ANNA BOROWSKA

ALGORYTMY I STRUKTURY DANYCH – ĆWICZENIA

Część I
Analiza i techniki projektowania algorytmów

Politechnika Białostocka
Białystok 2020

Anna Borowska

Algorytmy i struktury danych – ćwiczenia

Część I
Analiza i techniki projektowania algorytmów

Politechnika Białostocka

OFICYNA WYDAWNICZA POLITECHNIKI BIAŁOSTOCKIEJ
BIAŁYSTOK 2020

Recenzenci:
dr hab. Bożena Woźna-Szcześniak, prof. UJDWAT

Redaktor naukowy dyscypliny:
prof. dr hab. Jarosław Stepaniuk

Redakcja i korekta językowa:
Katarzyna Duniewska

Projekt okładki:
Marcin Dominów

Zdjęcie na okładce:
jeanvdmeulen, https://pixabay.com/pl/photos/hq-tre-hp-core-laptop-z-i5-laptop-3718328

Redakcja techniczna, skład, grafika:
Oficyna Wydawnicza Politechniki Białostockiej

ISBN 978-83-66391-35-2
e-ISBN 978-83-66391-36-9
DOI: 10.24427/978-83-66391-36-9

Druk: PARTNER POLIGRAFIA Andrzej Kardasz

Oficyna Wydawnicza Politechniki Białostockiej
ul. Wiejska 45C, 15-351 Białystok
tel.: 85 746 91 37
e-mail: oficyna.wydawnicza@pb.edu.pl
www.pb.edu.pl

Moim Rodzicom – Teresie i Antoniemu

Spis treści

Wstęp

Niniejszy skrypt jest przeznaczony dla Czytelników zainteresowanych projektowaniem efektywnych algorytmów, w tym przede wszystkim dla studentów studiów informatycznych. Może także posłużyć jako podręcznik do samodzielnej nauki dla uczniów szkół średnich pasjonujących się programowaniem lub jako wskazówka przy pisaniu konspektu do przedmiotu algorytmy i struktury danych. Część I skryptu prezentuje podstawy współczesnych metod projektowania i analizy algorytmów. Planowana część II – *Struktury danych* – będzie zawierała algorytmy wykorzystujące podstawowe struktury danych, takie jak listy, grafy i drzewa.

Skrypt powstał na podstawie wybranych materiałów do wykładu, ćwiczeń i pracowni komputerowej z przedmiotu algorytmy i struktury danych, przeznaczonych do nauki dla studentów informatyki Politechniki Białostockiej.

Autorka zakłada, że Czytelnik opanował podstawy kombinatoryki, matematyki dyskretnej i (w przypadku rozdziału 1) analizy matematycznej oraz potrafi zaimplementować (zrozumieć) algorytm w języku C++.

Część I skryptu stanowi sześć rozdziałów. Rozdział 1 to wprowadzenie do analizy algorytmów. Omówiono w nim pojęcie specyfikacji algorytmu, poprawność algorytmu, a także złożoność obliczeniową oraz rzędy wielkości funkcji. Rozdział 2 – *Rekurencja* – prezentuje zalety i wady stosowania rozwiązań rekurencyjnych w matematyce i w programowaniu, podaje przykłady różnych rodzajów rekurencji, przypadki jej nieuzasadnionego stosowania oraz metody rozwiązywania różnych zależności rekurencyjnych. Rozdział 3 jest poświęcony najbardziej popularnej technice projektowania algorytmów, jaką jest „dziel i zwyciężaj". Rozdział 4 zawiera przykłady algorytmów wykorzystujących programowanie dynamiczne jako technikę optymalizacji. Rozdział 5 proponuje użyteczne algorytmy wykorzystujące metodę zachłanną. Natomiast w ostatnim rozdziale zostały przedstawione różne sposoby wyszukiwania wzorca w tekście. W ramach przykładów zastosowania omówionych technik pojawiły się wybrane metody sortowania.

Każdy rozdział zawiera krótki opis techniki lub omawianego problemu, intuicyjne przykłady, schematy i gotowe rozwiązania w postaci kodu zapisanego w języku C++. Dla przećwiczenia poznanego materiału, na zakończenie każdego rozdziału, autorka zaproponowała zadania z pełnymi odpowiedziami. W nagłówkach podrozdziałów podana jest lista odnośników do pozycji, do których Czytelnik może zajrzeć w celu rozszerzenia wiedzy dotyczącej omawianego problemu. Lista cytowanych publikacji znajduje się na końcu skryptu.

Rozdział 1. Analiza algorytmów

1. Specyfikacja algorytmu
2. Poprawność częściowa i całkowita algorytmu
3. Złożoność obliczeniowa algorytmu
4. Rzędy wielkości funkcji
5. Zadania różne

Analizę algorytmów przeprowadza się w celu wyselekcjonowania najlepszego algorytmu do rozwiązania zadanego problemu przy użyciu komputera. Należy zatem sprawdzić, który ze znanych algorytmów jest najbardziej odpowiedni do uzyskania żądanego wyniku, biorąc pod uwagę ustalone założenia, a także to, czy jest to algorytm optymalny pod względem złożoności czasowej i pamięciowej, czy rozwiąże on sformułowane zadanie na komputerze w dostępnym czasie i pamięci oraz czy jesteśmy w stanie uzasadnić, że wybrany algorytm zawsze (w oczekiwanych przypadkach) rozwiąże przedstawiony problem prawidłowo.

1. Specyfikacja algorytmu (por. [CLR], [DS], [MG])

Definicja 1 (warunek początkowy algorytmu WP)
Warunek początkowy algorytmu WP to warunek, jaki muszą spełniać dane wejściowe (wartościowanie początkowe zmiennych) algorytmu. Inaczej mówiąc – własności przysługujące danym wejściowym.

Definicja 2 (warunek końcowy algorytmu WK)
Warunek końcowy algorytmu WK to warunek, jaki muszą spełniać dane wyjściowe (wartościowanie końcowe zmiennych) algorytmu uzyskane dla danych wejściowych spełniających WP. Inaczej mówiąc – własności przysługujące danym wyjściowym.

Definicja 3 (specyfikacja algorytmu <WP, WK>)
Specyfikacja algorytmu <WP, WK> określa warunek początkowy WP, jaki muszą spełniać dane, by algorytm mógł zadziałać, i warunek końcowy WK, jaki mają spełniać wyniki obliczenia algorytmu, gdy dane wejściowe spełniają warunek WP.

Przykład 1. Specyfikacja algorytmu obliczającego wartość funkcji $f(x) = \sqrt{x}$ dla $x \in R_+ \cup \{0\}$.

WP: $x \in R_+ \cup \{0\}$

WK: $y = \sqrt{x} \Leftrightarrow x^2 = y$

Przykład 2. Warunek początkowy i końcowy algorytmu znajdującego dla dwóch liczb całkowitych dodatnich x, y ich największy wspólny dzielnik $z = \mathrm{NWD}(x, y)$.

WP: $x, y \in Z_+$

WK: $z \in Z_+ \wedge z \mid x \wedge z \mid y \wedge (\forall v \in Z_+ ((v \mid x \wedge v \mid y) \Rightarrow v \mid z))$

2. Poprawność częściowa i całkowita algorytmu (por. [CLR], [DS], [MG])

Definicja 4 (poprawność częściowa algorytmu)
Algorytm A jest *częściowo poprawny* względem specyfikacji <WP, WK> wtw dla dowolnych danych wejściowych spełniających WP, jeżeli algorytm A kończy obliczenia, to dane wyjściowe algorytmu spełniają WK. WP nie gwarantuje zatrzymania algorytmu.

Definicja 5 (poprawność całkowita algorytmu)
Algorytm A jest *całkowicie poprawny* względem specyfikacji <WP, WK> wtw dla dowolnych danych wejściowych spełniających WP algorytm A kończy obliczenia i dane wyjściowe algorytmu spełniają WK. WP jest gwarancją otrzymania poprawnych wyników.

Przykład 3. Dana jest specyfikacja algorytmu obliczającego sumę k kolejnych liczb naturalnych i propozycja algorytmu rozwiązującego ten problem. Czy podany algorytm jest poprawny ze względu na zadaną specyfikację?

WP: $k \in N_+$

WK: $s = \sum_{i=1}^{k} i$

```
int A1(int k) {
  int s = 0, i = 1;
  while(i <= k) {
    s += i; i++;
  }
  return s;
}
```

Rozwiązanie: Algorytm A1 jest poprawny całkowicie ze względu na podaną specyfikację. Przy dowolnych danych wejściowych spełniających WP program zawsze się zatrzymuje, a zwracane wartości spełniają WK.

Uzasadnienie indukcyjne: Dla $k = 1$ algorytm A1 wykona tylko jedną iterację pętli while (dla $i = 1$) i osiągnie wynik $s = 1$. Załóżmy, że dla $k = t$ (gdzie t jest

pewną liczbą naturalną) algorytm A1 wyznaczy $s = \sum_{i=1}^{t} i$. Pokażemy, że wówczas dla $k = t + 1$ algorytm A1 zwróci wynik $s = \sum_{i=1}^{t+1} i$. Po wykonaniu iteracji (pętli while) dla $i = t$ uzyskamy $s = \sum_{i=1}^{t} i$ (z założenia indukcyjnego) oraz $i = t + 1$. Wyrażenie kontrolne $(i <= k)$ zwróci wartość true, ponieważ $k = t + 1$. Stąd dla $i = t + 1$ wykonane zostaną instrukcje wewnątrz pętli while, w efekcie czego uzyskamy $s = \sum_{i=1}^{t} i + (t + 1) = \sum_{i=1}^{t+1} i$ oraz $i = t + 2$. W tym momencie wyrażenie kontrolne pętli while zwróci wartość false i algorytm A1 ostatecznie zwróci sumę $s = \sum_{i=1}^{t+1} i$. Na mocy zasady indukcji matematycznej możemy stwierdzić, że algorytm A1 jest poprawny całkowicie ze względu na zadaną specyfikację.

Przykład 4. Dana jest specyfikacja <WP, WK> algorytmu obliczającego iloczyn postaci $il = 10 \cdot 20 \cdot \ldots \cdot (E(n/10) \cdot 10)$ (gdzie $E(x)$ oznacza część całkowitą liczby x) i propozycja algorytmu rozwiązującego ten problem. Czy podany algorytm jest poprawny ze względu na zadaną specyfikację?

WP: $n \in N_+$

WK: $il = 10 \cdot 20 \cdot \ldots \cdot (E(n/10) \cdot 10)$

```
int A2(int n) {
  int il = 1, i = 0;
  while(i != n) {
    i += 10; il *= i;
  }
  return il;
}
```

Rozwiązanie: Algorytm A2 jest poprawny częściowo, ale nie jest poprawny całkowicie ze względu na zadaną specyfikację. Dla n niebędącego wielokrotnością 10 algorytm się nie zatrzymuje, ale dla n będącego wielokrotnością 10 algorytm kończy obliczenia i zwraca wartość określoną w WK.

Uzasadnienie: Fakt, że dla n niebędącego wielokrotnością 10 algorytm A2 się nie zatrzymuje, jest oczywisty, ponieważ po pierwszej iteracji i już zawsze będzie wielokrotnością 10.

Uzasadnimy indukcyjnie, że dla $n = 10 \cdot k$ (gdzie $k \in N_+$) algorytm A2 zawsze kończy obliczenia i zwraca wartość określoną w WK.

Dla $k = 1$ $(n = 10)$ algorytm A2 osiągnie wynik $il = 10$.

Załóżmy, że dla $n = 10 \cdot t$ (gdzie t jest pewną liczbą naturalną) algorytm A2 wyznaczy wartość

$$il = 10 \cdot 20 \cdot \ldots \cdot (E((10 \cdot t)/10) \cdot 10) = 10 \cdot 20 \cdot \ldots \cdot (10 \cdot t).$$

Pokażemy, że wówczas dla $n = 10 \cdot (t + 1)$ algorytm A2 zwróci iloczyn

$$il = 10 \cdot 20 \cdot \ldots \cdot (E((10 \cdot (t + 1))/10) \cdot 10) = 10 \cdot 20 \cdot \ldots \cdot (10 \cdot (t + 1)).$$

W momencie, gdy sprawdzane jest wyrażenie kontrolne pętli while dla $i = 10 \cdot t$, mamy wyznaczony iloczyn $il = 10 \cdot 20 \cdot \ldots \cdot (10 \cdot t)$ (z założenia indukcyjnego). Wyrażenie kontrolne $(i \,!= n)$ zwróci wartość true, ponieważ

$n = 10 \cdot (t + 1)$. Stąd wewnątrz pętli wykonane zostaną instrukcje, w wyniku których otrzymamy:

$i = 10 \cdot t + 10 = 10 \cdot (t + 1)$ oraz $il = 10 \cdot 20 \cdot \ldots \cdot (10 \cdot t) \cdot (10 \cdot (t + 1))$.

W tym momencie wyrażenie kontrolne pętli while zwróci wartość false (ponieważ $n = 10 \cdot (t + 1)$) i algorytm A2 ostatecznie zwróci iloczyn $il = 10 \cdot 20 \cdot \ldots \cdot (10 \cdot t) \cdot (10 \cdot (t + 1))$. Na mocy zasady indukcji matematycznej możemy stwierdzić, że dla n będącego wielokrotnością 10 algorytm A2 zawsze kończy obliczenia i zwraca wartość określoną w WK.

