyższym rozumowaniu nie ma nic, co wymuszałoby, aby pierwszą macierzą była A_1 lub aby ostatnią macierzą była A_6. Przykładowo, moglibyśmy otrzymać podobny rezultat mnożąc macierz A_2 przez A_6. Zatem możemy uogólnić ten rezultat w celu otrzymania następującej właściwości rekurencyjnej, związanej z mnożeniem n macierzy. Dla $1 \le i \le j \le n$

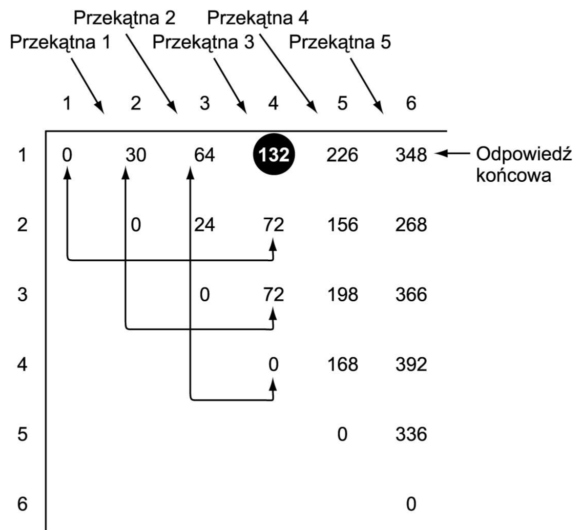
$$M[i][j] = \underbrace{minimum}_{i \le k \le j-1}(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j), \text{ jeżeli } i < j \quad (3.5)$$

$$M[i][i] = 0$$

Algorytm typu *dziel i zwyciężaj* oparty na tej właściwości jest wykonywany w czasie wykładniczym. Poniżej opracujemy wydajniejszy algorytm, wykorzystując programowanie dynamiczne w celu obliczenia wartości $M[i][j]$ w kolejnych etapach.

Używana jest siatka podobna do trójkąta Pascala (patrz podrozdział 3.1). Obliczenia, które są nieco bardziej skomplikowane, niż miało to miejsce w podrozdziale 3.1, są oparte na poniżej opisanej właściwości z równania 3.5. Element $M[i][j]$ jest obliczany na podstawie wszystkich wpisów ze swojego wiersza, znajdujących się po jego lewej stronie, oraz wpisów ze swojej kolumny, znajdujących się poniżej niego. Wykorzystując tę właściwość można obliczyć wartości elementów w M w poniżej opisany sposób. Najpierw ustawiamy wartość tych elementów na głównej przekątnej na 0. Następnie obliczamy wszystkie elementy na przekątnej powyżej (nazywamy ją przekątną 1). Następnie obliczamy wszystkie wartości na przekątnej 2 itd. Kontynuujemy te działania do momentu, aż obliczymy jedyną wartość na przekątnej 5, która jest naszą odpowiedzią końcową, $M[1][6]$. Procedurę tę zilustrowano na rysunku 3.8 dla macierzy z przykładu 3.5. Poniższy przykład zawiera odpowiednie obliczenia.

Rysunek 3.8.
Tablica M opracowana w przykładzie 3.5. Element M[1][4], który oznaczono kółkiem, jest obliczany na podstawie par wskazanych wartości



Przykład 3.6. Załóżmy, że mamy sześć macierzy określonych w przykładzie 3.5. Poniżej opiszemy kolejne działania wykonywane przez algorytm programowania dynamicznego. Odpowiednie wyniki przedstawiono na rysunku 3.8.

Obliczamy przekątną 0:

$$M[i][i] = 0 \quad \text{dla}\, 1 \le i \le 6$$

Obliczamy przekątną 1:

$$\begin{aligned} M[1][2] &= \underbrace{minimum}_{1 \le k \le 1}(M[1][k] + M[k+1][2] + d_0 d_k d_2) \\ &= M[1][1] + M[2][2] + d_0 d_1 d_2 \\ &= 0 + 0 + 5 \times 2 \times 3 = 30 \end{aligned}$$

Wartości $M[2][3]$, $M[3][4]$, $M[4][5]$ oraz $M[5][6]$ są obliczane w ten sam sposób. Przedstawiono je na rysunku 3.8.

Obliczamy przekątną 2:

$$M[1][3] = \underbrace{minimum}_{1\le k\le 2}(M[1][k] + M[k+1][3] + d_0 d_k d_3)$$
$$= minimum(M[1][1] + M[2][3] + d_0 d_1 d_3,$$
$$M[1][2] + M[3][3] + d_0 d_2 d_3$$
$$= minimum(0 + 24 + 5\times 2\times 4, 30 + 0 + 5\times 3\times 4) = 64$$

Wartości $M[2][4]$, $M[3][5]$ oraz $M[4][6]$ są obliczane w ten sam sposób. Przedstawiono je na rysunku 3.8.

Obliczamy przekątną 3:

$$M[1][4] = \underbrace{minimum}_{1\le k\le 3}(M[1][k] + M[k+1][4] + d_0 d_k d_4)$$
$$= minimum(M[1][1] + M[2][4] + d_0 d_1 d_4,$$
$$M[1][2] + M[3][4] + d_0 d_2 d_4,$$
$$M[1][3] + M[4][4] + d_0 d_3 d_4)$$
$$= minimum(0 + 72 + 5\times 2\times 6, 30 + 72 + 5\times 3\times 6, 64 + 0 + 5\times 4\times 6) = 132$$

Wartości $M[2][5]$ oraz $M[3][6]$ są obliczane w ten sam sposób. Przedstawiono je na rysunku 3.8.

Obliczamy przekątną 4:

Wartości na przekątnej 4 są obliczane w ten sam sposób i przedstawiono je na rysunku 3.8.

Obliczamy przekątną 5:

Wartość na przekątnej 5 jest obliczana w ten sam sposób. Jest ona rozwiązaniem realizacji problemu: jest to minimalna liczba elementarnych operacji mnożenia i wynosi ona:

$$M[1][6] = 348$$

Przedstawiony poniżej algorytm stanowi implementację tej metody. Wymiary n macierzy, a konkretnie wartości od d_0 do d_n, są jedynymi danymi wejściowymi algorytmu. Same macierze nie stanowią danych wejściowych, ponieważ wartości w macierzach nie mają znaczenia dla istoty problemu. Tablica P utworzona przez algorytm może zostać wykorzystana do wydrukowania optymalnej kolejności. Zostanie to omówione po przeanalizowaniu algorytmu 3.6.

Algorytm 3.6. Minimalna liczba operacji mnożenia

Problem: określ minimalną liczbę elementarnych operacji mnożenia, wymaganych w celu pomnożenia *n* macierzy oraz kolejność wykonywanych mnożeń, która zapewnia minimalną liczbę operacji.

Dane wejściowe: liczba macierzy *n* oraz tablica liczb całkowitych *d*, indeksowana od 0 do *n*, gdzie $d[i-1]\times d[i]$ jest rozmiarem *i*-tej macierzy.

Dane wyjściowe: *minmult*, minimalna liczba elementarnych operacji mnożenia, wymaganych w celu pomnożenia *n* macierzy; dwuwymiarowa tablica *P*, na podstawie której można określić optymalną kolejność. *P* posiada wiersze indeksowane od 1 do $n-1$ oraz kolumny indeksowane od 1 do *n*. $P[i][j]$ jest punktem, w którym macierze od *i* do *j* zostają rozdzielone w kolejności optymalnej dla mnożenia macierzy.

```
int minmult (int n,
             const int d[],
             index P[][])
{
   index i, j, k, diagonal;
   int M[1..n][1..n];

   for (i = 1; i <= n; i++)
      M[i][i] = 0;
   for (diagonal = 1; diagonal <= n-1; diagonal++) // Przekątna 1 znajduje
      for (i = 1; i <= n - diagonal; i++) {        // się tuż nad główną
         j = i+diagonal;                           // przekątną
         M[i][j] = minimum (M[i][k]+M[k+1][j] +d[i - 1]*d[k]*d[j]);
                  i ≤ k ≤ j-1

         P[i][j] = wartość k, która dała minimum;
      }
   return M[1][n];
}
```

Poniżej dokonamy analizy algorytmu 3.6.

Analiza algorytmu 3.6.

Złożoność czasowa w każdym przypadku (minimalna liczba operacji mnożenia)

Operacja podstawowa: jako operację podstawową możemy traktować instrukcje, wykonywane dla każdej wartości *k*. Uwzględniono również porównanie sprawdzające, czy wartość jest minimalna.

Rozmiar danych wejściowych: *n*, liczba macierzy, które mają zostać pomnożone.