3. Złożoność obliczeniowa algorytmu (por. [BDR], [CLR], [DS], [MG])

Definicja 6 (złożoność obliczeniowa algorytmu)
Złożoność obliczeniowa algorytmu to ilość zasobów komputerowych potrzebnych do jego wykonania. Podstawowymi zasobami rozważanymi podczas analizy algorytmów są ilość zajmowanej pamięci oraz czas działania. W związku z tym, na złożoność obliczeniową algorytmu składają się:

- *złożoność pamięciowa* (za jednostkę przyjmuje się zwykle słowo pamięci maszyny) oraz
- *złożoność czasowa* (za jednostkę przyjmuje się wykonanie jednej operacji dominującej).

Definicja 7 (operacja dominująca (elementarna) algorytmu)
Operacja dominująca (*elementarna*) *algorytmu* to operacja charakterystyczna dla danego algorytmu oraz taka, że łączna liczba jej wykonań jest proporcjonalna do liczby wykonań wszystkich operacji jednostkowych w dowolnej komputerowej realizacji algorytmu.

Definicja 8 (złożoność czasowa algorytmu A (ozn. $T(\mathrm{A}, n)$))
Złożoność czasowa algorytmu A to liczba operacji dominujących wykonanych przez algorytm A (w czasie jego realizacji), wyrażona jako funkcja rozmiaru danych n.

Definicja 9 (złożoność pamięciowa algorytmu A (ozn. $S(\mathrm{A}, n)$))
Złożoność pamięciowa algorytmu A to ilość jednostek pamięci potrzebnych do wykonania tego algorytmu (w czasie jego realizacji), wyrażona jako funkcja rozmiaru danych n.

Przykład 5. Dany jest algorytm obliczający iloczyn dwóch macierzy kwadratowych $\mathrm{A}, \mathrm{B} \in Z^{n \times n}$ o elementach całkowitych. Wyznacz złożoność pamięciową i czasową algorytmu. Przyjmij za operację elementarną mnożenie dwóch liczb całkowitych.

```
A3:  for(int i = 1; i <= n; i++)
       for(int j = 1; j <= n; j++) {
         C[i][j] = 0;
         for(int k = 1; k <= n; k++)
           C[i][j] += A[i][k] * B[k][j];
       }
```

$C = [c_{ij}]_{i,j=1,\dots,n} \in Z^{nxn}$, gdzie
$c_{ij} = \sum_{k=1}^{n}(a_{ik} \cdot b_{kj})$ dla $i, j = 1, \dots, n$

Rozwiązanie:
- złożoność pamięciowa $S(\text{A3}, n) = 3n^2 + 4$
  - trzy n^2-elementowe macierze i cztery zmienne typu int
- złożoność czasowa $T(\text{A3}, n) = n^3$
  - n^3 operacji elementarnych (mnożeń)

3.1. Złożoność obliczeniowa średnia i pesymistyczna (por. [BDR], [CLR], [DS], [MG])

Złożoność obliczeniową czasową algorytmu wyrażamy jako funkcję rozmiaru danych n. Wyróżnia się:
- *złożoność pesymistyczną*,
- *złożoność oczekiwaną* (*średnią*).

Definicja 10 (złożoność pesymistyczna algorytmu (ozn. $T_{\max}(\text{A}, n)$))
(nieformalnie) – ilość zasobów komputerowych potrzebnych przy „najgorszych" danych wejściowych rozmiaru n.
(formalnie) – funkcja $T_{\max}(\text{A}, n) = \sup\{t(\text{A}, d): d \in D_n\}$,
przy czym sup{} oznacza kres górny zbioru.

Definicja 11 (złożoność oczekiwana (średnia) algorytmu A (ozn. $T_{\text{sr}}(\text{A}, n)$))
(nieformalnie) – ilość zasobów komputerowych potrzebnych przy „typowych" danych wejściowych rozmiaru n.
(formalnie) – funkcja $T_{\text{sr}}(\text{A}, n) = \sum_{d \in D_n}\big(\text{pr}(d) \cdot t(\text{A}, d)\big)$, gdzie:
D_n – zbiór zestawów danych wejściowych rozmiaru n (przestrzeń wszystkich danych rozmiaru n dla problemu P),
A – algorytm rozwiązujący problem P,
d – dane wejściowe,
$t(\text{A}, d)$ – liczba operacji elementarnych wykonanych przez algorytm A dla zestawu danych d,
$\text{pr}(d)$ – prawdopodobieństwo wystąpienia zestawu danych d.

Przykład 6. Dany jest ciąg uporządkowanych rosnąco liczb całkowitych $(a_n) = (a_0, \dots, a_{n-1}), n \in N$.
(a) Napisz algorytm sekwencyjny (naiwny), który dla dowolnej liczby $x \in Z$ odpowiada na pytanie „Czy x jest elementem ciągu (a_n)?".

(b) Wyznacz złożoność pamięciową i czasową (średnią i pesymistyczną) algorytmu. Za operację elementarną przyjmij porównanie elementu ciągu i liczby x.

Rozwiązanie:

```
bool A4(int n, int *A, int x) {
 int i = 0;
 while(i < n && A[i] != x) i++;
 return i < n;
}
```

- rozmiar danych: n (długość ciągu);
- złożoność pamięciowa: $S(A4, n) = n + 3$ (wektor n-elementowy i 3 zmienne typu int);
- złożoność czasową wyznaczymy dla operacji elementarnej – porównanie elementów wektora z wartością x;
- złożoność czasową pesymistyczną uzyskujemy, gdy $x = A[n–1]$ lub gdy $x \notin A$, wówczas $T_{\max}(A4, n) = n$;
- średnia złożoność czasowa:
  założenia: $\text{pr}(x \in A) = p$, $\text{pr}(x \notin A) = 1 - p$, $\forall_{i=0,\ldots,n-1}\ \text{pr}(x = a_i) = \frac{p}{n}$
  (zakładamy, że prawdopodobieństwo wystąpienia wartości x na dowolnej pozycji jest takie samo)

$$T_{\text{sr}}(A4, n) = \sum_{k=0}^{n-1}\left(\text{pr}(x = a_k)\cdot t(x = a_k)\right) + \text{pr}(x \notin A)\cdot t(x \notin A) =$$

$$= \sum_{k=0}^{n-1}\left(\frac{p}{n}\cdot(k+1)\right) + (1-p)\cdot n = \frac{p}{n}\cdot\sum_{k=1}^{n} k + (1-p)\cdot n =$$

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

4. Rzędy wielkości funkcji (por. [BDR], [CLR], [DS], [MG])

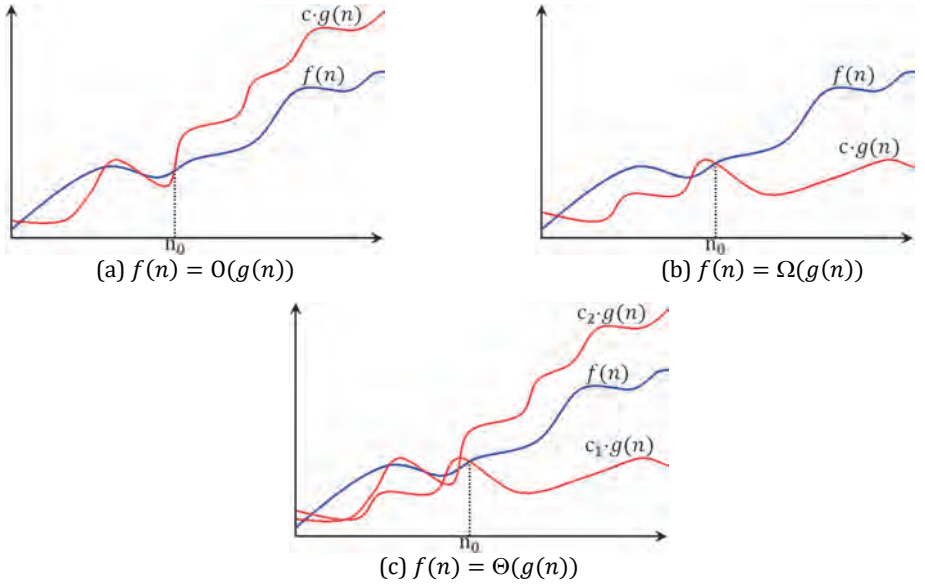
Faktyczna złożoność czasowa algorytmu (rzeczywista ilość wykonanych operacji elementarnych) w trakcie jego realizacji na komputerze (dla konkretnych danych wejściowych) różni się od tej wyznaczonej teoretycznie współczynnikiem proporcjonalności. Obliczając funkcję złożoności algorytmu, liczymy rząd wielkości czasu jego działania (dla dostatecznie dużych danych wejściowych), tzn. zajmujemy się asymptotyczną złożonością algorytmu (inaczej mówiąc, podajemy, jak szybko wzrasta czas działania algorytmu, gdy rozmiar danych dąży do nieskończoności). Na podstawie rzędu wielkości funkcji, za pomocą której opisuje się czas działania algorytmu, określamy jego efektywność oraz możemy porównać złożoność dwóch różnych algorytmów.

Oznaczmy przez f i g funkcje $f, g: N \to R_+ \cup \{0\}$. Dla rzędów wielkości funkcji używamy poniższych oznaczeń.

Definicja 12 (notacja O – asymptotyczna granica górna)
Funkcja f *jest co najwyżej rzędu* g (ozn. $f(n) = O(g(n))$ lub $f = O(g)$), jeżeli istnieją stała rzeczywista $c > 0$ i stała naturalna n_0 takie, że nierówności $0 \le f(n) \le cg(n)$ zachodzą dla każdego $n \ge n_0$. Mówimy wtedy również, że funkcja g *jest majorantą* (*ograniczeniem górnym*) wartości funkcji f. Funkcja g *szacuje z góry* funkcję f.

Definicja 13 (notacja Ω – asymptotyczna granica dolna)
Funkcja f *jest co najmniej rzędu* g (ozn. $f(n) = \Omega(g(n))$ lub $f = \Omega(g)$), jeżeli istnieją stała rzeczywista $c > 0$ i stała naturalna n_0 takie, że nierówności $f(n) \ge cg(n) \ge 0$ zachodzą dla każdego $n \ge n_0$. Mówimy wtedy również, że funkcja g *szacuje z dołu* funkcję f.

Definicja 14 (notacja Θ (theta))
Funkcje f i g *są dokładnie tego samego rzędu* (ozn. $f(n) = \Theta(g(n))$ lub $f = \Theta(g)$), jeżeli istnieją stałe rzeczywiste $c_1, c_2 > 0$ oraz stała naturalna n_0 takie, że dla każdego $n \ge n_0$ zachodzą nierówności $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$. Mówimy wtedy również, że funkcja g *jest asymptotycznie dokładnym oszacowaniem* dla funkcji f.



(a) $f(n) = O(g(n))$ (b) $f(n) = \Omega(g(n))$

(c) $f(n) = \Theta(g(n))$

Rys. 1.1. Intuicje notacji O, Ω, Θ

Twierdzenie 1 (por. [CLR])
Dla dowolnych dwóch funkcji $f(n)$, $g(n)$ mamy $f(n) = \Theta(g(n))$ wtw $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$.

Przykład 7. Korzystając z formalnych definicji Θ i O, udowodnij następujące własności:
(a) $0.01n^2 + 4n = \Theta(n^2)$ (b) $9n^3 \neq O(n)$

Rozwiązanie:
(a) $0.01n^2 + 4n = \Theta(n^2)$

Należy wskazać const $= c_1, c_2 \in R_+$ i $n_0 \in N$ takie, że:

$\forall_{n \geq n_0}$ $c_1 n^2 \leq 0.01n^2 + 4n \leq c_2 n^2$ / n^2 (zakładamy, że $n \neq 0$)

$c_1 \leq 0.01 + \frac{4}{n} \leq c_2$

Prawa strona jest prawdziwa dla każdego $n \geq 1$, gdy $4.01 \leq c_2$, a lewa strona jest prawdziwa dla każdego $n \geq 1$, gdy $c_1 < 0.01$. Zatem wybierając $c_1 = 0.009$, $c_2 = 4.01$ i $n_0 = 1$, możemy powiedzieć, że $0.01n^2 + 4n = \Theta(n^2)$, ponieważ z definicji 14 wynika, że:

$$\exists_{c_1=0.009, c_2=4.01} \exists_{n_0=1} \forall_{n \geq n_0} \quad 0 \leq c_1 n^2 \leq 0.01n^2 + 4n \leq c_2 n^2$$

Uwaga. Istnieją inne możliwości wyboru stałych, ale istotne jest to, że istnieje jakakolwiek możliwość.
(b) $9n^3 \neq O(n)$

Załóżmy przez zaprzeczenie, że:

$$\exists_{\substack{c \in R_+ \\ c=\text{const}}} \exists_{n_0 \in N} \forall_{n \geq n_0} 9n^3 \leq cn \quad / n \text{ (zakładamy, że } n \neq 0)$$

$$\exists_{\substack{c \in R_+ \\ c=\text{const}}} \exists_{n_0 \in N} \forall_{n \geq n_0} 9n^2 \leq c$$

$$\exists_{\substack{c \in R_+ \\ c=\text{const}}} \exists_{n_0 \in N} \forall_{n \geq n_0} \frac{-\sqrt{c}}{3} \leq n \leq \frac{\sqrt{c}}{3} \quad \text{sprzeczność}$$

Nie może być prawdą, że dla dowolnie dużego n jest $n \leq \frac{\sqrt{c}}{3}$, ponieważ $c = \text{const}$.