Mamy do czynienia z pętlą w pętli w pętli. Ze względu na fakt, że $j = i+diagonal$, dla danych wartości *diagonal* oraz *i* liczba przebiegów pętli *k* wynosi

$$j-1-i+1 = i+diagonal-1-i+1 = diagonal$$

Dla danej wartości *diagonal* liczba przebiegów pętli `for`-*i* wynosi $n-diagonal$. Ze względu na fakt, że *diagonal* może przyjmować wartości od 1 do $n-1$, całkowita liczba powtórzeń operacji podstawowej wynosi

$$\sum_{diagonal=1}^{n-1}[(n - diagonal) \times diagonal]$$

W ćwiczeniach zostanie wykazane, że wyrażenie to jest równe:

$$\frac{n(n-1)(n+1)}{6} \in \Theta(n^3)$$

Poniżej pokażemy, w jaki sposób można otrzymać optymalną kolejność na podstawie tablicy P. Wartości zawarte w tej tablicy w momencie, gdy algorytm jest stosowany dla wymiarów z przykładu 3.5, przedstawiono na rysunku 3.9. Fakt, że na przykład $P[2][5] = 4$ oznacza, że optymalna kolejność dla mnożenia macierzy od A_2 do A_5 posiada rozkład:

$$(A_2 A_3 A_4) A_5$$

gdzie macierze znajdujące się w nawiasie są mnożone w kolejności optymalnej. Oznacza to, że $P[2][5]$, czyli 4, jest punktem, w którym macierze powinny zostać rozdzielone w celu otrzymania czynników. Możemy określić optymalną kolejność, odwiedzając najpierw element $P[1][n]$ w celu określenia rozkładu na najwyższym poziomie. Ze względu na fakt, że $n = 6$ oraz $P[1][6] = 1$, rozkład na najwyższym poziomie dla kolejności optymalnej ma postać:

$$A_1 (A_2 A_3 A_4 A_5 A_6)$$

Rysunek 3.9. *Tablica P, utworzona w momencie, gdy algorytm 3.6 został zastosowany względem wymiarów z przykładu 3.5*

	1	2	3	4	5	6
1		1	1	1	1	1
2			2	3	4	5
3				3	4	5
4					4	5
5						5

Następnie określamy rozkład w kolejności optymalnej dla mnożenia macierzy od A_2 do A_6, odwiedzając element $P[2][6]$. Jego wartością jest 5, więc rozkład ma postać:

$$(A_2 A_3 A_4 A_5) A_6$$

Wiemy, że rozkład w kolejności optymalnej ma postać:

$$A_1 ((A_2 A_3 A_4 A_5) A_6)$$

gdzie rozkład dla mnożenia macierzy od A_2 do A_5 wciąż należy określić. Sprawdzamy wartość elementu $P[2][5]$ i kontynuujemy w ten sposób dalsze działania do momentu, w którym zostaną określone wszystkie rozkłady. Odpowiedź ma postać:

$$A_1 (((A_2 A_3) A_4) A_5) A_6)$$

Poniższy algorytm stanowi implementację opisanej metody.

Algorytm 3.7. Wyświetlanie optymalnej kolejności

Problem: wyświetl optymalną kolejność dla mnożenia *n* macierzy.

Dane wejściowe: dodatnia liczba całkowita *n* oraz tablica *P*, utworzona przez algorytm 3.6. *P*[*i*][*j*] jest punktem, w którym macierze od *i* do *j* są rozdzielane w kolejności optymalnej dla mnożenia tych macierzy.

Dane wyjściowe: optymalna kolejność mnożenia macierzy.

```
void order (index i, index j)
{
   if (i == j)
      cout << "A" << i;
   else {
      k = P[i][j];
      cout << "(";
      order(i, k);
      order(k+1, j);
      cout << ")";
   }
}
```

Zgodnie z konwencjami przyjętymi odnośnie pisania funkcji rekurencyjnych, *P* i *n* nie są danymi wejściowymi dla procedury *order*, ale są danymi wejściowymi algorytmu. Gdyby algorytm zaimplementowano poprzez zdefiniowanie *P* i *n* globalnie, wywołanie procedury *order* na najwyższym poziomie miałoby postać:

```
order(1, n);
```

Kiedy wymiary są takie same, jak w przykładzie 3.5, algorytm wyświetla następujące informacje:

$$(A1((((A2A3)A4)A5)A6))$$

Całe wyrażenie ujęto w nawiasy, ponieważ algorytm wstawia je wokół każdego składnika złożonego. W ćwiczeniach zostanie wykazane, że dla algorytmu 3.7

$$T(n) \in \Theta(n)$$

Opracowany algorytm $\Theta(n^3)$ dla łańcuchowego mnożenia macierzy pochodzi z pracy Godbole'a (1973). Yao (1982) opracował metody przyspieszające pewne rozwiązania programowania dynamicznego. Wykorzystując te metody można utworzyć algorytm $\Theta(n^2)$ dla łańcuchowego mnożenia macierzy. Hu oraz Shing (1982, 1984) opisują algorytm $\Theta(n \lg n)$ dla łańcuchowego mnożenia macierzy.

3.5. Optymalne drzewa wyszukiwania binarnego

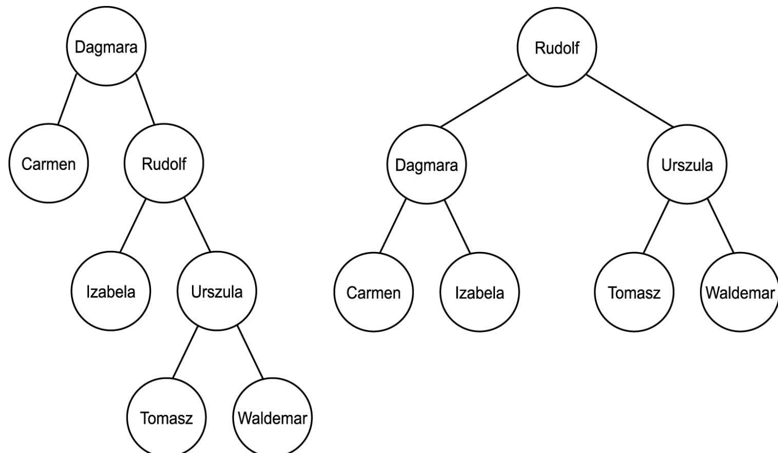
Kolejny algorytm, jaki opracujemy, służy do określania optymalnego sposobu zorganizowania zbioru elementów w postaci drzewa wyszukiwania binarnego. Zanim omówimy, jaka forma organizacji jest uważana za optymalną, przedstawimy pewne ogólne informacje na temat takich drzew. Dla każdego wierzchołka w drzewie binarnym poddrzewo, którego korzeniem jest lewy potomek tego wierzchołka, nosi nazwę ***lewego poddrzewa*** (ang. *left subtree*) wierzchołka. Lewe poddrzewo korzenia drzewa nosi nazwę lewego poddrzewa drzewa. Analogicznie definiuje się ***prawe poddrzewo*** (ang. *right subtree*).

Definicja

Drzewo wyszukiwania binarnego (ang. *binary search tree*) jest binarnym drzewem elementów (zwykle nazywanych kluczami) pochodzących ze zbioru uporządkowanego. Musi spełniać następujące warunki:

1. Każdy wierzchołek zawiera jeden klucz.
2. Każdy klucz w lewym poddrzewie danego wierzchołka jest mniejszy lub równy kluczowi tego wierzchołka.
3. Klucze znajdujące się w prawym poddrzewie danego wierzchołka są większe lub równe kluczowi tego wierzchołka.

Na rysunku 3.10 przedstawiono dwa drzewa wyszukiwania binarnego, każde z tymi samymi kluczami. W drzewie po lewej stronie przyjrzyjmy się prawemu poddrzewu wierzchołka, zawierającego klucz *Rudolf*. Poddrzewo to zawiera klucze *Tomasz*, *Urszula* oraz *Waldemar* i wszystkie te imiona są „większe” od *Rudolfa*, zgodnie z porządkiem alfabetycznym. Choć ogólnie klucz może występować w drzewie binarnym więcej niż raz, dla uproszczenia zakładamy, że klucze są unikatowe.



Rysunek 3.10. *Dwa drzewa przeszukiwania binarnego*

Głębokość (ang. *depth*) wierzchołka w drzewie jest liczbą krawędzi w unikatowej drodze, wiodącej od korzenia do tego wierzchołka. Jest ona również zwana ***poziomem*** (ang. *level*) wierzchołka w drzewie. Zwykle mówimy, że wierzchołek *posiada* głębokość oraz że znajduje się *na* poziomie. Przykładowo, w drzewie znajdującym się po lewej stronie na rysunku 3.10 wierzchołek zawierający klucz *Urszula* posiada głębokość 2. Możemy również stwierdzić, że znajduje się on na poziomie 2. Korzeń posiada głębokość 0 i znajduje się na poziomie 0. ***Głębokość*** drzewa to maksymalna głębokość wszystkich wierzchołków. Drzewo znajdujące się po lewej stronie na rysunku 3.10 posiada głębokość 3, natomiast drzewo znajdujące się po prawej stronie posiada głębokość 2. Drzewo binarne jest nazywane ***zrównoważonym*** (ang. *balanced*), jeżeli głębokość dwóch poddrzew każdego wierzchołka nigdy nie różni się o więcej niż 1. Drzewo znajdujące się po lewej stronie na rysunku 3.10 nie jest zrównoważone, ponieważ lewe poddrzewo korzenia posiada głębokość 0, natomiast prawe poddrzewo posiada głębokość 2. Drzewo znajdujące się po prawej stronie jest zrównoważone.

Zazwyczaj drzewo wyszukiwania binarnego zawiera pozycje, które są pobierane zgodnie z wartościami kluczy. Naszym celem jest takie zorganizowanie kluczy w drzewie wyszukiwania binarnego, aby średni czas zlokalizowania klucza był minimalny (patrz podrozdział A.8.2, gdzie omówiono problem wartości średniej). Drzewo, które jest zorganizowane w ten sposób, jest zwane ***optymalnym***. Nietrudno zauważyć, że jeżeli wszystkie klucze charakteryzuje to samo prawdopodobieństwo zostania ***kluczem wyszukiwania*** (ang. *search key*), to drzewo znajdujące się po prawej stronie rysunku 3.10 jest optymalne. Jesteśmy zainteresowani przypadkiem, gdy klucze nie charakteryzują się tym samym prawdopodobieństwem. Przykładem takiego przypadku byłoby przeszukanie jednego z drzew z rysunku 3.10 w poszukiwaniu losowo wybranego imienia mieszkańców Polski. Imię *Tomasz* występuje częściej niż *Urszula*, więc należałoby przypisać mu większe prawdopodobieństwo (w podrozdziale A.8.1 w dodatku A zawarto omówienie problematyki losowości).

Omówimy przypadek, w którym wiadomo, że klucz wyszukiwania występuje w drzewie. Uogólnienie, w przypadku którego klucz może nie występować w drzewie, zostanie bliżej zbadane w ćwiczeniach. W celu zminimalizowania średniego czasu wyszukiwania musimy znać złożoność czasową operacji lokalizowania klucza. Dlatego, zanim przejdziemy dalej, zapiszemy i przeanalizujemy algorytm, który służy do wyszukiwania klucza w drzewie wyszukiwania binarnego. Algorytm wykorzystuje następującą strukturę danych:

```
struct nodetype
{
   keytype key;
   nodetype* left;
   nodetype* right;
};

typedef nodetype* node_pointer;
```

Deklaracja ta oznacza, że zmienna typu `node_pointer` jest wskaźnikiem do rekordu typu `nodetype`. Oznacza to, że jej wartością jest adres pamięci takiego rekordu.

Algorytm 3.8. Przeszukiwanie drzewa binarnego

Problem: określ wierzchołek zawierający klucz w drzewie wyszukiwania binarnego. Zakłada się, że klucz występuje w drzewie.

Dane wejściowe: wskaźnik *tree* do drzewa wyszukiwania binarnego oraz klucz *keyin*.

Dane wyjściowe: wskaźnik *p* do wierzchołka zawierającego klucz.

```
void search (node_pointer tree,
             keytype keyin,
             node_pointer& p)
{
   bool found;

   p = tree;
   found = false;
   while (!found)
      if (p->key == keyin)
         found = true;
      else if (keyin < p->key)
         p = p->left;          // Przejście do lewego potomka.
      else
         p = p->right;         // Przejście do prawego potomka.
}
```

Liczba porównań wykonywanych przez procedurę *search* w celu zlokalizowania klucza nosi nazwę ***czasu wyszukiwania*** (ang. *search time*). Naszym celem jest określenie drzewa, dla którego średni czas wyszukiwania jest najmniejszy. Zgodnie z dyskusją zawartą w podrozdziale 1.2 zakładamy, że porównania są implementowane w sposób wydajny. Przy tym założeniu w każdym przebiegu pętli `while` wykonywane jest tylko jedno porównanie. Stąd czas wyszukiwania danego klucza wynosi

$$depth(key) + 1$$

gdzie *depth*(*key*) jest głębokością wierzchołka zawierającego klucz. Przykładowo, ze względu na fakt, że głębokość klucza zawierającego wartość *Urszula* wynosi 2 w lewym poddrzewie na rysunku 3.10, czas wyszukiwania klucza *Urszula* wynosi

$$depth(Urszula) + 1 = 2 + 1 = 3$$

Niech $Key_1, Key_2, \ldots, Key_n$ będą n uporządkowanymi kluczami oraz niech p_i będzie prawdopodobieństwem tego, że Key_i jest kluczem wyszukiwania. Jeżeli c_i oznacza liczbę porównań koniecznych do znalezienia klucza Key_i w danym drzewie, to średni czas wyszukiwania dla tego drzewa wynosi

$$\sum_{i=1}^{n} c_i p_i$$

Jest to wartość, którą chcemy zminimalizować.

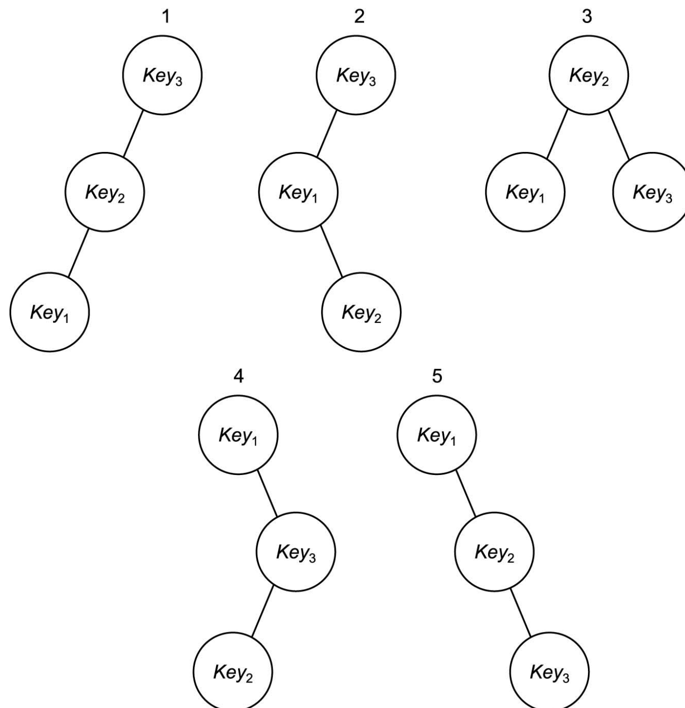
Przykład 3.7. Rysunek 3.11 przedstawia pięć różnych drzew dla $n = 3$. Faktyczne wartości kluczy nie są istotne. Jedynym wymaganiem jest to, aby były one uporządkowane. Jeżeli

$$p_1 = 0{,}7 \quad p_2 = 0{,}2 \quad \text{oraz} \quad p_3 = 0{,}1$$

to średnie czasy wyszukiwania dla drzew z rysunku 3.11 wynoszą:

1. 3(0,7)+2(0,2)+1(0,1) = 2,6
2. 2(0,7)+3(0,2)+1(0,1) = 2,1
3. 2(0,7)+1(0,2)+2(0,1) = 1,8
4. 1(0,7)+3(0,2)+2(0,1) = 1,5
5. 1(0,7)+2(0,2)+3(0,1) = 1,4

Rysunek 3.11. *Możliwe drzewa wyszukiwania binarnego dla przypadku trzech kluczy*



Piąte drzewo jest optymalne.

Ogólnie rzecz biorąc, nie możemy znaleźć optymalnego drzewa wyszukiwania binarnego poprzez rozpatrzenie wszystkich drzew wyszukiwania binarnego, gdyż liczba takich drzew jest co najmniej wykładnicza w stosunku do n. Udowodnimy to wykazując, że jeżeli tylko rozważymy wszystkie drzewa wyszukiwania binarnego o głębokości $n–1$, otrzymamy wykładniczą liczbę drzew. W drzewie wyszukiwania binarnego o głębokości $n–1$ pojedynczy wierzchołek na każdym z $n–1$ poziomów (oprócz korzenia) może znajdować albo na lewo, albo na prawo od

swojego rodzica, co oznacza, że istnieją dwie możliwości na każdym z tych poziomów. To oznacza, że liczba różnych drzew wyszukiwania binarnego o głębokości $n-1$ wynosi 2^{n-1}.