Do oszacowania złożoności czasowej znanych algorytmów zwykle używamy jednej z następujących funkcji:

- $\log_2 n$ – *złożoność logarytmiczna* (ozn. $\lg n$). Złożoność taką ma np. algorytm wyszukiwania binarnego wartości x w ciągu uporządkowanym. W każdej i-tej iteracji zadanie rozmiaru n_i zostaje sprowadzone do zadania o rozmiarze $n_i/2$ plus stała liczba operacji elementarnych.
- n – *złożoność liniowa*. Złożoność taką ma np. sekwencyjny algorytm znajdowania minimum i maksimum w n-elementowym ciągu liczb. Dla każdego z n elementów wykonywany jest pewien stały zestaw operacji elementarnych.
- $n \lg n$ – *złożoność liniowo-logarytmiczna*. Złożoność taką ma np. algorytm sortowania przez kopcowanie. Przy budowie kopca wykonywanych jest $O(n)$ operacji elementarnych, a przy $(n-1)$-krotnym powtórzeniu dwóch kroków – usuń korzeń kopca oraz przywróć drzewu (tablicy) własność kopca – algorytm wykonuje $O(n \lg n)$ operacji elementarnych.

- n^2 – *złożoność kwadratowa*. Taką złożoność ma np. algorytm dodawania dwóch macierzy kwadratowych. Wewnątrz podwójnej pętli iteracyjnej wykonywana jest stała liczba operacji elementarnych.
- n^k dla $k \in N$ – *złożoność wielomianowa*. Złożoność n^k dla $k = 3$ ma np. algorytm mnożenia dwóch macierzy kwadratowych.

Do rozwiązania problemu, dla którego nie istnieje algorytm o złożoności czasowej szacowanej z góry funkcją n^k (dla $k \in N$), zwykle stosujemy w praktyce algorytmy przybliżone.

- 2^n – *złożoność wykładnicza* 2^n. Złożoność taką ma np. algorytm rozwiązujący problem wież z Hanoi.
- $n!$ – *złożoność* $n!$. Złożoność $n!$ ma np. algorytm, który dla każdej permutacji danych wejściowych wykonuje pewną stałą liczbę operacji elementarnych.

4.1. Fakty pomocnicze do badania rzędów wielkości funkcji (por. [BDR], [CLR], [DA], [DS], [GM])

Lemat 1 (por. [BDR])
Niech f i g oznaczają funkcje $f, g: N \to R_+ \cup \{0\}$. Niech $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \mathrm{c}$, wówczas:

- jeżeli $\mathrm{c} \in R_+$, to $f(n) = \Theta(g(n))$,
- jeżeli $\mathrm{c} = 0$, to $f(n) = O(g(n))$ i $\neg\left(f(n) = \Omega\big(g(n)\big)\right)$,
- jeżeli $\mathrm{c} = +\infty$, to $g(n) = O(f(n))$ i $\neg\left(g(n) = \Omega\big(f(n)\big)\right)$.

Przykład 8. Zbadaj, czy $\sqrt{n} = \Theta(\lg n)$.
Rozwiązanie:

$\lim_{n\to\infty} \frac{\sqrt{n}}{\lg n} \overset{\left[\frac{\infty}{\infty}\right],(a)}{=} \lim_{n\to\infty} \frac{n\ln 2}{2\sqrt{n}} = \lim_{n\to\infty} \left(\frac{\ln 2\sqrt{n}}{2}\right) = \infty$, zatem $\lg n = O(\sqrt{n})$ i $\lg n \neq \Omega(\sqrt{n})$,

więc $\sqrt{n} \neq \Theta(\lg n)$

$\left[\frac{\infty}{\infty}\right]$ stosujemy regułę de L'Hospitala

(a) $\left(\sqrt{n}\right)' = \frac{1}{2\sqrt{n}}$, $(\log_a n)' = \frac{1}{n \ln a}$ dla $a > 0$, $a \neq 1$, $n \neq 0$

Lemat 2

(a) (por. [CLR]) Dla każdego asymptotycznie dodatniego wielomianu $p(n)$ stopnia d ($d \in Z_+$) mamy: $p(n) = \Theta(n^d)$.

(b) Funkcje uporządkowane rosnąco ze względu na ich rząd wielkości: $1, \lg n, \ldots, \sqrt[3]{n}, \sqrt{n}, n, n\lg n, n^2, n^3, \ldots, 2^n, 3^n, n!, n^n$.

(c) (por. [CLR]) Dla dowolnych dwóch funkcji $f(n), g(n)$ mamy:
$f(n) = O(g(n))$ wtw $g(n) = \Omega(f(n))$.

Twierdzenie 2 (por. [GM], s. 56)

$$\forall_{n \in N}\left(\left(a_n \neq 0 \wedge \lim_{n\to\infty}\left|\frac{a_{n+1}}{a_n}\right| = q < 1\right) \Rightarrow \lim_{n\to\infty} a_n = 0\right)$$

Przykład 9. Zbadaj, czy $n^2 = O(2^n)$.
Rozwiązanie:

$a_n = \frac{n^2}{2^n} \neq 0$, $a_{n+1} = \frac{(n+1)^2}{2^{(n+1)}}$, $\lim_{n\to\infty}\left|\frac{(n+1)^2}{2^{(n+1)}} \cdot \frac{2^n}{n^2}\right| = \lim_{n\to\infty}\left|\frac{n^2+2n+1}{2n^2}\right| = \frac{1}{2} < 1$,

więc $\lim_{n\to\infty} \frac{n^2}{2^n} = 0$. Stąd (na mocy lematu 1) $n^2 = O(2^n)$ i $\neg(n^2 = \Omega(2^n))$.

Definicja 15

$(a \in R_+ \setminus \{1\} \wedge x \in R_+) \Rightarrow (\log_a x = y$ wtw $a^y = x)$

5. Zadania różne (por. [CLR], [KW], [MG])

[01] Uzasadnij lemat 2 (b).
Rozwiązanie:

(1) Należy pokazać, że $1 = O(\lg n)$ i $\neg\left(1 = \Omega(\lg n)\right)$. Korzystam z lematu 1.
$\lim_{n\to\infty} \frac{1}{\lg n} = 0$, stąd $1 = O(\lg n)$ i $\neg(1 = \Omega(\lg n))$
Dodatkowo, z lematu 2 (c) wynika, że $\lg n = \Omega(1)$ i $\neg(\lg n = O(1))$.

(2) Należy pokazać, że $\lg n = O(\sqrt[3]{n})$ i $\neg\left(\lg n = \Omega\left(\sqrt[3]{n}\right)\right)$. Korzystam z lematu 1.
$\lim_{n\to\infty} \frac{\lg n}{\sqrt[3]{n}} = {}^{[\frac{\infty}{\infty}],(a)}= \lim_{n\to\infty} \frac{3\sqrt[3]{n^2}}{n\ln 2} = \frac{3}{\ln 2}\lim_{n\to\infty} \frac{1}{\sqrt[3]{n}} = 0$
Stąd $\lg n = O\left(\sqrt[3]{n}\right)$ oraz $\neg\left(\lg n = \Omega\left(\sqrt[3]{n}\right)\right)$.
Dodatkowo, z lematu 2 (c) wynika, że $\sqrt[3]{n} = \Omega(\lg n)$ i $\neg(\sqrt[3]{n} = O(\lg n))$.
(a) $\left(\sqrt[3]{n}\right)' = \frac{1}{3\sqrt[3]{n^2}}$, $(\log_a n)' = \frac{1}{n\ln a}$ dla $a > 0, a \neq 1, n \neq 0$

(3) Należy pokazać, że $\sqrt[3]{n} = O(\sqrt{n})$ i $\neg\left(\sqrt[3]{n} = \Omega\left(\sqrt{n}\right)\right)$. Korzystam z lematu 1.
$\lim_{n\to\infty} \frac{\sqrt[3]{n}}{\sqrt{n}} = \lim_{n\to\infty} \frac{1}{\sqrt[6]{n}} = 0$, stąd $\sqrt[3]{n} = O\left(\sqrt{n}\right)$ i $\neg\left(\sqrt[3]{n} = \Omega\left(\sqrt{n}\right)\right)$
Dodatkowo, z lematu 2 (c) wynika, że $\sqrt{n} = \Omega\left(\sqrt[3]{n}\right)$ i $\neg\left(\sqrt{n} = O\left(\sqrt[3]{n}\right)\right)$.

(4) Należy pokazać, że $\sqrt{n} = O(n)$ i $\neg\left(\sqrt{n} = \Omega(n)\right)$. Korzystam z lematu 1.

$\lim_{n\to\infty} \frac{\sqrt{n}}{n} = \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$, stąd $\sqrt{n} = O(n)$ i $\neg\left(\sqrt{n} = \Omega(n)\right)$

Dodatkowo, z lematu 2 (c) wynika, że $n = \Omega\left(\sqrt{n}\right)$ i $\neg\left(n = O\left(\sqrt{n}\right)\right)$.

(5) Należy pokazać, że $n = O(n\lg n)$ i $\neg\left(n = \Omega(n\lg n)\right)$.

$\lim_{n\to\infty} \frac{n}{n\lg n} = \lim_{n\to\infty} \frac{1}{\lg n} = 0$, stąd $n = O(n\lg n)$ i $\neg\left(n = \Omega(n\lg n)\right)$.

Dodatkowo, z lematu 2 (c) wynika, że $n\lg n = \Omega(n)$ i $\neg(n\lg n = O(n))$.

(6) Należy pokazać, że $n\lg n = O(n^2)$ i $\neg\left(n\lg n = \Omega(n^2)\right)$.

$\lim_{n\to\infty} \frac{n\lg n}{n^2} = \lim_{n\to\infty} \frac{\lg n}{n} =^{\left[\frac{\infty}{\infty}\right]} \lim_{n\to\infty} \frac{1}{n\ln 2} = 0$

Stąd $n\lg n = O(n^2)$ i $\neg\left(n\lg n = \Omega(n^2)\right)$.

Dodatkowo, z lematu 2 (c) wynika, że $n^2 = \Omega(n\lg n)$ i $\neg(n^2 = O(n\lg n))$.

(7) Należy pokazać, że $n^2 = O(n^3)$ i $\neg\left(n^2 = \Omega(n^3)\right)$. Korzystam z lematu 1.

$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{1}{n} = 0$, stąd $n^2 = O(n^3)$ i $\neg\left(n^2 = \Omega(n^3)\right)$.

Dodatkowo, z lematu 2 (c) wynika, że $n^3 = \Omega(n^2)$ i $\neg(n^3 = O(n^2))$.

(8) Należy pokazać, że $n^3 = O(2^n)$ i $\neg\left(n^3 = \Omega(2^n)\right)$.

Korzystam z twierdzenia 2. $a_n = \frac{n^3}{2^n} \neq 0$, $a_{n+1} = \frac{(n+1)^3}{2^{(n+1)}}$,

$\lim_{n\to\infty} \left|\frac{(n+1)^3}{2^{(n+1)}} \cdot \frac{2^n}{n^3}\right| = \lim_{n\to\infty} \left|\frac{n^3+3n^2+3n+1}{2n^3}\right| = \frac{1}{2} < 1$, więc $\lim_{n\to\infty} \frac{n^3}{2^n} = 0$

Stąd (na mocy lematu 1) $n^3 = O(2^n)$ i $\neg(n^3 = \Omega(2^n))$.

Dodatkowo, z lematu 2 (c) wynika, że $2^n = \Omega(n^3)$ i $\neg(2^n = O(n^3))$.

(9) Należy pokazać, że $2^n = O(3^n)$ i $\neg(2^n = \Omega(3^n))$. Korzystam z lematu 1.

$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$, stąd $2^n = O(3^n)$ i $\neg(2^n = \Omega(3^n))$

Dodatkowo, z lematu 2 (c) wynika, że $3^n = \Omega(2^n)$ i $\neg(3^n = O(2^n))$.

(10) Należy pokazać, że $3^n = O(n!)$ i $\neg(3^n = \Omega(n!))$. Korzystam z twierdzenia 2.

$a_n = \frac{3^n}{n!} \neq 0$, $a_{n+1} = \frac{3^{(n+1)}}{(n+1)!}$, $\lim_{n\to\infty} \left|\frac{3^{(n+1)}}{(n+1)!} \cdot \frac{n!}{3^n}\right| = \lim_{n\to\infty} \left|\frac{3}{(n+1)}\right| = 0 < 1$,

więc $\lim_{n\to\infty} \frac{3^n}{n!} = 0$. Stąd (na mocy lematu 1) $3^n = O(n!)$ i $\neg(3^n = \Omega(n!))$.

Dodatkowo, z lematu 2 (c) wynika, że $n! = \Omega(3^n)$ i $\neg(n! = O(3^n))$.

(11) Należy pokazać, że $n! = O(n^n)$ i $\neg(n! = \Omega(n^n))$.

Korzystam z twierdzenia 2. $a_n = \frac{n!}{n^n} \neq 0$, $a_{n+1} = \frac{(n+1)!}{(n+1)^{n+1}}$,

$\lim_{n\to\infty} \left|\frac{(n+1)!}{(n+1)^{n+1}} \cdot \frac{n^n}{n!}\right| = \lim_{n\to\infty} \left|\left(\frac{n}{n+1}\right)^n\right| = \lim_{n\to\infty} \frac{1}{\left(1+\frac{1}{n}\right)^n} =^{(a)} \frac{1}{e} < 1$,

więc $\lim_{n\to\infty} \frac{n!}{n^n} = 0$, a stąd (na mocy lematu 1) wynika, że $n! = O(n^n)$

i $\neg(n! = \Omega(n^n))$.