Wykorzystamy programowanie dynamiczne do opracowania bardziej wydajnego algorytmu. Załóżmy, że klucze od Key_i do Key_j są ułożone w drzewie, które minimalizuje

$$\sum_{m=i}^{j} c_m p_m$$

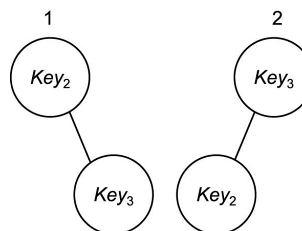
gdzie c_m jest liczbą porównań, wymaganych do zlokalizowania klucza Key_m w drzewie. Takie drzewo będziemy nazywać ***optymalnym*** dla tych kluczy. Wartość optymalną będziemy oznaczać przez $A[i][j]$. Ze względu na fakt, że zlokalizowanie klucza w drzewie zawierającym jeden klucz wymaga jednego porównania, $A[i][i] = p_i$.

Przykład 3.8. Załóżmy, że mamy trzy klucze oraz prawdopodobieństwa określone w przykładzie 3.7, to znaczy:

$$p_1 = 0{,}7 \quad p_2 = 0{,}2 \quad \text{oraz} \quad p_3 = 0{,}1$$

W celu określenia $A[2][3]$ musimy rozważyć dwa drzewa z rysunku 3.12. Dla tych dwóch drzew otrzymujemy:

1. $1(p_2)+2(p_3) = 1(0{,}2)+2(0{,}1) = 0{,}4$
2. $2(p_2)+1(p_3) = 2(0{,}2)+1(0{,}1) = 0{,}5$



Rysunek 3.12. *Drzewa wyszukiwania binarnego utworzone z kluczy Key_2 oraz Key_3*

Pierwsze drzewo jest optymalne i

$$A[2][3] = 0{,}4$$

Należy zauważyć, że drzewo optymalne otrzymane w przykładzie 3.8 stanowi prawe poddrzewo korzenia drzewa optymalnego, otrzymanego w przykładzie 3.7. Nawet gdyby nie było ono dokładnie takie samo, jak prawe poddrzewo, związany z nim średni czas wyszukiwania musiałby być taki sam. W przeciwnym wypadku moglibyśmy zastąpić drzewem optymalnym z przykładu 3.8 wspomniane prawe poddrzewo z przykładu 3.7, otrzymując w wyniku drzewo o krótszym średnim czasie wyszukiwania. Ogólnie rzecz biorąc, dowolne poddrzewo drzewa optymalnego

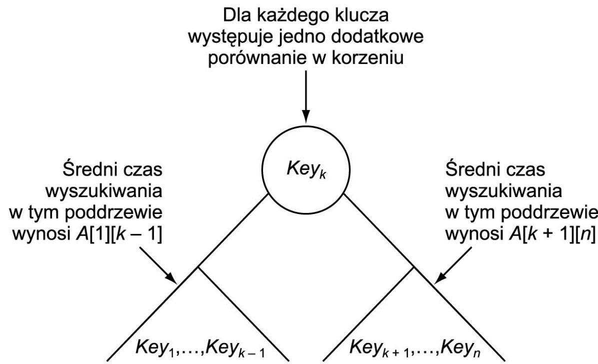
musi być optymalne dla kluczy w tym poddrzewie. Zatem zachowana zostaje zasada optymalności.

Niech drzewo 1 będzie drzewem optymalnym przy uwzględnieniu ograniczenia mówiącego o tym, że klucz Key_1 znajduje się w korzeniu, drzewo 2 niech będzie drzewem optymalnym przy uwzględnieniu ograniczenia mówiącego o tym, że klucz Key_2 znajduje się w korzeniu, …, niech drzewo n będzie drzewem optymalnym przy uwzględnieniu ograniczenia mówiącego o tym, że klucz Key_n znajduje się w korzeniu. Dla $1 \leq k \leq n$ poddrzewa drzewa k muszą być optymalne, a więc średnie czasy wyszukiwania w tych poddrzewach są zgodne z tym, co przedstawiono na rysunku 3.13. Rysunek ten pokazuje również, że dla każdego $m \neq k$ wymagana jest dokładnie o jeden większa liczba porównań w celu zlokalizowania klucza Key_m w drzewie k niż w celu zlokalizowania tego klucza w poddrzewie, w którym się on znajduje (dodatkowe porównanie jest związane z korzeniem). Owo dodatkowe porównanie dodaje $1 \times p_m$ do średniego czasu wyszukiwania klucza Key_m w drzewie k. Określiliśmy, że średni czas wyszukiwania dla drzewa k wynosi:

$$\underbrace{A[1][k-1]}_{\substack{\text{Średni czas} \\ \text{w lewym poddrzewie}}} + \underbrace{p_1 + \ldots p_{k-1}}_{\substack{\text{Dodatkowy czas związany} \\ \text{z porównaniem w korzeniu}}} + \underbrace{p_k}_{\substack{\text{Średni czas} \\ \text{wyszukania korzenia}}} + \underbrace{A[k+1][n]}_{\substack{\text{Średni czas} \\ \text{w prawym poddrzewie}}} + \underbrace{p_{k+1} + \ldots p_n}_{\substack{\text{Dodatkowy czas związany} \\ \text{z porównaniem w korzeniu}}}$$

co jest równoważne:

$$A[1][k-1] + A[k+1][n] + \sum_{m=1}^{n} p_m$$



Rysunek 3.13. *Optymalne drzewo wyszukiwania binarnego przy założeniu, że klucz Key_k jest korzeniem*

Ze względu na fakt, że jedno z k drzew musi być optymalne, średni czas wyszukiwania optymalnego drzewa określa zależność:

$$A[1][n] = \underbrace{minimum}_{1 \leq k \leq n}(A[1][k-1] + A[k+1][n]) + \sum_{m=1}^{n} p_m$$

gdzie $A[1][0]$ i $A[n+1][n]$ są z definicji równe 0. Chociaż suma prawdopodobieństw w ostatnim wyrażeniu wynosi bez wątpienia 1, zapisaliśmy ją jako sumę, ponieważ teraz chcemy uogólnić rezultat. W powyższej dyskusji nie ma nic, co wymagałoby, aby klucze miały wartości od Key_1 do Key_n. Oznacza to, że dyskusja odnosi się ogólnie do kluczy od Key_i do Key_j, gdzie $i < j$. W ten sposób otrzymujemy:

$$A[1][j] = \underbrace{minimum}_{i \le k \le j}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m \quad i < j$$
$$A[i][i] = p_i \tag{3.6}$$
$$A[i][i-1] \text{ oraz } A[j+1][j] \text{ są z definicji równe } 0$$

Wykorzystując zależności 3.6 możemy zapisać algorytm służący do określania optymalnego drzewa wyszukiwania binarnego. Ze względu na fakt, że wartość $A[i][j]$ jest obliczana na podstawie elementów z i-tego wiersza, ale na lewo od elementu $A[i][j]$ oraz na podstawie elementów z j-tej kolumny, ale poniżej elementu $A[i][j]$, kontynuujemy działania, obliczając po kolei wartości na każdej przekątnej (podobnie, jak miało to miejsce w przypadku algorytmu 3.6). Poszczególne etapy algorytmu są tak podobne do zawartych w algorytmie 3.6, że nie zamieścimy przykładu ilustrującego wykonywane operacje. Zamiast tego podamy po prostu algorytm, po którym zostanie przedstawiony obszerny przykład, prezentujący rezultaty jego zastosowania. Tablica R utworzona przez algorytm zawiera indeksy kluczy wybranych dla korzenia w każdym etapie. Przykładowo, $R[1][2]$ jest indeksem klucza znajdującego się w korzeniu drzewa optymalnego, zawierającego pierwsze dwa klucze, natomiast $R[2][4]$ jest indeksem klucza znajdującego się w korzeniu drzewa optymalnego zawierającego drugi, trzeci i czwarty klucz. Po przeanalizowaniu algorytmu omówimy sposób budowania drzewa optymalnego na podstawie tablicy R.

Algorytm 3.9. Optymalne drzewo przeszukiwania binarnego

Problem: określ optymalne drzewo wyszukiwania binarnego dla zbioru kluczy, z których każdy posiada przypisane odpowiednie prawdopodobieństwo zostania kluczem wyszukiwania.

Dane wejściowe: n, liczba kluczy, oraz tablica liczb rzeczywistych p indeksowana od 1 do n, gdzie $p[i]$ jest prawdopodobieństwem wyszukiwania i-tego klucza.

Dane wyjściowe: zmienna *minavg*, której wartością jest średni czas wyszukiwania optymalnego drzewa wyszukiwania binarnego oraz dwuwymiarowa tablica R, na podstawie której można skonstruować drzewo optymalne. R posiada wiersze indeksowane od 1 do $n+1$ oraz kolumny indeksowane od 0 do n. $R[i][j]$ jest indeksem klucza znajdującego się w korzeniu drzewa optymalnego, zawierającego klucze od i-tego do j-tego.

```
void optsearch (int n,
                const float p[],
                float& minavg,
                index R[][])
{
   index i, j, k, diagonal;
   float A[1..n+1][0..n];

   for (i = 1; i <= n; i++) {
      A[i][i - 1] = 0;
      A[i][i] = p[i];
      R[i][i] = i;
```

```
      R[i][i - 1] = 0;
   }
   A[n+1][n] = 0;
   A[n+1][n] = 0;
   for (diagonal = 1; diagonal <= n - 1; diagonal++)
      for (i = 1; i <= n - diagonal; i++) {      // Przekątna 1 znajduje się
                                                 // tuż nad główną przekątną.
         j = i+diagonal;
         A[i][j] = minimum (A[i][k - 1]+A[k+1][j])+ Σ_{m=i}^{j} p_m ;
                   i ≤ k ≤ j
         R[i][j] = wartość k, która dała minimum;
      }
   minavg = A[1][n];
}
```

Analiza algorytmu 3.9.

Złożoność czasowa w każdym przypadku (optymalne drzewo wyszukiwania binarnego)

Operacja podstawowa: instrukcje wykonywane dla każdej wartości *k*. W ich skład wchodzi porównanie sprawdzające występowanie wartości minimalnej. Wartość sumy $\sum_{m=i}^{j} p_m$ nie musi być obliczana za każdym razem od początku. W ćwiczeniach Czytelnik zostanie poproszony o znalezienie wydajnego sposobu obliczania tych sum.

Rozmiar danych wejściowych: *n*, liczba kluczy.

Sterowanie tym algorytmem jest niemal identyczne, jak w przypadku algorytmu 3.6. Jedyna różnica polega na tym, że dla danych wartości *diagonal* oraz *i* operacja podstawowa jest wykonywana *diagonal*+1 razy. Analiza, podobna jak w przypadku algorytmu 3.6, pozwala określić, że:

$$T(n) = \frac{n(n-1)(n+4)}{6} \in \Theta(n^3)$$

Poniższy algorytm konstruuje drzewo binarne na podstawie tablicy *R*. Należy przypomnieć, że tablica *R* zawiera indeksy kluczy wybranych w każdym etapie jako korzenie.

Algorytm 3.10. Budowa optymalnego drzewa przeszukiwania binarnego

Problem: zbuduj optymalne drzewo wyszukiwania binarnego.

Dane wejściowe: *n*, liczba kluczy, tablica *Key*, zawierająca *n* uporządkowanych kluczy oraz tablica *R*, utworzona przez algorytm 3.9. *R*[*i*][*j*] jest indeksem klucza w korzeniu drzewa optymalnego, zawierającego klucze od *i*-tego do *j*-tego.

Dane wyjściowe: wskaźnik *tree* do optymalnego drzewa wyszukiwania binarnego, zawierającego *n* kluczy.

```
node_pointer tree (index i, j)
{
   index k;
   node_pointer p;

   k = R[i][j];
   if (k == 0)
      return NULL;
   else
      p = new nodetype;
      p -> key = Key[k];
      p -> left = tree(i, k - 1);
      p -> right = tree(k+1, j);
      return p;
   }
}
```

Instrukcja `p = new nodetype` tworzy nowy wierzchołek i umieszcza jego adres w *p*. Zgodnie z przyjętymi konwencjami zapisu algorytmów rekurencyjnych parametry *n*, *Key* oraz *R* nie są danymi wejściowymi funkcji *tree*. Gdyby algorytm zaimplementowano definiując *n*, *Key* oraz *R* globalnie, wskaźnik *root* do korzenia optymalnego drzewa wyszukiwania binarnego byłby otrzymywany poprzez wywołanie funkcji *tree* w następujący sposób:

```
root = tree(1, n);
```

Nie zilustrowaliśmy działań wykonywanych przez algorytm 3.9, ponieważ są one podobne, jak w przypadku algorytmu 3.6 (minimalna liczba operacji mnożenia). Podobnie i tym razem nie będziemy ilustrować działań wykonywanych przez algorytm 3.10, ponieważ jest on podobny do algorytmu 3.7 (wyświetlenie kolejności optymalnej). Zamiast tego przedstawimy obszerny przykład, obrazujący wyniki zastosowania algorytmów 3.9 oraz 3.10.

Przykład 3.9. Załóżmy, że mamy następujące wartości w tablicy *Key*:

Damian	Izabela	Rudolf	Waldemar
Key[1]	Key[2]	Key[3]	Key[4]

oraz

$$p_1 = \frac{3}{8} \quad p_2 = \frac{3}{8} \quad p_3 = \frac{1}{8} \quad p_4 = \frac{1}{8}$$

Tablice *A* oraz *R* utworzone przez algorytm 3.9 przedstawiono na rysunku 3.14, natomiast drzewo utworzone przez algorytm 3.10 przedstawiono na rysunku 3.15. Minimalny średni czas wyszukiwania wynosi 7/4.

Należy zauważyć, że *R*[1][2] może być równe 1 lub 2. Wynika to z faktu, że któryś z tych indeksów może być indeksem korzenia w drzewie optymalnym, zawierającym tylko pierwsze dwa klucze. Stąd oba te indeksy dają minimalną wartość

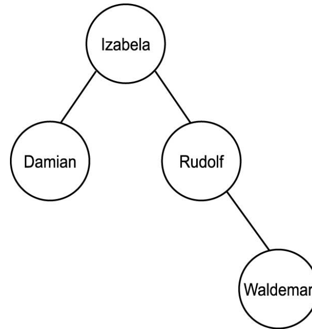
Rysunek 3.14.
Tablice A oraz R utworzone przez algorytm 3.9 w przypadku użycia go wobec realizacji problemu z przykładu 3.9

	1	2	3	4	5
1	0	$\frac{3}{8}$	$\frac{9}{8}$	$\frac{11}{8}$	$\frac{7}{4}$
2		0	$\frac{3}{8}$	$\frac{5}{8}$	1
3			0	$\frac{1}{8}$	$\frac{3}{8}$
4				0	$\frac{1}{8}$
5					0

A

	1	2	3	4	5
1	0	1	1	2	2
2		0	2	2	2
3			0	3	3
4				0	4
5					0

B



Rysunek 3.15.
Drzewo utworzone w przypadku zastosowania algorytmów 3.9 oraz 3.10 wobec realizacji problemu z przykładu 3.9

A[1][2] w algorytmie 3.9, co oznacza, że dowolny z nich może zostać wybrany jako *R*[1][2].

Przedstawiony algorytm określania optymalnego drzewa wyszukiwania binarnego pochodzi z pracy Gilberta i Moore'a (1959). Algorytm $\Theta(n^2)$ można otrzymać z metody przyspieszającej programowania dynamicznego autorstwa Yao (1982).

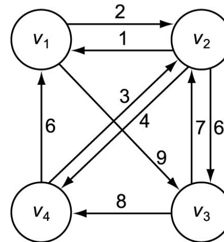
3.6. Problem komiwojażera

Załóżmy, że komiwojażer planuje podróż w interesach. Podróż ta uwzględnia odwiedzenie 20 miast. Każde miasto jest połączone z niektórymi innymi miastami za pomocą dróg. W celu zminimalizowania czasu podróży chcemy określić najkrótszą trasę, która rozpoczyna się w rodzinnym mieście komiwojażera, przebiega przez wszystkie miasta raz i kończy się w punkcie startu. Problem określania najkrótszej trasy nosi nazwę problemu komiwojażera.