Dodatkowo, z lematu 2 (c) wynika, że $n^n = \Omega(n!)$ i $\neg(n^n = O(n!))$.

(a) $\lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = e$, gdzie $e \approx 2.71828$

[02] Sprawdź, które z poniższych równości są prawdziwe, a które fałszywe.

(1) $\lg(n!) = O(n\lg n)$ (2) $\lg^2 n = O(\lg 2^n)$

(3) $(\sqrt{2})^{\lg n} = \Theta(\sqrt{n})$ (4) $2^{2n} = O(2\cdot 2^n)$

(5) $3^n = \Omega(2^n)$ (6) $(\sqrt{n}+6)^4 = \Theta(n^2)$

Rozwiązanie:

(1) $\lg(n!) = O(n\lg n)$?

$\lg(n!) = \lg(n\cdot(n-1)\cdot \ldots \cdot 2\cdot 1) = \underbrace{\lg n + \lg(n-1) + \cdots + \lg 2 + \lg 1}_{n \text{ składników}} \leq n\lg n,$

zatem $\lg(n!) = O(n\lg n)$

(2) $\lg^2 n = O(\lg 2^n)$?

$\lg 2^n = n$

$\lim\limits_{n\to\infty} \frac{\lg^2 n}{n} =^{[\frac{\infty}{\infty}],(a)} \lim\limits_{n\to\infty} \frac{2\lg n}{n\ln 2} = \underbrace{\frac{2}{\ln 2}}_{c_1} \lim\limits_{n\to\infty} \frac{\lg n}{n} =^{[\frac{\infty}{\infty}]} c_1 \lim\limits_{n\to\infty} \frac{1}{n\ln 2} = 0$, zatem

$\lg^2 n = O(\lg 2^n)$

(a) $(\lg^2 n)' = \frac{2\lg n}{n\ln 2}$

(3) $(\sqrt{2})^{\lg n} = \Theta(\sqrt{n})$?

$(\sqrt{2})^{\lg n} = 2^{\frac{1}{2}\lg n} = 2^{\lg\sqrt{n}} = x$

Z definicji 15 mamy $2^{\lg\sqrt{n}} = x$ wtw $\lg\sqrt{n} = \log_2 x$, więc $x = \sqrt{n}$

Stąd $(\sqrt{2})^{\lg n} = x = \sqrt{n}$, więc $(\sqrt{2})^{\lg n} = \Theta(\sqrt{n})$ // $\lg x$ ozn. $\log_2 x$

(4) $2^{2n} = O(2\cdot 2^n)$?

$\lim\limits_{n\to\infty} \frac{2^{2n}}{2\cdot 2^n} = \lim\limits_{n\to\infty} \frac{2^n\cdot 2^n}{2\cdot 2^n} = \lim\limits_{n\to\infty} 2^{n-1} = \infty$, zatem $2^{2n} \neq O(2\cdot 2^n)$

(5) $3^n = \Omega(2^n)$?

$\lim\limits_{n\to\infty} \frac{3^n}{2^n} = \lim\limits_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$, więc $3^n = \Omega(2^n)$

(6) $(\sqrt{n}+6)^4 = \Theta(n^2)$?

$\lim\limits_{n\to\infty} \frac{(\sqrt{n}+6)^4}{n^2} = \lim\limits_{n\to\infty} \left(\frac{(\sqrt{n}+6)^2}{n}\right)^2 = \lim\limits_{n\to\infty} \left(\frac{n+12\sqrt{n}+36}{n}\right)^2 = \lim\limits_{n\to\infty} \left(1+\frac{12}{\sqrt{n}}+\frac{36}{n}\right)^2 = 1 \in R_+$

Stąd $(\sqrt{n}+6)^4 = \Theta(n^2)$.

[03] Niech A będzie algorytmem, którego złożoność czasowa jest określona funkcją $T(\mathrm{A}, n)$ dla danych rozmiaru n. Przypuśćmy, że czas potrzebny do wykonania algorytmu A na komputerze C1 dla danych rozmiaru n jest równy t.

(1) Jaki jest czas potrzebny do wykonania tego algorytmu dla danych 10 razy większych?

(2) Jaki jest rozmiar zadania, które można rozwiązać przy pomocy algorytmu A w czasie 10 razy większym?

$T(\mathrm{A}, n)$	n	$\lg n$	n^2	n^3	2^n
czas					
rozmiar zadania					

Rozwiązanie:

(1) $T(\mathrm{A}, n) = n = t$
$T(\mathrm{A}, 10n) = 10n = 10t$
$T(\mathrm{A}, n) = \lg n = t$
$T(\mathrm{A}, 10n) = \lg(10n) = \lg 10 + \lg n \approx 3.3 + t$
$T(\mathrm{A}, n) = n^2 = t$
$T(\mathrm{A}, 10n) = (10n)^2 = 100t$
$T(\mathrm{A}, n) = n^3 = t$
$T(\mathrm{A}, 10n) = (10n)^3 = 1000t$
$T(\mathrm{A}, n) = 2^n = t$
$T(\mathrm{A}, 10n) = 2^{10n} = (2^n)^{10} = t^{10}$

(2) $T(\mathrm{A}, n) = n = t$
$T(\mathrm{A}, x) = x = 10t = 10n \quad \Rightarrow x = 10n$
$T(\mathrm{A}, n) = \lg n = t$
$T(\mathrm{A}, x) = \lg x = 10t = 10\lg n \Rightarrow x = n^{10}$
$T(\mathrm{A}, n) = n^2 = t$
$T(\mathrm{A}, x) = x^2 = 10t = 10n^2 \Rightarrow x = \sqrt{10}n$
$T(\mathrm{A}, n) = n^3 = t$
$T(\mathrm{A}, x) = x^3 = 10t = 10n^3 \Rightarrow x = \sqrt[3]{10}n$
$T(\mathrm{A}, n) = 2^n = t$
$T(\mathrm{A}, x) = 2^x = 10t = 10{\cdot}2^n \Rightarrow x = \log_2(10{\cdot}2^n) =$
$= \log_2 10 + \log_2 2^n = n + \log_2 10$

[04] Wyznacz złożoność czasową funkcji B1 i określ rząd wielkości uzyskanej funkcji złożoności. Za operację elementarną przyjmij porównanie elementu tablicy T i wartości x.

```
void B1(int **T, int n) {
 int x = 0;
 for(int i = 1; i <= n – 1; i++)
  for(int j = 1; j <= n – i; j++)
   for(int s = j; s <= i + j – 1; s++)
    if(T[j][s] < x) x = T[j][s];
}
```

Rozwiązanie:

$$T(\mathrm{B1}, n) = \sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-i}\left(\sum_{s=j}^{i+j-1} 1\right)\right) = \sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-i} i\right) = \sum_{i=1}^{n-1}(i(n-i)) =$$

$$= \sum_{i=1}^{n-1}(i{\cdot}n) - \sum_{i=1}^{n-1} i^2 = \frac{n^2(n-1)}{2} - \sum_{i=1}^{n-1} i^2 =^{(a)} \frac{n^2(n-1)}{2} - \frac{(n-1)n(2n-1)}{6} =$$

$$= \frac{n(n-1)(3n-2n+1)}{6} = \frac{n(n-1)(n+1)}{6} = \Theta(n^3)$$

(a) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, więc $\sum_{i=1}^{n-1} i^2 = \frac{(n-1)n(2n-1)}{6}$

[05] Jak liczny jest następujący ciąg: n, $\frac{n}{2}$, $\frac{n}{4}$, $\frac{n}{8}$, …, 2, 1 przy założeniu, że $n = 2^k$ dla $k \in N$?

Rozwiązanie:

n	$\frac{n}{2}$	$\frac{n}{4}$	$\frac{n}{8}$	…	2	1
2^k	2^{k-1}	2^{k-2}	2^{k-3}	…	2^1	2^0

$n = 2^k \Leftrightarrow \lg_2 n = k$, stąd dla $n = 2^k$ takich liczb jest $1 + \lg_2 n$

[06] Zaproponuj możliwie najefektywniejszy czasowo algorytm wyznaczający ostatnią cyfrę wartości $n!$. Określ pesymistyczną złożoność czasową podanego rozwiązania.

Rozwiązanie:

0!=1
1!=1
2!=2
3!=6
4!=24
5!=120
$T(\mathrm{B2}, n) = 4 = \Theta(1)$

```
int B2(int n) {
  if(n < 2) return 1;
  if(n == 2) return 2;
  if(n == 3) return 6;
  if(n == 4) return 4;
  return 0;
}
```

[07] Dana jest specyfikacja algorytmu obliczającego wartość $y = x^n$ dla $n, x \in N_+$ i propozycja algorytmu rozwiązującego ten problem. Czy podany algorytm jest poprawny ze względu na zadaną specyfikację?

WP: $n, x \in N_+$
WK: $y = x^n$

```
int Pow1(int x, int n) {
  int y = 1;
  while(n) {
    if(n % 2) y *= x;
    n /= 2;
    x *= x;
  }
  return y;
}
```

Rozwiązanie: Algorytm Pow1 jest poprawny całkowicie ze względu na podaną specyfikację. Przy dowolnych danych wejściowych spełniających WP program zawsze się zatrzymuje, a zwracane wartości to $y = x^n$.

[08] Dana jest specyfikacja algorytmu obliczającego sumę kwadratów kolejnych liczb nieparzystych nie większych niż n i propozycja algorytmu rozwiązującego ten problem. Czy podany algorytm jest poprawny częściowo (całkowicie) ze względu na zadaną specyfikację? Odpowiedź uzasadnij.

WP: $n \in N_+$

WK: s = suma kwadratów kolejnych liczb nieparzystych $\leq n$

```
int B3(int n) {
  int i = 1, s = 1;
  while(i != n) { i += 2; s += (i * i); }
  return s;
}
```

Rozwiązanie: Program jest poprawny częściowo, ale nie jest poprawny całkowicie ze względu na zadaną specyfikację. Dla n parzystych pętla się nie zatrzymuje, ale dla n nieparzystych algorytm kończy obliczenia i zwraca wartość zgodną z WK.

Uzasadnienie: Fakt, że dla n parzystych pętla się nie zatrzymuje, jest oczywisty, ponieważ wartość i zawsze będzie nieparzysta. Fakt, że dla n nieparzystych algorytm B3 zawsze kończy obliczenia i zwraca wartość zgodną z WK uzasadnimy indukcyjnie. Dla $n = 1$ wyrażenie kontrolne pętli while od razu daje wartość false i algorytm zwraca wynik $s = 1$. Załóżmy, że dla $n = 2 \cdot t + 1$ (gdzie $t \in N$) algorytm B3 wyznaczy sumę $s = \sum_{i=0}^{t} (2 \cdot i + 1)^2$. Pokażemy, że wówczas dla $n = 2 \cdot (t + 1) + 1$ algorytm B3 zwróci wynik $s = \sum_{i=0}^{t+1} (2 \cdot i + 1)^2$. W momencie, gdy sprawdzane jest wyrażenie kontrolne $(i != n)$ pętli dla $i = 2 \cdot t + 1$, mamy wyznaczoną sumę $s = \sum_{i=0}^{t} (2 \cdot i + 1)^2$ (z założenia indukcyjnego). Wyrażenie kontrolne $(i != n)$ zwróci wartość true, ponieważ $n = 2 \cdot (t + 1) + 1$. Stąd wewnątrz pętli wykonane zostaną instrukcje, w wyniku których uzyskamy $i = 2 \cdot (t + 1) + 1$ oraz $s = \sum_{i=0}^{t} (2 \cdot i + 1)^2 + (2 \cdot (t + 1) + 1)^2 = \sum_{i=0}^{t+1} (2 \cdot i + 1)^2$. W tym momencie wyrażenie kontrolne pętli while zwróci wartość false i algorytm B3 ostatecznie zakończy działanie z wynikiem $s = \sum_{i=0}^{t+1} (2 \cdot i + 1)^2$. Na mocy zasady indukcji matematycznej możemy stwierdzić, że dla n nieparzystych algorytm B3 zawsze kończy obliczenia i zwraca wartość zgodną z WK.

[09] Uporządkuj niemalejąco poniższy ciąg funkcji według ich rzędów. Odpowiedź uzasadnij.

$f_1(n) = \frac{3}{4}n^2 + 7n + 5$, $f_2(n) = \log_2 n^{20}$, $f_3(n) = \log_2(n!)$,

$f_4(n) = |\sin(n!)|$, $f_5(\mathrm{n}) = \sqrt{n}$

Rozwiązanie:

$f_1(n) = \frac{3}{4}n^2 + 7n + 5 = \Theta(n^2)$ – na mocy lematu 2 (a)

$f_2(n) = \log_2 n^{20} = 20 \cdot \log_2 n = \Theta(\log_2 n)$

$f_3(n) = \log_2(n!) = \mathrm{O}(n \lg n)$ – patrz zadanie [02] (1)

$f_4(n) = |\sin(n!)| = \Theta(1)$ – dla ilustracji wystarczy narysować wykresy następujących funkcji: $f(x) = |\sin(x)| + 1$, $g_1(x) = 1$ i $g_2(x) = 2$

$f_5(n) = \sqrt{n} = \Theta(\sqrt{n})$

Ostatecznie, na mocy lematu 2 (b), podane funkcje uporządkowano niemalejąco ze względu na ich rząd wielkości: $f_4(n)$, $f_2(n)$, $f_5(n)$, $f_3(n)$, $f_1(n)$.

[10] Uporządkuj ze względu na rosnący rząd wielkości następujące funkcje:

$f_1(n) = n^2 \lg n$ $\quad f_2(n) = n^{\frac{3}{2}}/\lg n$ $\quad f_3(n) = 0.001n^2$

$f_4(n) = n^{\frac{3}{4}} + \lg n$ $\quad f_5(n) = n^3/\sqrt{n - \lg n}$ $\quad f_6(n) = n^2 \lg n + n\sqrt{n^2 + n^{\frac{3}{2}}}$

$f_7(n) = 4^{\lg n}$ $\quad f_8(n) = \left(\sqrt{2}\right)^{\lg n}$ $\quad f_9(n) = 2^{2n}$

$f_{10}(n) = \lg(n!)$ $\quad f_{11}(n) = \lg(n^{10})$ $\quad f_{12}(n) = 3^{4+n}$

Rozwiązanie:

$f_1(n) = n^2 \lg n = \Theta(n^2 \lg n)$ (7)

$f_2(n) = n^{\frac{3}{2}}/\lg n = \Theta\left(\frac{n\sqrt{n}}{\lg n}\right)$ (5)

$f_3(n) = 0.001n^2 = \Theta(n^2)$ (6)

$f_4(n) = n^{\frac{3}{4}} + \lg n = \Theta(n^{\frac{3}{4}})$ (3)

$f_5(n) = n^3/\sqrt{n - \lg n} = \Theta(n^{\frac{5}{2}})$ (8)

$f_6(n) = n^2 \lg n + n\sqrt{n^2 + n^{\frac{3}{2}}} = \Theta(n^2 \lg n)$ (7)

$f_7(n) = 4^{\lg n} = \Theta(n^2)$ (6)

$f_8(n) = (\sqrt{2})^{\lg n} = \Theta(\sqrt{n})$ (2)

$f_9(n) = 2^{2n} = \Theta(4^n)$ (10)

$f_{10}(n) = \lg(n!) = \Theta(n \lg n)$ (4)

$f_{11}(n) = \lg(n^{10}) = \Theta(\lg n)$ (1)

$f_{12}(n) = 3^{4+n} = \Theta(3^n)$ (9)

Uzasadnienie:

- $f_4(n) = n^{\frac{3}{4}} + \lg n$. Korzystamy z lematu 1.

  $\lim\limits_{n\to\infty} \frac{n^{\frac{3}{4}}}{\lg n} \overset{[\frac{\infty}{\infty}],(a)}{=} \lim\limits_{n\to\infty} \left(\frac{\frac{3}{4} n \ln 2}{\sqrt[4]{n}}\right) = \underbrace{\frac{3}{4}\ln 2}_{c_1} \lim\limits_{n\to\infty} \frac{n}{\sqrt[4]{n}} = c_1 \lim\limits_{n\to\infty} \sqrt[4]{n^3} = \infty$, zatem

  $\lg n = O\left(n^{\frac{3}{4}}\right)$, stąd $f_4(n) = \Theta\left(n^{\frac{3}{4}}\right)$

  (a) $\left(n^{\frac{3}{4}}\right)' = \frac{3}{4} n^{-\frac{1}{4}}$, $(\lg n)' = \frac{1}{n \ln 2}$

- $f_7(n) = 4^{\lg n}$

  $4^{\lg n} = 2^{2 \lg n} = 2^{\lg n^2} = n^2$, zatem $f_7(n) = \Theta(n^2)$

- $f_8(n) = (\sqrt{2})^{\lg n}$

  $(\sqrt{2})^{\lg n} = 2^{\frac{1}{2}\lg n} = 2^{\lg\sqrt{n}} = x$

  Z definicji 15 wynika, że $2^{\lg\sqrt{n}} = x$ wtw $\log_2\sqrt{n} = \log_2 x$, więc $x = \sqrt{n}$.
  Ostatecznie $f_8(n) = \Theta(\sqrt{n})$.
- $n\lg n = O\left(\frac{n\sqrt{n}}{\lg n}\right)$? Korzystamy z lematu 1.

  $\lim_{n\to\infty} \frac{\sqrt{n}}{\lg^2 n} =^{[\frac{\infty}{\infty}],(a)} \lim_{n\to\infty} \left(\frac{n\ln 2}{4\sqrt{n}\lg n}\right) = \frac{\ln 2}{4} \lim_{n\to\infty} \frac{\sqrt{n}}{\lg n} =^{[\frac{\infty}{\infty}]} \frac{\ln 2}{4} \lim_{n\to\infty} \frac{n\ln 2}{2\sqrt{n}} =$

  $= \frac{\ln^2 2}{8} \lim_{n\to\infty} \sqrt{n} = \infty$, zatem $\lg n = O\left(\frac{\sqrt{n}}{\lg n}\right)$ i $\neg\left(\lg n = \Omega\left(\frac{\sqrt{n}}{\lg n}\right)\right)$.

  Ostatecznie $n\lg n = O\left(\frac{n\sqrt{n}}{\lg n}\right)$.

  (a) $\left(\sqrt{n}\right)' = \frac{1}{2\sqrt{n}}$, $(\lg^2 n)' = \frac{2\lg n}{n\ln 2}$

[11] Które równości są prawdziwe? Odpowiedź uzasadnij.

(1) $2^{n+1} = O(2^n)$ (2) $(n+1)^2 = O(n^2)$

(3) $2^{2n} = O(2^n)$ (4) $(\lg n)^{73} = O(\sqrt{n})$

(5) $40^n = O(n!)$

Rozwiązanie:

(1) $2^{n+1} = 2\cdot 2^n = \Theta(2^n)$, więc $2^{n+1} = O(2^n)$

(2) $(n+1)^2 = n^2 + 2n + 1 = \Theta(n^2)$, więc $(n+1)^2 = O(n^2)$

(3) $2^{2n} = 4^n = \Theta(4^n)$, więc $2^{2n} \neq O(2^n)$ – patrz lemat 2 (b)

(4) $\lim_{n\to\infty} \frac{(\lg n)^{73}}{\sqrt{n}} =^{[\frac{\infty}{\infty}],(a)} \lim_{n\to\infty} \left(\frac{73(\lg n)^{72} 2\sqrt{n}}{n\ln 2}\right) = \underbrace{\frac{2\cdot 73}{\ln 2}}_{c_1} \lim_{n\to\infty} \left(\frac{(\lg n)^{72}}{\sqrt{n}}\right) =^{[\frac{\infty}{\infty}]}$

$= c_1 \lim_{n\to\infty} \left(\frac{72(\lg n)^{71} 2\sqrt{n}}{n\ln 2}\right) = \underbrace{\frac{2\cdot 72 c_1}{\ln 2}}_{c_2} \lim_{n\to\infty} \left(\frac{(\lg n)^{71}}{\sqrt{n}}\right) =^{[\frac{\infty}{\infty}]} c_2 \lim_{n\to\infty} \left(\frac{71(\lg n)^{70} 2\sqrt{n}}{n\ln 2}\right) =$

$= \ldots = c_{72} \lim_{n\to\infty} \frac{\lg n}{\sqrt{n}} = 0$, więc $(\lg n)^{73} = O(\sqrt{n})$

(a) $(\lg n)' = \frac{1}{n\ln 2}$, $((\lg n)^{73})' = \frac{73(\lg n)^{72}}{n\ln 2}$, $\left(\sqrt{n}\right)' = \frac{1}{2\sqrt{n}}$

(5) Korzystam z twierdzenia 2.

$a_n = \frac{40^n}{n!} \neq 0$, $a_{n+1} = \frac{40^{n+1}}{(n+1)!}$, $\lim_{n\to\infty} \left|\frac{40^{n+1}}{(n+1)!} \cdot \frac{n!}{40^n}\right| = \lim_{n\to\infty} \left|\frac{40}{(n+1)}\right| = 0 < 1$, więc

$\lim_{n\to\infty} \frac{40^n}{n!} = 0$, stąd $40^n = O(n!)$

[12] Proszę sprawdzić, które z poniższych równości są fałszywe, a które prawdziwe.

(1) $2^{2n} = O(2\cdot 2^n)$ (2) $2^{2+n} = O(2^n)$

(3) $3^{1+n} = O(2^{3+n})$ (4) $3^{n+2} = \Omega(2^n)$

(5) $2^{10+n} = \Theta(2^n)$
(6) $n! = O(2^n)$
(7) $1^{n+1} = O(1)$
(8) $\sqrt{n} = \Omega(\lg n)$
(9) $2^{\lg n} = \Theta(n)$
(10) $(n+1)^2 = O(n^2)$
(11) $(\sqrt{n}+6)^4 = O(n^2)$
(12) $(2n)! = O(n!)$
(13) $(\lg n)^{74} = O(\sqrt{n})$
(14) $\lg n^n = O(\sqrt{n}\lg n)$
(15) $(\lg n)^2 = O(\lg 2^n)$
(16) $n! = O(n^n)$
(17) $n^4 \cdot \lg^6 n \cdot 3^n = O(n^2 \cdot \lg^4 n \cdot 4^n)$

Rozwiązanie:

(1) $2^{2n} = O(2 \cdot 2^n)$ [–]
(2) $2^{2+n} = O(2^n)$ [+]
(3) $3^{1+n} = O(2^{3+n})$ [–]
(4) $3^{n+2} = \Omega(2^n)$ [+]
(5) $2^{10+n} = \Theta(2^n)$ [+]
(6) $n! = O(2^n)$ [–]
(7) $1^{n+1} = O(1)$ [+]
(8) $\sqrt{n} = \Omega(\lg n)$ [+]
(9) $2^{\lg n} = \Theta(n)$ [+]
(10) $(n+1)^2 = O(n^2)$ [+]
(11) $(\sqrt{n}+6)^4 = O(n^2)$ [+]
(12) $(2n)! = O(n!)$ [–]
(13) $(\lg n)^{74} = O(\sqrt{n})$ [+]
(14) $\lg n^n = O(\sqrt{n}\lg n)$ [–]
(15) $(\lg n)^2 = O(\lg 2^n)$ [+]
(16) $n! = O(n^n)$ [+]
(17) $n^4 \cdot \lg^6 n \cdot 3^n = O(n^2 \cdot \lg^4 n \cdot 4^n)$ [+]

Uzasadnienie:

(3) $3^{1+n} = O(2^{3+n})$?

$\lim_{n\to\infty} \frac{3 \cdot 3^n}{2^3 \cdot 2^n} = \underbrace{\frac{3}{2^3}}_{c_1} \lim_{n\to\infty} \frac{3^n}{2^n} = c_1 \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$, zatem $3^{1+n} \neq O(2^{3+n})$

(6) $n! = O(2^n)$? Korzystam z twierdzenia 2.

$a_n = \frac{2^n}{n!} \neq 0$, $a_{n+1} = \frac{2^{n+1}}{(n+1)!}$, $\lim_{n\to\infty} \left|\frac{2^{n+1}}{(n+1)!} \cdot \frac{n!}{2^n}\right| = \lim_{n\to\infty} \left|\frac{2}{(n+1)}\right| = 0 < 1$,

więc $\lim_{n\to\infty} \frac{2^n}{n!} = 0$, stąd $n! \neq O(2^n)$

(12) $(2n)! = O(n!)$?

$\lim_{n\to\infty} \frac{(2n)!}{n!} = \lim_{n\to\infty} \frac{n! \cdot (n+1) \cdot (n+2) \cdot \ldots \cdot 2n}{n!} = \infty$, więc $(2n)! \neq O(n!)$

(13) $(\lg n)^{74} = O(\sqrt{n})$ – patrz zadanie [11] (4)
(15) $(\lg n)^2 = O(\lg 2^n)$ – patrz zadanie [02] (2)
(16) $n! = O(n^n)$? Korzystam z twierdzenia 2.