Realizacja tego problemu może być reprezentowana przez graf ważony, w którym każdy wierzchołek reprezentuje miasto. Podobnie jak w podrozdziale 3.2 uogólniamy problem, tak aby uwzględnić przypadek, w którym waga (odległość) związana z jednym kierunkiem połączenia dwóch wierzchołków może być różna niż w przypadku drugiego kierunku. Ponownie przyjmujemy, że wagi mają wartości nieujemne. Na rysunkach 3.2 oraz 3.16 przedstawiono takie grafy ważone. ***Trasa*** (ang. *tour*), określana również mianem *drogi Hamiltona* (ang. *Hamiltonian circuit*), w grafie skierowanym jest drogą wiodącą z wierzchołka do niego samego, przechodzącą przez wszystkie inne wierzchołki dokładnie raz. ***Optymalna trasa*** (ang. *optimal*

tour) w ważonym grafie skierowanym jest taką drogą, która posiada najmniejszą długość. Problem komiwojażera polega na znalezieniu optymalnej trasy w ważonym grafie skierowanym, kiedy istnieje przynajmniej jedna trasa. Ze względu na fakt, że wierzchołek początkowy nie ma wpływu na długość optymalnej trasy, jako wierzchołek początkowy będziemy traktować wierzchołek v_1. Poniżej przedstawiono trzy trasy i ich długości dla grafu z rysunku 3.16:

$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 21$$



Rysunek 3.16. *Optymalna trasa ma postać: $[v_1, v_3, v_4, v_2, v_1]$*

Ostatnia trasa jest optymalna. Realizację problemu rozwiązaliśmy, rozważając po prostu wszystkie możliwe trasy. W ogólnym przypadku może istnieć krawędź łącząca każdy wierzchołek z każdym innym wierzchołkiem. Jeżeli rozważymy wszystkie możliwe trasy, drugi wierzchołek na trasie może być jednym spośród $n-1$ wierzchołków, trzeci wierzchołek — jednym spośród $n-2$ wierzchołków, …, n-ty wierzchołek — ostatnim wierzchołkiem. Stąd całkowita liczba tras wynosi

$$(n-1)(n-2)\cdots 1 = (n-1)!$$

co oznacza wartość gorszą od wykładniczej.

Pojawia się pytanie, czy można do tego problemu zastosować programowanie dynamiczne. Należy zauważyć, że jeżeli v_k jest pierwszym wierzchołkiem po v_1 na trasie optymalnej, to droga podrzędna tej trasy, wiodąca z v_k do v_1, musi być najkrótszą drogą z v_k do v_1, która przechodzi przez wszystkie pozostałe wierzchołki dokładnie raz. Oznacza to, że zasada optymalności ma zastosowanie i możemy wykorzystać programowanie dynamiczne. W tym celu będziemy reprezentować graf przez macierz przyległości W, podobnie jak miało to miejsce w podrozdziale 3.2. Na rysunku 3.17 przedstawiono macierz przyległości, reprezentującą graf z rysunku 3.16. Niech

V = zbiór wszystkich wierzchołków

A = podzbiór zbioru V

$D[v_i][A]$ = długość najkrótszej drogi z v_i do v_1 przechodzącej przez każdy wierzchołek w A dokładnie raz

Rysunek 3.17.
Macierz przyległości W, reprezentująca graf z rysunku 3.16

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Przykład 3.10. Dla grafu z rysunku 3.16

$$V = \{v_1, v_2, v_3, v_4\}$$

Należy zauważyć, że zapis $\{v_1, v_2, v_3, v_4\}$ wykorzystuje nawiasy klamrowe w celu reprezentowania zbioru, gdy zapis $[v_1, v_2, v_3, v_4]$ wykorzystuje nawiasy kwadratowe w celu reprezentowania drogi. Jeżeli $A = \{v_3\}$, to

$$\begin{aligned} D[v_2][A] &= length[v_2, v_3, v_1] \\ &= \infty \end{aligned}$$

Jeżeli $A = \{v_3, v_4\}$, to

$$\begin{aligned} D[v_2][A] &= minimum(length[v_2, v_3, v_4, v_1], length[v_2, v_4, v_3, v_1]) \\ &= minimum(20, \infty) = 20 \end{aligned}$$

Ze względu na fakt, że zbiór $V-\{v_1, v_j\}$ zawiera wszystkie wierzchołki oprócz v_1 oraz v_j i ma tu zastosowanie zasada optymalności, możemy stwierdzić, co następuje:

$$\text{Dlugość trasy minimalnej} = \underbrace{minimum}_{2 \le j \le n}(W[1][j] + D[v_j][V - \{v_1, v_j\}])$$

i ogólnie dla $i \neq 1$ oraz v_i nie należącego do A

$$\begin{aligned} &D[v_i][A] = \underbrace{minimum}_{j:v_j \in A}(W[i][j] + D[v_j][A - \{v_j\}]) \text{ } jeżeli\ A \neq \varnothing \\ &D[v_i][\varnothing] = W[i][1] \end{aligned} \qquad (3.7)$$

Możemy utworzyć algorytm programowania dynamicznego dla problemu komiwojażera, korzystając z zależności 3.7. Jednak najpierw pokażemy, jak algorytm ten działa.

Przykład 3.11. Poniżej określimy optymalną trasę dla grafu reprezentowanego na rysunku 3.17. Najpierw bierzemy pod uwagę zbiór pusty:

$$D[v_2][\varnothing] = 1$$
$$D[v_3][\varnothing] = \infty$$
$$D[v_4][\varnothing] = 6$$

Następnie rozważamy wszystkie zbiory, zawierające jeden element:

$$D[v_3][\{v_2\}] = minimum(W[3][j] + D[v_j][\{v_2\} - \{v_j\}])$$
$$= W[3][2] + D[v_2][\varnothing] = 7 + 1 = 8$$

Podobnie:

$$D[v_4][\{v_2\}] = 3 + 1 = 4$$
$$D[v_2][\{v_3\}] = 6 + \infty = \infty$$
$$D[v_4][\{v_3\}] = \infty + \infty = \infty$$
$$D[v_2][\{v_4\}] = 4 + 6 = 10$$
$$D[v_3][\{v_4\}] = 8 + 6 = 14$$

Następnie rozważamy wszystkie zbiory, zawierające dwa elementy:

$$D[v_4][\{v_2,v_3\}] = \underbrace{minimum}_{j:v_j \in \{v_2,v_3\}}(W[4][j] + D[v_j][\{v_2,v_3\} - \{v_j\}])$$
$$= minimum(W[4][2] + D[v_2][\{v_3\}], W[4][3] + D[v_3][\{v_2\}])$$
$$= minimum(3 + \infty, \infty + 8) = \infty$$

Podobnie:

$$D[v_3][\{v_2, v_4\}] = minimum(7 + 10, 8 + 4) = 12$$
$$D[v_2][\{v_3, v_4\}] = minimum(6 + 14, 4 + \infty) = 20$$

W końcu obliczamy długość optymalnej trasy:

$$D[v_1][\{v_2,v_3,v_4\}] = \underbrace{minimum}_{j:v_j \in \{v_2,v_3,v_4\}}(W[1][j] + D[v_j][\{v_2,v_3,v_4\} - \{v_j\}])$$
$$= minimum(W[1][2] + D[v_2][\{v_3,v_4\}],$$
$$W[1][3] + D[v_3][\{v_2,v_4\}],$$
$$W[1][4] + D[v_4][\{v_2,v_3\}])$$
$$= minimum(2 + 20, 9 + 12, \infty + \infty) = 21$$

Poniżej przedstawiono algorytm programowania dynamicznego dla problemu komiwojażera.

Algorytm 3.11. Algorytm programowania dynamicznego dla problemu komiwojażera

Problem: określ optymalną trasę w ważonym grafie skierowanym. Wagi są liczbami nieujemnymi.

Dane wejściowe: ważony graf skierowany oraz n, liczba wierzchołków w grafie. Graf jest reprezentowany przez dwuwymiarową tablicę W, której wiersze i kolumny są indeksowane od 1 do n. $W[i][j]$ jest wagą krawędzi, prowadzącej od wierzchołka i-tego do j-tego.

Dane wyjściowe: zmienna *minlength*, której wartością jest długość optymalnej trasy oraz dwuwymiarowa tablica P, na podstawie której można skonstruować optymalną trasę. Wiersze tablicy P są indeksowane od 1 do n, zaś jej kolumny są indeksowane przez wszystkie podzbiory zbioru $V-\{v_1\}$. Element $P[i][A]$ jest indeksem pierwszego wierzchołka, znajdującego się po v_i na najkrótszej drodze z v_i do v_1, która przechodzi przez wszystkie wierzchołki należące do A dokładnie raz.

```
void travel (int n,
             const number W[][],
             index P[][],
             number& minlength)
{
   index i, j, k;
   number D[1..n][podzbiór zbioru V - {v1}];

   for (i = 2; i <= n; i++)
      D[i][∅] = W[i][1];
   for (k = 1; k <= n - 2; k++)
      for (wszystkie podzbiory A ⊆ V - {v1} zawierające k wierzchołków)
         for (i, takie że i ≠ 1 oraz vi nie należy do A) {
            D[i][A] = minimum(W[i][j]+D[j][A - {vj}]);
                      j: vj ∈ A
            P[i][A] = wartość j, która daje minimum;
         }
   D[1][V - {v1}] = minimum(W[1][j]+D[j][V - {v1, vj}]);
                    2 ≤ j ≤ n
   P[1][V - {v1}] = wartość j, która daje minimum;
   minlength = D[1][V - {v1}];
}
```

Zanim pokażemy, w jaki sposób można otrzymać optymalną trasę na podstawie tablicy P, przeanalizujemy algorytm. Po pierwsze, potrzebne jest nam twierdzenie.