$a_n = \frac{n!}{n^n} \neq 0$, $a_{n+1} = \frac{(n+1)!}{(n+1)^{n+1}}$,

$\lim_{n\to\infty} \left|\frac{(n+1)!}{(n+1)^{n+1}} \cdot \frac{n^n}{n!}\right| = \lim_{n\to\infty} \left|\left(\frac{n}{n+1}\right)^n\right| = \lim_{n\to\infty} \frac{1}{\left(1+\frac{1}{n}\right)^n} =^{(a)} \frac{1}{e} < 1$,

więc $\lim_{n\to\infty} \frac{n!}{n^n} = 0$, a stąd wynika, że $n! = O(n^n)$

(a) $\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = \mathrm{e}$, gdzie $\mathrm{e}\approx 2.71828$

(17) $n^4\cdot\lg^6 n\cdot 3^n = \mathrm{O}(n^2\cdot\lg^4 n\cdot 4^n)$

$$\lim_{n\to\infty}\frac{n^4\cdot\lg^6 n\cdot 3^n}{n^2\cdot\lg^4 n\cdot 4^n} = \lim_{n\to\infty}\frac{n^2\cdot\lg^2 n}{\left(\frac{4}{3}\right)^n} =^{[\frac{\infty}{\infty}],(a),(b)} \lim_{n\to\infty}\left(\frac{2n\cdot\lg^2 n+n^2\cdot 2\lg n\cdot\frac{1}{n\ln 2}}{\left(\frac{4}{3}\right)^n\ln\frac{4}{3}}\right) =$$

$$= \underbrace{\frac{2}{\ln\frac{4}{3}}}_{c_1}\lim_{n\to\infty}\left(\frac{n\cdot\lg^2 n+n\cdot\lg n\cdot\frac{1}{\ln 2}}{\left(\frac{4}{3}\right)^n}\right) =^{[\frac{\infty}{\infty}],(c)} c_1\lim_{n\to\infty}\frac{\left(\lg^2 n+n\cdot 2\lg n\cdot\frac{1}{n\ln 2}\right)+\left(\frac{1}{\ln 2}\left(\lg n+\frac{1}{\ln 2}\right)\right)}{\left(\frac{4}{3}\right)^n\ln\frac{4}{3}} =$$

$$= \underbrace{\frac{c_1}{\ln\frac{4}{3}}}_{c_2}\lim_{n\to\infty}\frac{\left(\lg^2 n+2\lg n\cdot\frac{1}{\ln 2}\right)+\left(\frac{1}{\ln 2}\left(\lg n+\frac{1}{\ln 2}\right)\right)}{\left(\frac{4}{3}\right)^n} =^{[\frac{\infty}{\infty}],(d)} c_2\lim_{n\to\infty}\frac{\frac{2\lg n}{n\ln 2}+\frac{3}{n(\ln 2)^2}}{\left(\frac{4}{3}\right)^n\ln\frac{4}{3}} =$$

$$= \underbrace{\frac{c_2}{\ln 2\cdot\ln\frac{4}{3}}}_{c_3}\lim_{n\to\infty}\frac{2\lg n+\frac{3}{\ln 2}}{n\cdot\left(\frac{4}{3}\right)^n} = 0, \text{ zatem } n^4\cdot\lg^6 n\cdot 3^n = \mathrm{O}(n^2\cdot\lg^4 n\cdot 4^n)$$

(a) $(n^2\cdot\lg^2 n)' = 2n\cdot\lg^2 n + n^2\cdot 2\lg n\cdot\frac{1}{n\ln 2}$

(b) $\left(\left(\frac{4}{3}\right)^n\right)' =^{(e)} \left(\frac{4}{3}\right)^n\ln\frac{4}{3}$

(c) $\left(n\cdot\lg^2 n + n\cdot\lg n\cdot\frac{1}{\ln 2}\right)' = \lg^2 n + n\cdot 2\lg n\cdot\frac{1}{n\ln 2} + \frac{1}{\ln 2}\left(\lg n + \frac{n}{n\ln 2}\right)$

(d) $\left(\lg^2 n + 2\lg n\cdot\frac{1}{\ln 2} + \frac{1}{\ln 2}\left(\lg n + \frac{1}{\ln 2}\right)\right)' = \frac{2\lg n}{n\ln 2} + \frac{3}{n(\ln 2)^2}$

(e) $(\mathrm{a}^x)' = \mathrm{a}^x\ln\mathrm{a}$ dla $\mathrm{a}\in R_+$, $x\in R$

[13] Jeśli wiadomo, że zachodzą równości $f(n) = \mathrm{O}(\sqrt{n})$ i $g(n) = \mathrm{O}(\lg n)$, to czy wtedy prawdziwe są poniższe zależności?

(1) $f(n) + g(n) = \Theta(n)$ (2) $f(n) + g(n) = \mathrm{O}(\sqrt{n})$

(3) $f(n) + g(n) = \mathrm{O}(\lg n)$ (4) $f(n) = \Omega(g(n))$

Rozwiązanie:

(1) [–], (2) [+], (3) [–],

(4) [–], ponieważ np. $f(n) = 1 = \mathrm{O}(\sqrt{n})$, $g(n) = \lg n = \mathrm{O}(\lg n)$ i $f(n) \neq \Omega(g(n))$

[14] Niech A będzie algorytmem o kwadratowej złożoności czasowej. Przy pomocy tego algorytmu można rozwiązać problem o rozmiarze 100 w ciągu 1 sekundy. Jak duży problem można rozwiązać przy pomocy tego algorytmu w czasie 1 godziny?

Rozwiązanie:

$T(\mathrm{A}, n) = n^2$

$T(\mathrm{A}, 100) = 100^2 = 1\mathrm{s}$ | $100^2 = 1\mathrm{s}$ | $(100\cdot 60)^2\mathrm{s} = x^2\mathrm{s}$

$T(\mathrm{A}, x) = x^2 = 3600\mathrm{s}$ | $x^2 = 3600\mathrm{s}$ | $x = 6000$

[15] Wyznacz złożoności czasowe funkcji f1, f2, f3, f4 oraz określ rzędy wielkości uzyskanych funkcji złożoności. Za operację elementarną przyjmij porównanie elementów tablicy.

```
void f1(int n, int *T) {
 int x = 0;
 for(int i = 1; i <= n – 1; i++)
  for(int j = 1; j <= n – 1; j++)
   for(int s = i; s <= i + j – 1; s++)
    if(T[s] < T[j]) x = T[i];
}
```

```
void f2(int n, int **T) {
 int x = 0;
 for(int i = 1; i <= n; i++)
  for(int j = i; j <= n; j++)
   for(int s = j; s <= i + j; s++)
    if(T[j][s] < T[s][j]) x = T[j][s];
}
```

```
void f3(int n, int m, int *T) {
 int x = 0;
 for(int i = 2; i <= n; i++)
  for(int j = 1; j <= m; j++)
   if(!(j % 2))
    if(T[i] < T[j]) x = T[i];
}
```

```
void f4(int k, int m, int *T) {
 int x = 0;
  for(int i = 2; i <= k + 1; i++)
   for(int j = 1; j <= m – 1; j++)
    if(!(j % 2))
     if(T[i] < T[j]) x = T[i];
}
```

Rozwiązanie:

$$T(\mathrm{f1},n)=\sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-1}\left(\sum_{s=i}^{i+j-1}1\right)\right)=\sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-1}j\right)=$$

$$=\sum_{i=1}^{n-1}(1+2+\cdots+(n-1))=$$

$$=\sum_{i=1}^{n-1}\left(\frac{n(n-1)}{2}\right)=\left(\frac{n(n-1)}{2}\right)\sum_{i=1}^{n-1}1=\frac{n(n-1)^2}{2}=\Theta(n^3)$$

$$T(\mathrm{f2},n)=\sum_{i=1}^{n}\left(\sum_{j=i}^{n}\left(\sum_{s=j}^{i+j}1\right)\right)=\sum_{i=1}^{n}\left(\sum_{j=i}^{n}(i+1)\right)=$$

$$=\sum_{i=1}^{n}\left((i+1)\sum_{j=i}^{n}1\right)=\sum_{i=1}^{n}\left((i+1)(n-i+1)\right)=$$

$$=\sum_{i=1}^{n}(ni-i^2+n+1)=n\sum_{i=1}^{n}i-\sum_{i=1}^{n}i^2+(n+1)\sum_{i=1}^{n}1=^{(a)}$$

$$=\frac{n^2(n+1)}{2}-\frac{n(n+1)(2n+1)}{6}+n(n+1)=\frac{(n+1)\left[3n^2-2n^2-n+6n\right]}{6}=$$

$$=\frac{(n+1)n(n+5)}{6}=\Theta(n^3)$$

(a) $\sum_{i=1}^{n}i^2=\frac{n(n+1)(2n+1)}{6}$

$$T(\mathrm{f3},n,m)=\begin{cases}\frac{(n-1)m}{2} & \text{dla } m=2k,k\in N\\ \frac{(n-1)(m-1)}{2} & \text{dla } m=2k+1,k\in N\end{cases}\qquad T(\mathrm{f3},n,m)=\Theta(n\cdot m)$$

$$T(\mathrm{f4},k,m)=\begin{cases}\frac{k(m-2)}{2} & \text{dla } m=2k,k\in N\\ \frac{k(m-1)}{2} & \text{dla } m=2k+1,k\in N\end{cases}\qquad T(\mathrm{f4},k,m)=\Theta(k\cdot m)$$

[16] Jaka jest dokładna pesymistyczna złożoność czasowa poniższych algorytmów mierzona liczbą porównań elementów tablicy A i elementu x? Zakładamy, że $n = 2^k$ dla pewnego $k \in N$.

```
void f5(int n, int *A) {
 int i = n, j = 0, x = 0;
 while(i >= 1) {
  j = i;
  while(j <= n) {
   if(A[j] == x) A[i–1] = ++x;
   j *= 2;
  }
  i /= 2;
 }
}
```

```
void f6(int n, int *A) {
 int j = 0, x = 0, m = 0;
 for(int i = 1; i <= n – 1; i++) {
  j = n;
  while(j >= 1) {
   if(x == A[(n–i) % (j+1)]) m = A[i];
   else
    if(x < A[n–i]) m = A[j–1];
   j /= 2;
  }
 }
}
```

Rozwiązanie:
$T_{\max}(\text{f5}, n) = 1 + 2 + \cdots + (1 + \lg n) = \frac{(2+\lg n)(1+\lg n)}{2} = \Theta((\lg n)^2)$ – porównaj zadanie [05]
$T_{\max}(\text{f6}, n) = 2(1 + \lg n)(n - 1) = \Theta(n \lg n)$ – porównaj zadanie [05]

[17] Wyznacz pesymistyczną złożoność czasową poniższego algorytmu. Operacją elementarną jest porównanie elementów tablicy A.

```
void f7(int n, int *A) {
 int p = 0;
 for(int i = 0; i <= n – 2; i++)
  for(int j = i + 1; j <= n – 1; j++)
   if(A[j–1] > A[j]) {
    p = A[j–1]; A[j–1] = A[j]; A[j] = p;
   }
}
```

Rozwiązanie:

$$T(\text{f7}, n) = \sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} 1\right) = \sum_{i=0}^{n-2}(n - i - 1) = (n-1)\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i =$$

$$= (n-1)^2 - \frac{(n-1)(n-2)}{2} = \frac{(n-1)n}{2} = \Theta(n^2)$$

[18] Wyznacz pesymistyczną złożoność czasową algorytmu BubbleSort( ) (sortowanie bąbelkowe).
Idea algorytmu (sortowanie bąbelkowe) (por. [WN])

- Potraktuj tablicę do posortowania A jako pionową kolumnę, w której elementy mniejsze (lżejsze) będziesz wypychać do góry, a elementy większe (cięższe) do dołu.
- W każdej iteracji w trakcie przeglądania tablicy A od góry do dołu zamień miejscami dwa sąsiadujące ze sobą elementy, o ile znajdują się one w niewłaściwej kolejności względem siebie.
- Powtarzaj przechodzenie przez nieposortowany fragment tablicy, przesuwając w każdej iteracji największy element z nieposortowanego fragmentu tablicy na jego dno.

```
void BubbleSort(int n, int *A) {
 int pom = 0;
 for(int i = 0; i < n – 1; i++)
  for(int j = 0; j < n – (i + 1); j++)
   if(A[j] > A[j+1]) {
    pom = A[j];
    A[j] = A[j+1];
    A[j+1] = pom;
   }
}
```

Operacja elementarna – porównanie dwóch elementów tablicy.

$$T(n) = \sum_{i=0}^{n-2} \left(\sum_{j=0}^{n-(i+2)} 1 \right) = \sum_{i=0}^{n-2} (n - i - 1) =$$

$$= (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Stała liczba porównań w przypadku średnim i pesymistycznym.

[19] Wyznacz pesymistyczną złożoność czasową algorytmu InsertSort( ) (sortowanie przez proste wstawianie).
Idea algorytmu (sortowanie przez proste wstawianie) (por. [WN])

- Elementy ciągu są podzielone (umownie) na dwa podciągi: ciąg uporządkowany a_0, ..., a_{i-1} i ciąg nieuporządkowany a_i, ..., a_{n-1}.
- W każdej iteracji (począwszy od $i = 1$ do $i = n - 1$) pierwszy element (tj. a_i) ciągu nieuporządkowanego jest przenoszony do podciągu uporządkowanego na odpowiednią pozycję, tak aby podciąg ten był nadal uporządkowany.

```
void InsertSort(int n, int *A) {
 int pom = 0, j = 0;
 for(int i = 1; i < n; i++) {
  pom = A[i]; j = i – 1;
  while(j >= 0 && A[j] > pom)
   A[j+1] = A[j––];
  A[++j] = pom;
 }
}
```

Operacja elementarna – porównanie dwóch elementów tablicy.
Przypadek pesymistyczny – ciąg uporządkowany w odwrotnej kolejności.

$$T_{\max}(n) = \sum_{i=1}^{n-1}\left(\sum_{j=0}^{i-1} 1\right) = \sum_{i=1}^{n-1} i = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

[20] Wyznacz złożoność czasową algorytmu SelectSort( ) (sortowanie przez proste wybieranie).
Idea algorytmu (sortowanie przez proste wybieranie) (por. [WN])

- W każdej i-tej iteracji ($i = 0$, …, $n-2$) wybieramy element o najmniejszym kluczu spośród elementów a_i, …, a_{n-1}, po czym zamieniamy go z elementem a_i.

```
void SelectSort(int n, int *A) {
 int min = 0, pom = 0;
 for(int i = 0; i < n – 1; i++) {
  min = i;
  for(int j = i + 1; j < n; j++)
   if(A[j] < A[min]) min = j;
  pom = A[i];
  A[i] = A[min];
  A[min] = pom;
 }
}
```

Operacja elementarna – porównanie dwóch elementów tablicy.

$$T(n) = \sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} 1\right) = \sum_{i=0}^{n-2}(n-i-1) =$$

$$= (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Stała liczba porównań w przypadku średnim i pesymistycznym.