Twierdzenie 3.1

Dla każdego $n \geq 1$

$$\sum_{k=1}^{n} k\binom{n}{k} = n2^{n-1}$$

Dowód. Jako ćwiczenie dla Czytelnika pozostawiono wykazanie, że:

$$k\binom{n}{k} = n\binom{n-1}{k-1}$$

Stąd:

$$\begin{aligned}\sum_{k=1}^{n} k\binom{n}{k} &= \sum_{k=1}^{n} n\binom{n-1}{k-1} \\ &= n\sum_{k=0}^{n-1}\binom{n-1}{k} \\ &= n2^{n-1}\end{aligned}$$

Ostatnie równanie otrzymano dzięki wykorzystaniu wyniku z przykładu A.10 z dodatku A.

Poniżej przedstawiono analizę algorytmu 3.11.

Analiza algorytmu 3.11.

Złożoność czasowa i przestrzenna w każdym przypadku (algorytm programowania dynamicznego dla problemu komiwojażera)

Operacja podstawowa: czas wykonania pierwszej i ostatniej pętli można pominąć w porównaniu z czasem wykonania pętli środkowej, ponieważ zawiera ona różne poziomy zagnieżdżenia. Stąd jako operację podstawową będziemy traktować instrukcje wykonywane dla każdej wartości v_j. Należy do nich instrukcja dodawania.

Rozmiar danych wejściowych: n, liczba wierzchołków w grafie.

Dla każdego zbioru A zawierającego k wierzchołków musimy rozpatrzyć $n-1-k$ wierzchołków, a dla każdego z tych wierzchołków operacja podstawowa jest wykonywana k razy. Liczba podzbiorów A zbioru $V-\{v_1\}$, zawierających k wierzchołków, wynosi $\binom{n-1}{k}$, więc całkowitą liczbę powtórzeń operacji podstawowej określa zależność:

$$T(n) = \sum_{k=1}^{n-2} (n-1-k)k\binom{n-1}{k} \tag{3.8}$$

Nietrudno wykazać, że:

$$(n-1-k)\binom{n-1}{k} = (n-1)\binom{n-2}{k}$$

Podstawiając to równanie do równania 3.8 otrzymujemy:

$$T(n) = (n-1)\sum_{k=1}^{n-2} k\binom{n-2}{k}$$

Ostatecznie, stosując twierdzenie 3.1, otrzymujemy:

$$T(n) = (n-1)(n-2)2^{n-3} \in \Theta\left(n^2 2^n\right)$$

Ze względu na fakt, że algorytm używa również dużej ilości pamięci, przeanalizujemy także jego złożoność pamięciową, którą określimy symbolem $M(n)$. Pamięć użyta do przechowania tablic $D[v_i][A]$ oraz $P[v_i][A]$ stanowi oczywiście główny składnik poziomu zajętości pamięci. Określimy więc rozmiar tych tablic. Ze względu na fakt, że zbiór $V-\{v_1\}$ zawiera $n-1$ wierzchołków, możemy zastosować wyniki otrzymane w przykładzie A.10 w dodatku A, stwierdzając, że posiada on 2^{n-1} podzbiorów A. Pierwszy indeks tablic D i P należy do zakresu od 1 do n. Stąd:

$$M(n) = 2 \times n2^{n-1} = n2^n \in \Theta\left(n2^n\right)$$

W tym momencie Czytelnik może się zastanawiać, co osiągnęliśmy, skoro nasz algorytm jest rzędu $\Theta(n^2 2^n)$. Poniższy przykład pokazuje jednak, że nawet algorytm o złożoności na takim poziomie może być niekiedy przydatny.

Przykład 3.12. Rudolf i Natalia współzawodniczą o tę samą posadę sprzedawcy. Szef powiedział im w piątek, że ta osoba, która od poniedziałku szybciej odbędzie podróż po całym rewirze, na którego obszarze leży 20 miast, zdobędzie posadę. Na obszarze rewiru znajduje się biuro główne, do którego należy powrócić po odwiedzeniu innych miejscowości. Z każdego miasta istnieje droga do każdego innego miasta. Rudolf stwierdził, że ma cały weekend na opracowanie trasy, więc postanowił po prostu uruchomić na swoim komputerze algorytm siłowy, który rozpatrzyłby wszystkie (20–1)! tras. Natalia przypomniała sobie algorytm programowania dynamicznego, który poznała na zajęciach z algorytmiki. Stwierdzając, że musi wykorzystać każdą nadarzającą się okazję, uruchomiła ten algorytm na swoim komputerze. Zakładając, że czas przetworzenia operacji podstawowej przez algorytm Natalii wynosi 1 mikrosekundę oraz że ten sam czas zajmuje algorytmowi Rudolfa obliczenie długości jednej trasy, czas wykonania każdego z algorytmów wynosi:

$$\text{Algorytm siłowy: } 19!\ \mu\text{s} = 3857 \text{ lat.}$$

$$\text{Algorytm programowania dynamicznego: } (20-1)(20-2)2^{20-3}\ \mu\text{s} = 45 \text{ sekund.}$$

Jak widać, nawet algorytm rzędu $\Theta(n^2 2^n)$ może okazać się przydatny w sytuacji, gdy alternatywą jest algorytm rzędu silnia. Pamięć używana przez algorytm programowania dynamicznego w tym przykładzie ma rozmiar

$$20 \times 2^{20} = 20\,971\,520 \text{ elementów macierzy.}$$

Choć jest to dość duża liczba, z pewnością nie przekracza wartości oferowanej przez współczesne standardy.

Użycie algorytmu rzędu $\Theta(n^2 2^n)$ w celu znalezienia optymalnej trasy jest praktyczne jedynie dla małych wartości n. Gdyby na przykład chodziło o 60 miast, wykonanie algorytmu zajęłoby wiele lat.

Poniżej omówimy sposób określania optymalnej trasy na podstawie tablicy P. Nie podamy algorytmu, a jedynie zilustrujemy sposób postępowania. Elementy tablicy P, wymagane do określenia optymalnej trasy dla grafu reprezentowanego na rysunku 3.16, to:

$$P[1,\{v_2,v_3,v_4\}] \quad P[3,\{v_2,v_4\}] \quad P[4,\{v_2\}]$$

Optymalną trasę otrzymujemy w następujący sposób:

$$\text{Indeks pierwszego wierzchołka} = P[1][\{v_2,v_3,v_4\}] = 3$$

$$\text{Indeks drugiego wierzchołka} = P[3][\{v_2,v_4\}] = 4$$

$$\text{Indeks trzeciego wierzchołka} = P[4][\{v_2\}] = 2$$

Optymalna trasa ma zatem postać:

$$[v_1, v_3, v_4, v_2, v_1]$$

Jak dotąd nikomu nie udało się opracować takiego algorytmu dla problemu komiwojażera, którego złożoność w najgorszym przypadku byłaby lepsza niż wykładnicza. Jednakże nikt również nie udowodnił, że taki algorytm nie istnieje. Problem ten należy do szerokiej klasy blisko ze sobą związanych problemów, które dzielą tę właściwość. Są one tematem rozdziału 9.

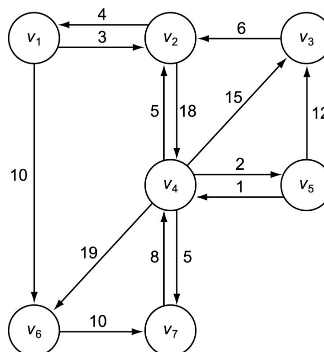
Ćwiczenia

Podrozdział 3.1

1. Wyprowadź równanie 3.1, podane w tym podrozdziale.
2. Użyj dowodu indukcyjnego względem n w celu wykazania, że algorytm typu *dziel i zwyciężaj* dla współczynnika dwumianowego (algorytm 3.1), oparty na równaniu 3.1, oblicza $2\binom{n}{k}-1$ składników w celu określenia wartości $\binom{n}{k}$.
3. Zaimplementuj oba algorytmy rozwiązania problemu współczynnika dwumianowego (algorytm 3.1 oraz 3.2) w swoim systemie i przeanalizuj ich wydajność przy użyciu różnych realizacji problemu.
4. Zmodyfikuj algorytm 3.2 (współczynnik dwumianowy przy użyciu programowania dynamicznego), tak aby wykorzystywał tylko tablicę jednowymiarową, indeksowaną od 0 do k.