[21] Wyznacz złożoność czasową algorytmu CountSort( ) (sortowanie przez zliczanie).
Założenie: Elementy n-elementowego porządkowanego ciągu są liczbami naturalnymi z przedziału od 0 do ustalonego „niedużego" k ($k \in N$) (bądź można je ponumerować kolejnymi liczbami naturalnymi z ustalonego „niedużego" zakresu).
Idea algorytmu (sortowanie przez zliczanie) (por. [CLR])

- Dla każdej liczby x ciągu oblicz, ile jest elementów mniejszych lub równych x.
- Dla każdej wartości $x = \mathrm{A}[i]$ nieuporządkowanego ciągu wykonaj (idąc od końca ciągu): jeśli istnieje y liczb mniejszych bądź równych x, to x umieść na pozycji $y - 1$ i przyjmij, że liczb mniejszych bądź równych x jest teraz $y - 1$.

```
void CountSort(int k, int n, int *A) {
 int* D = new int[k+1];
 int* B = new int[n];
 memset(D, 0, (k + 1) * sizeof(int));
 memset(B, 0, n * sizeof(int));
 for(int i = 0; i < n; i++) D[A[i]]++;
 for(int i = 1; i <= k; i++) D[i] += D[i–1];
 for(int i = n – 1; i >= 0; i––) {
  B[D[A[i]]–1] = A[i];
  D[A[i]]––;
  }
 for(int i = 0; i < n; i++) A[i] = B[i];
 delete [ ]D; delete [ ]B;
}
```

CountSort( ) nie używa porównań. Zliczane są przypisania.

$$T(n,k) = n + k + 2n + n = 4n + k = \Theta(n + k)$$

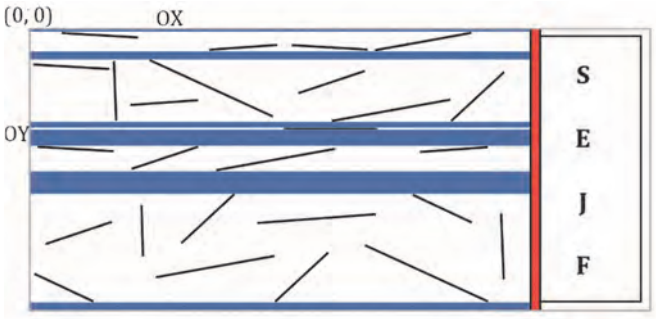
Stała liczba przypisań w przypadku średnim i pesymistycznym.

5.1. Zadania programistyczne

[22] (Prezes Bajt) Prezes firmy XY, pan Bajt, dostał nagrodę Lidera Biznesu za niekwestionowane osiągnięcia finansowe swojej firmy. Tradycyjnie na gali wręczenia nagrody Lider przedstawia krótką prezentację strategii prowadzenia firmy przed kolegami z branży. Pan Bajt pracuje na stanowisku już od n ($10 \le n \le 1000$) miesięcy, więc zlecił przygotowanie szczegółowego zestawienia finansowego opisującego, ile firma zarobiła/straciła w poszczególnych miesiącach. Prezes zdecydował, że opowie jedynie o tym okresie, w którym zarobił najwięcej. Poprosił informatyków o wyselekcjonowanie takiego okresu oraz o podanie kwoty (sumy zysków/strat) wypracowanej w tym czasie. Szukany odcinek czasu musi być nieprzerwany (tzn. pewna liczba bezpośrednio następujących po sobie miesięcy) oraz, co najważniejsze, suma zysków (przy

uwzględnieniu strat) uzyskanych podczas wyselekcjonowanych miesięcy musi być największa z możliwych. Z dokumentacji wynika, że zyski/straty w kolejnych miesiącach, w których Pan Bajt zarządzał firmą to liczby całkowite z przedziału $[-1000000, 1000000]$.
Wskazówka: Dla okresu $n = 12$ miesięcy, w których Pan Bajt zarobił/stracił odpowiednio (6, –10, 5, –1, 4, 8 –3, 9, 11, –40, 6, 18) jednostek, największą sumę uzyskamy dodając zyski/straty z kolejnych miesięcy od 3 do 9. Poprawny wynik to [3, 9, 33], gdzie 33 = 5 + (–1) + 4 + 8 + (–3) + 9 + 11 jednostek. Optymalny algorytm ma złożoność $\Theta(n)$. Każde pole tablicy zawierające zysk/stratę w danym miesiącu może być analizowane dokładnie jeden raz.

[23] (Sejf króla Bajtdocji) Droga do sejfu króla Bajtdocji prowadzi przez korytarz o szerokości n metrów ($n \in N$, $1 \le n \le 10000$) zabezpieczony sprzętem laserowym. Urządzenie stanowi m prętów ($m \in N$, $3 \le m \le 50000$) zamocowanych do sufitu, w które wbudowano emitery laserowe (liczba emiterów na jednym pręcie zależy od jego długości). Działanie urządzenia polega na aktywowaniu i dezaktywowaniu kolejnych prętów. Z każdego uruchomionego pręta emitowana jest przez moment wiązka światła (z losowo wybranego emitera), po czym uruchamiany jest kolejny pręt. Zmiana pręta następuje co 1μs. Kolejność aktywacji prętów jest określona i powtarza się cyklicznie.



Rys. 1.2. Ilustracja do zadania [23]

Aby wyłączyć zabezpieczenie (przy braku znajomości hasła) należy naprowadzić pocisk na dowolne miejsce poziomej (czerwonej) listwy opasającej ścianę sejfu (szerokość sejfu jest równa szerokości korytarza). Zadanie polega na znalezieniu wszystkich możliwych bezpiecznych pasm (na całej szerokości korytarza) dla toru pocisku. Optymalny algorytm ma złożoność $\Theta(m + n)$.

Algorytm powinien:

- Czytać z pliku tekstowego opis korytarza i lokalizację prętów.
- Wyznaczać wszystkie możliwe bezpieczne pasma dla toru pocisku.
- Zapisać wyznaczone pasma w rosnącej (lub malejącej) kolejności do pliku tekstowego.

Dane:

- W pierwszym wierszu pliku znajdują się dwie liczby całkowite n (szerokość korytarza) i m (liczba prętów), oddzielone pojedynczą spacją.
- W kolejnych m liniach umieszczone są po cztery liczby całkowite, oddzielone pojedynczą spacją, reprezentujące współrzędne prętów ($x1_i$, $y1_i$, $x2_i$, $y2_i$). Zakładamy, że $0 \leq x1_i \leq x2_i \leq n$, $0 \leq y1_i \leq y2_i \leq n$. Uwaga: pręt może być zamocowany wzdłuż korytarza.
- Oś OX układu prowadzi od początku korytarza do ściany sejfu. Oś OY – od strony lewej korytarza do prawej.
- Zakładamy, że dowolny emiter z *i*-tego pręta o współrzędnych ($x1_i$, $y1_i$, $x2_i$, $y2_i$) nie może wysyłać wiązki poza pasmem ($y1_i, y2_i$).

Wyjście:

- W kolejnych wierszach pliku należy wypisać w nawiasach pary liczb ($y1_i, y2_i$) (gdzie $y1_i \neq y2_i$) reprezentujące bezpieczne pasma.
- W ostatniej linii należy podać ilość bezpiecznych pasm.

Przykład: Dla $n = 11$, $m = 5$ oraz prętów (2 5 2 6), (4 1 4 4), (4 10 10 10), (1 6 5 9), (3 8 7 9) algorytm powinien zwrócić cztery następujące bezpieczne pasma: (0 1), (4 5), (9 10), (10 11).

Rozdział 2. Rekurencja

1. Definicja rekurencyjna
2. Rodzaje rekurencji
3. Nieuzasadnione użycie rekurencji
4. Metody rozwiązywania rekurencji
5. Sortowanie przez kopcowanie (HeapSort)
6. Zadania różne

Definicje rekurencyjne są używane m.in. do generowania obiektów zbiorów nieskończonych oraz do sprawdzania, czy dany obiekt należy do takiego zbioru. Składają się z dokładnego opisu elementów podstawowych oraz reguł umożliwiających generowanie nowych obiektów na podstawie elementów podstawowych lub tych określonych wcześniej. Mechanizm rekurencji jest wykorzystywany we współczesnych językach programowania. Jedna instancja funkcji R( ) może się odwoływać do innej instancji tej samej funkcji R( ) na różne sposoby. Stąd mamy różne rodzaje rekurencji. Zaletą stosowania rozwiązań rekurencyjnych jest prostota i czytelność zapisywanych definicji. Należy jednak pamiętać, że stosowanie rekurencji w nieuzasadnionych przypadkach prowadzi do zwiększenia złożoności czasowej i pamięciowej.

1. Definicja rekurencyjna (por. [BDR], [DA], [DS])

Definicja 1 (definicja rekurencyjna) (por. [DA])
Definicja rekurencyjna zawiera:

- *warunek brzegowy* (*początkowy*) określający przypadki (lub elementy) podstawowe,
- *reguły* umożliwiające analizę nowych przypadków na podstawie przypadków podstawowych lub tych rozpatrzonych wcześniej (analogicznie – *reguły* umożliwiające generowanie nowych obiektów na podstawie elementów podstawowych lub tych zbudowanych wcześniej).

Zdefiniowanie przynajmniej jednego przypadku (elementu) podstawowego jest *konieczne* do prawidłowego zakończenia rekurencji.

Przykład 1. Rekurencyjna definicja potęgi o podstawie 2 i wykładniku naturalnym $n \in N$.

$$\begin{cases} a_0 = 1 \\ a_n = 2 \cdot a_{n-1} \text{dla } n \in N_+ \end{cases}$$

```
int Pow(int n) {
  if(n == 0) return 1;
  return Pow(n – 1) * 2;
}
```

Stosując powyższą definicję, otrzymujemy następujący ciąg liczb: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, …

Przykład 2. Rekurencyjna definicja n-tego wyrazu ciągu Fibonacciego.
Ciąg Fibonacciego to ciąg liczb, którego pierwsze dwie wartości to 0 i 1, a każda następna liczba jest sumą dwóch poprzedzających ją wartości (ciąg Fibonacciego: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...).

$$F(n) = \begin{cases} n & \text{dla} \quad n \leq 1 \\ F(n-1) + F(n-2) & \text{dla} \quad n > 1 \end{cases}$$

```
int Fib(int n) {
  if(n <= 1) return n;
  return Fib(n – 2) + Fib(n – 1);
}
```

Przykład 3. Algorytm Euklidesa wyznaczający największy wspólny dzielnik (NWD) dwóch liczb całkowitych nieujemnych.
Idea algorytmu (NWD)

Mając do policzenia NWD(a, b), sprawdzamy czy b = 0. Jeśli tak jest, to NWD(a, b) = a, w przeciwnym przypadku wywołujemy rekurencyjnie algorytm dla liczb b i (a % b).
NWD(54, 69) = NWD(69, 54) = NWD(54, 15) = NWD(15, 9) = NWD(9, 6) = = NWD(6, 3) = NWD(3, 0) = 3

```
int NWD(int a, int b) {
  if(b == 0) return a;
  return NWD(b, a % b);
}
```

Przykład 4. (Magiczny kwadrat) Algorytm wpisuje w kwadrat o rozmiarze $n \times n$ tekst T zgodnie z podanym na rysunku wzorem (tzn. po każdorazowym rekurencyjnym wywołaniu funkcji elementy tekstu T są umieszczane na obwodzie kwadratu w następującym porządku: górna krawędź (od lewej do prawej), prawa krawędź (od góry do dołu), dolna krawędź (od prawej do lewej), lewa krawędź (od dołu do góry)).
T = "AlaMaKota!", $n = 13$, $m = 10$ (długość tekstu)

```
[A, l, a, M, a, K, o, t, a, !, A, l, a]
[t, A, l, a, M, a, K, o, t, a, !, A, M]
[o, !, A, l, a, M, a, K, o, t, a, l, a]
[K, a, l, A, l, a, M, a, K, o, !, a, K]
[a, t, A, M, A, l, a, M, a, t, A, M, o]
[M, o, !, a, K, A, l, a, K, a, l, a, t]
[a, K, a, l, a, t, l, M, o, !, a, K, a]
[l, a, t, A, M, o, K, a, t, A, M, o, !]
[A, M, o, !, a, l, A, !, a, l, a, t, A]
[!, a, K, a, t, o, K, a, M, a, K, a, l]
[a, l, a, M, a, l, A, !, a, t, o, !, a]
[t, A, !, a, t, o, K, a, M, a, l, A, M]
[o, K, a, M, a, l, A, !, a, t, o, K, a]
```

Rys. 2.1. Magiczny kwadrat dla wywołania MKwadrat(0, 13, 10, A, T);

```
void MKwadrat(int k, int n, int m, char **A, char *T) {
 if(n <= 0) return; int x = 0;
 for(int j = k; j < n + k; j++) A[k][j] = T[(x++) % m];
 for(int i = k + 1; i < n + k – 1; i++) A[i][n–1+k] = T[(x++) % m];
 for(int j = n + k – 1; j > k – 1; j––) A[n–1+k][j] = T[(x++) % m];
 for(int i = n + k – 2; i > k; i––) A[i][k] = T[(x++) % m];
 MKwadrat(k + 1, n – 2, m, A, T);
}
```

2. Rodzaje rekurencji (por. [DA], [DS])

Funkcja rekurencyjna R( ) (użyta jako składowa programu komputerowego) może się odwoływać do funkcji R( ) na różne sposoby: bezpośrednio, poprzez inne funkcje, za pomocą jednego bądź wielu wywołań rekurencyjnych umieszczonych w różnych miejscach ciała funkcji. Stąd rozróżniamy kilka rodzajów rekurencji.