Podrozdział 3.2

5. Użyj algorytmu Floyda dla problemu najkrótszych dróg 2 (algorytm 3.4) w celu skonstruowania macierzy D, która będzie zawierać długości najkrótszych dróg, oraz macierzy P, która będzie zawierać najwyższe indeksy wierzchołków pośrednich, należących do najkrótszych dróg dla poniższego grafu. Przedstaw wykonywane działania krok po kroku.



6. Użyj algorytmu wyświetlania najkrótszej drogi (algorytm 3.5) w celu znalezienia najkrótszej drogi z wierzchołka v_7 do wierzchołka v_3 w grafie z ćwiczenia 5, wykorzystując macierz P, określoną w tym ćwiczeniu. Przedstaw wykonywane działania krok po kroku.
7. Przeanalizuj algorytm wyświetlania najkrótszej drogi (algorytm 3.5) i wykaż, że charakteryzuje go liniowa złożoność czasowa.
8. Zaimplementuj algorytm Floyda dla problemu najkrótszych dróg 2 (algorytm 3.4) w swoim systemie i przeanalizuj jego wydajność, używając różnych grafów.
9. Czy algorytm Floyda dla problemu najkrótszych dróg 2 (algorytm 3.4) można tak zmodyfikować, aby dawał w wyniku najkrótszą drogę z danego wierzchołka do innego określonego wierzchołka w grafie? Uzasadnij swoją odpowiedź.
10. Czy algorytm Floyda dla problemu najkrótszych dróg 2 (algorytm 3.4) może być używany do znajdowania najkrótszych dróg w grafie zawierającym ujemne wartości wag? Uzasadnij swoją odpowiedź.

Podrozdział 3.3

11. Znajdź problem optymalizacji, w przypadku którego zasada optymalności nie ma zastosowania, a stąd rozwiązanie optymalne nie może zostać otrzymane przy użyciu programowania dynamicznego. Uzasadnij swoją odpowiedź.

Podrozdział 3.4

12. Znajdź optymalną kolejność oraz koszt obliczenia iloczynu macierzy $A_1 \times A_2 \times A_3 \times A_4 \times A_5$, gdzie

$$\begin{aligned} &A_1 \text{ ma rozmiar } (10 \times 4) \\ &A_2 \text{ ma rozmiar } (4 \times 5) \\ &A_3 \text{ ma rozmiar } (5 \times 20) \\ &A_4 \text{ ma rozmiar } (20 \times 2) \\ &A_5 \text{ ma rozmiar } (2 \times 50) \end{aligned}$$

13. Zaimplementuj algorytm minimalnej liczby operacji mnożenia (algorytm 3.6) oraz algorytm wyświetlania optymalnej kolejności (algorytm 3.7) w swoim systemie i przeanalizuj ich wydajność, używając różnych realizacji problemu.
14. Pokaż, że algorytm typu *dziel i zwyciężaj*, oparty na równaniu 3.5, charakteryzuje wykładnicza złożoność czasowa.
15. Wyprowadź równanie:

$$\sum_{diagonal=1}^{n-1} \left[(n - diagonal) \times diagonal \right] = \frac{n(n-1)(n+1)}{6}$$

Jest ono wykorzystywane w analizie złożoności czasowej w każdym przypadku algorytmu 3.6.

16. Pokaż, że w celu pełnego opatrzenia nawiasami wyrażenia zawierającego n macierzy potrzebnych jest $n-1$ par nawiasów.

17. Przeanalizuj algorytm 3.7 i pokaż, że charakteryzuje go liniowa złożoność czasowa.

18. Zapisz wydajny algorytm, który znajduje optymalną kolejność mnożenia n macierzy $A_1 \times A_2 \times \ldots \times A_n$, gdzie rozmiary każdej macierzy wynoszą 1×1, $1\times d$, $d\times 1$ lub $d\times d$ dla pewnej dodatniej liczby całkowitej d. Przeanalizuj swój algorytm i przedstaw wyniki, używając notacji rzędu.

Podrozdział 3.5

19. Ile różnych drzewa wyszukiwania binarnego można skonstruować przy użyciu sześciu różnych kluczy?

20. Utwórz optymalne drzewo wyszukiwania binarnego dla następujących elementów, których prawdopodobieństwa wystąpienia podano w nawiasach: CASE (0,05), ELSE (0,15), END (0,05), IF (0,35), OF (0,05), THEN (0,35).

21. Znajdź wydajny sposób obliczania sumy $\sum_{m=i}^{j} p_m$, która jest używana w algorytmie optymalnego drzewa wyszukiwania binarnego (algorytm 3.9).

22. Zaimplementuj algorytm optymalnego drzewa wyszukiwania binarnego (algorytm 3.9) oraz algorytm budowania optymalnego drzewa wyszukiwania binarnego (algorytm 3.10) w swoim systemie i przeanalizuj ich wydajność, używając różnych realizacji problemu.

23. Przeanalizuj algorytm 3.10 i przedstaw jego złożoność czasową przy użyciu notacji rzędu.

24. Uogólnij algorytm wyszukiwania optymalnego drzewa binarnego (algorytm 3.9), tak aby uwzględnić przypadek, w którym klucz wyszukiwania może nie występować w drzewie. Oznacza to, że należy przyjąć q_i, gdzie $i = 0, 1, 2, \ldots, n$ i jest prawdopodobieństwem tego, że brakujący klucz można umiejscowić między kluczami Key_i a Key_{i+1}. Przeanalizuj swoją uogólnioną wersję algorytmu i przedstaw wyniki, używając notacji rzędu.

25. Pokaż, że algorytm typu *dziel i zwyciężaj* oparty na równaniu 3.6 charakteryzuje wykładnicza złożoność czasowa.

Podrozdział 3.6

26. Znajdź optymalną trasę dla ważonego grafu skierowanego, reprezentowanego przez poniższą macierz W. Przedstaw wykonywane działania krok po kroku.

$$W = \begin{bmatrix} 0 & 8 & 13 & 18 & 20 \\ 3 & 0 & 7 & 8 & 10 \\ 4 & 11 & 0 & 10 & 7 \\ 6 & 6 & 7 & 0 & 11 \\ 10 & 6 & 2 & 1 & 0 \end{bmatrix}$$

27. Zapisz bardziej szczegółową wersję algorytmu programowania dynamicznego dla problemu komiwojażera (algorytm 3.11).

28. Zaimplementuj swoją bardziej szczegółową wersję algorytmu 3.11 z ćwiczenia 27 w swoim systemie i przeanalizuj jego wydajność, używając kilku realizacji problemu.

Ćwiczenia dodatkowe

29. Podobnie jak w przypadku algorytmów obliczających n-ty wyraz ciągu Fibonacciego (patrz ćwiczenie 34 w rozdziale 1.) rozmiar danych wejściowych algorytmu 3.2 (współczynnik dwumianowy przy użyciu programowania dynamicznego) jest liczbą symboli, użytych do zakodowania liczb n i k. Przeanalizuj ten algorytm pod względem rozmiaru jego danych wejściowych.

30. Określ liczbę możliwych kolejności mnożenia n macierzy $A_1 \times A_2 \times \ldots \times A_n$.

31. Pokaż, że liczbę drzew wyszukiwania binarnego o n kluczach określa wzór:

$$\frac{1}{n+1}\binom{2n}{n}$$

32. Czy można opracować algorytm optymalnego drzewa wyszukiwania binarnego (algorytm 3.9), wykonywany w czasie kwadratowym?

33. Wykorzystaj programowanie dynamiczne w celu zapisania algorytmu znajdującego maksymalną sumę w dowolnej ciągłej liście podrzędnej danej listy n liczb rzeczywistych. Przeanalizuj swój algorytm i przedstaw rezultaty, używając notacji rzędu.

34. Rozważmy dwa ciągi znaków: S_1 oraz S_2. Przykładowo, mogą one mieć postać S_1 = A$CMA*MN oraz S_2 = AXMC4ANB. Zakładając, że ciąg podrzędny ciągu może zostać skonstruowany poprzez usunięcie dowolnej liczby znaków z dowolnych pozycji, wykorzystaj programowanie dynamiczne w celu utworzenia algorytmu, znajdującego najdłuższy wspólny ciąg podrzędny ciągów S_1 i S_2. Algorytm ten zwraca wspólny ciąg podrzędny o maksymalnej długości każdego z ciągów.