2.1. Rekurencja końcowa i niekońcowa

Definicja 2 (rekurencja końcowa (ogonkowa))
Funkcja RK( ) jest zdefiniowana z zastosowaniem *rekurencji końcowej*, gdy zawiera dokładnie jedno wywołanie rekurencyjne RK( ) oraz wywołanie to stanowi ostatnią instrukcję w ciele tej funkcji.

```
void odkonca(string s, int n) {
 if(n >= 0) {
  cout << s[n] << " ";
  odkonca(s, n – 1);
 }
}
```

rezultat dla wywołania
odkonca(”ABCDEFGHIJKLMNOPQ”, 16);:
Q P O N M L K J I H G F E D C B A

Definicja 3 (rekurencja niekońcowa nierozgałęziona)
Funkcja RNK( ) jest zdefiniowana z zastosowaniem *rekurencji niekońcowej nierozgałęzionej*, gdy zawiera dokładnie jedno wywołanie rekurencyjne RNK( ) oraz wywołanie to nie jest ostatnim rozkazem w ciele tej funkcji.

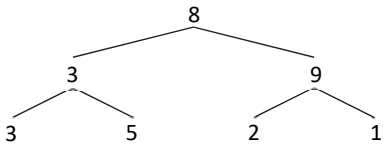
```
void odpoczatku(string s, int n) {
 if(n >= 0) {
  odpoczatku(s, n – 1);
  cout << s[n] << " ";
 }
}
```

rezultat dla wywołania
odpoczatku(”ABCDEFGHIJKLMNOPQ”, 16);:
A B C D E F G H I J K L M N O P Q

W przypadku *rekurencji niekońcowej rozgałęzionej* w ciele funkcji rekurencyjnej RNKR( ) występuje więcej niż jedno wywołanie rekurencyjne oraz po niektórych (lub po wszystkich) takich wywołaniach zapisane są jeszcze inne instrukcje do wykonania.

Przykład 5. Metoda klasy TREE wypisująca elementy drzewa binarnego w porządku LPK (tj. najpierw lewe poddrzewo, potem prawe, a na końcu korzeń) zdefiniowana z użyciem rozgałęzionej rekurencji niekońcowej.

```
void TREE::LPK(Node *p) {
 if(p) {
  LPK(p–>left);
  LPK(p–>right);
  cout << p–>key << ", ";
 }
}
//---------
TREE *b = new TREE( );
// utworzenie drzewa
b–>LPK(b–>root);
```



Rys. 2.2. Drzewo binarne

Dla przedstawionego na rysunku drzewa binarnego oraz wywołania b–>LPK(b–>root); uzyskamy rezultat: 3, 5, 3, 2, 1, 9, 8.

2.2. Rekurencja bezpośrednia i pośrednia

Definicja 4 (rekurencja bezpośrednia)
Funkcja RB( ) jest zdefiniowana z użyciem *rekurencji bezpośredniej*, gdy wywołanie rekurencyjne RB( ) znajduje się bezpośrednio w ciele tej funkcji.

Definicja 5 (rekurencja pośrednia)
Funkcja RP( ) jest zdefiniowana z użyciem *rekurencji pośredniej*, gdy wywołuje funkcję RP( ) w sposób pośredni, poprzez łańcuch innych wywołań.

Przykład 6. (Rekurencja pośrednia)
(a) RP( )→R( )→RP( )→R( ) – funkcja RP( ) wywołuje funkcję R( ), a funkcja R( ) wywołuje funkcję RP( )
(b) RP( )→R1( )→R2( )→R3( )→R4( )→...→RP( )

Przykład 7. Wyznaczanie wartości funkcji trygonometrycznych można zrealizować przy użyciu rekurencji pośredniej (por. [DS]).

```
double Math_::sin_(double x) {
 if(fabs(x) < 0.01)
   return (x – pow(x, 3) / 6);
 double x1 = x / 3;
 double tg_2 = pow(tg_(x1), 2);
 return sin_(x1) * (3 – tg_2) / (1 + tg_2);
}

double Math_::tg_(double x) {
 return sin_(x) / cos_(x);
}

double Math_::cos_(double x) {
 return 1 – 2 * pow(sin_(x / 2), 2);
}
```

$$\sin(x) = \begin{cases} x - \frac{x^3}{6} & \text{dla } |x| < 0.01 \\ \sin\left(\frac{x}{3}\right) \cdot \frac{\left(3 - \mathrm{tg}^2\left(\frac{x}{3}\right)\right)}{\left(1 + \mathrm{tg}^2\left(\frac{x}{3}\right)\right)} & \text{dla } |x| \geq 0.01 \end{cases}$$

$$\mathrm{tg}(x) = \frac{\sin(x)}{\cos(x)}$$

$$\cos(x) = 1 - 2\sin^2\left(\frac{x}{2}\right)$$

rezultaty dla wywołań:
sin_(1.56888); : 0.999998
sin_(0); : 0.0

2.3. Rekurencja zagnieżdżona

Definicja 6 (rekurencja zagnieżdżona)
Funkcja RZ( ) jest zdefiniowana z użyciem *rekurencji zagnieżdżonej*, gdy odwołanie rekurencyjne RZ( ) jest zapisane w ciele tej funkcji jako jeden z parametrów.

Przykład 8. Przykładem takiego zagnieżdżenia może być definicja funkcji Steinhausa-Mosera. Funkcja Steinhausa-Mosera jest określona dla wszystkich trójek ze zbioru:

$$\{(n,m,p)\in N^3: n\geq 1 \wedge m\geq 1 \wedge p\geq 3\}.$$

Dla wszystkich $m\geq 1$ i $p\geq 3$ mamy $M(1,m,p)=1$.
Dla pozostałych $n>1, m\geq 1$ i $p\geq 3$ mamy:

$$M(n,1,3)=n^n$$

$$M(n,1,p+1)=M(n,n,p)$$

$$M(n,m+1,p)=M(M(n,1,p),m,p)$$

3. Nieuzasadnione użycie rekurencji (por. [DA], [DS])

Zaletami stosowania rekurencji są prostota, intuicyjność i przejrzystość definiowanych funkcji. W wielu przypadkach stosowanie rozwiązań rekurencyjnych jest jednak wysoce nieuzasadnione. Rekurencja zmusza do przechowywania na stosie programu dużej ilości dodatkowych elementów, których pamiętanie w przypadku nierekurencyjnego rozwiązania nie byłoby konieczne. Jeśli rekurencja jest zbyt głęboka, może dojść do przepełnienia stosu (i zawieszenia programu). Istnieje wiele prostych problemów (np. generowanie wartości symbolu Newtona), dla których rozwiązanie rekurencyjne prowadzi do wielokrotnego wykonywania tych samych rachunków (w takiej sytuacji stosowniej jest skorzystać z *techniki programowania dynamicznego* i zapamiętywać wyniki obliczeń częściowych w pewnej strukturze). Nieefektywność takiego rekurencyjnego algorytmu skutkuje tym, że czas jego działania dla większych danych staje się nieakceptowalny.

Przykład 9. Symbol Newtona – nieuzasadnione użycie rekurencji.
Aby wyznaczyć $\binom{6}{2}$, algorytm dwukrotnie oblicza wartość $\binom{4}{1}$ (patrz rys. 2.3).

$$\binom{n}{k}=\begin{cases}1 & \text{dla } k=0\vee k=n\\ \binom{n-1}{k-1}+\binom{n-1}{k} & \text{dla pozostałych } n,k\in N\end{cases}$$

```
long int Newt(int n, int k) {
 if(k == 0 || k == n) return 1;
 return Newt(n – 1, k – 1) + Newt(n – 1, k);
}
```

```
Newt(6,2)
   ->Newt(5,1)
      ->Newt(4,0)=1
      ->Newt(4,1)
         ->Newt(3,0)=1
         ->Newt(3,1)
            ->Newt(2,0)=1
            ->Newt(2,1)
               ->Newt(1,0)=1
               ->Newt(1,1)=1
   ->Newt(5,2)
      ->Newt(4,1)
         ->Newt(3,0)=1
         ->Newt(3,1)
            ->Newt(2,0)=1
            ->Newt(2,1)
               ->Newt(1,0)=1
               ->Newt(1,1)=1
      ->Newt(4,2)
         ->Newt(3,1)
            ->Newt(2,0)=1
            ->Newt(2,1)
               ->Newt(1,0)=1
               ->Newt(1,1)=1
         ->Newt(3,2)
            ->Newt(2,1)
               ->Newt(1,0)=1
               ->Newt(1,1)=1
            ->Newt(2,2)=1
```

Rys. 2.3. Drzewo wywołań dla Newt(6, 2)

Przy każdorazowym wywołaniu funkcji, na stosie programu (tworzonym na czas wykonania programu) przydzielany jest dynamicznie obszar pamięci w celu przechowania informacji niezbędnych do prawidłowego wykonania funkcji oraz adresu powrotu, gdzie należy oddać sterowanie po zakończeniu jej działania. Obszar ten, tzw. *rekord wywołania* (*aktywacji*, ang. *activation record*), jest zwalniany po wykonaniu funkcji, stąd jego czas istnienia jest zwykle bardzo krótki. Wyjątek stanowią funkcje wywoływane rekurencyjnie, ponieważ tworzone są rekordy dla nowych wywołań (instancji) funkcji, podczas gdy rekordy dla wcześniejszych wywołań (instancji) nadal na stosie istnieją.

Rekord wywołania umieszczany na stosie programu zwykle (zależy to m.in. od języka programowania) zawiera następujące elementy:

- wartości parametrów,
- zmienne lokalne (mogą być trzymane w innym miejscu, ale wówczas rekord aktywacji zawiera ich oznaczenia i wskazania do miejsc, gdzie się znajdują),
- adres powrotu (informacja, do której instrukcji oddać sterowanie po zakończeniu działania funkcji),

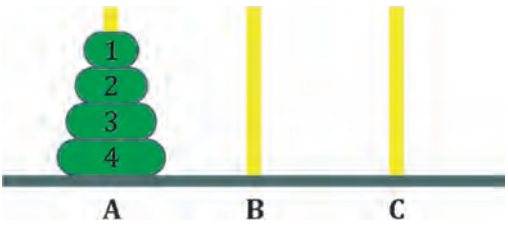
- dynamiczne dowiązanie – wskaźnik na poprzedni rekord aktywacji (ciąg rekordów połączonych wskaźnikami stanowi łańcuch dynamiczny, tzn. aktualną zawartość stosu),
- zwracaną wartość (jeśli funkcja nie jest zadeklarowana jako void).

Drzewo wywołań dla Newt(6, 2) z przykładu 9 pokazuje, ile rekordów aktywacji zostanie utworzonych na stosie programu w celu obliczenia wartości $\binom{6}{2}$, tzn. każde wywołanie funkcji Newt( ) to kolejny rekord aktywacji.

4. Metody rozwiązywania rekurencji (por. [CLR], [RW])

Wyznaczenie elementu ciągu (obiektu) zdefiniowanego rekurencyjnie (zarówno za pomocą wzoru matematycznego, jak i funkcji programu) wiąże się z dużą złożonością czasową i pamięciową. Aby obliczyć wartość elementu a_n tak zdefiniowanego ciągu, jesteśmy zmuszeni wyliczyć wszystkie lub niektóre elementy go poprzedzające, tj. a_{n-1}, a_{n-2}, …, a_0. Zasadne jest więc uzyskanie wzoru jawnego na n-ty wyraz ciągu zadanego rekurencyjnie. Metody otrzymywania wzorów jawnych okażą się przydatne do szacowania złożoności algorytmów rekurencyjnych. Różne postacie zależności rekurencyjnych wymagają innych metod pozyskiwania wzorów jawnych. Dla najprostszych wzór jawny udaje się odgadnąć i udowodnić indukcyjnie.

Przykład 10a. (Problem wież z Hanoi)
Dana jest podstawka z trzema pionowymi palikami. Na pierwszy z nich nałożono n krążków o różnych promieniach (im wyżej, tym mniejszy krążek). Należy przenieść wszystkie krążki z palika (A) na palik (C) przy pomocy palika (B), postępując zgodnie z regułami:
1. Za każdym razem wolno przenieść tylko jeden krążek z jednego palika na inny.
2. Nie można kłaść większego krążka na mniejszy.



Rys. 2.4. Wieża z Hanoi

Każdy ruch należy opisać dwoma cyframi (A_i–>B_i), co odpowiada przeniesieniu najwyższego krążka z palika A_i na palik B_i. Niech a_n oznacza minimalną liczbę ruchów niezbędną do wykonania zadania dla ustalonego n. Należy wyznaczyć wzór jawny na a_n.

Rozwiązanie:
Wyznaczmy wzór rekurencyjny:
$a_0 = 0$, $a_1 = 1$, $a_2 = 3$. Jak przenieść całą wieżę dla dowolnego $n \in N$?

- Przenosimy $n - 1$ krążków z palika (A) na palik pomocniczy (B) przy pomocy palika docelowego (C). Wymaga to a_{n-1} ruchów.
- Następnie przenosimy 1 największy krążek z palika (A) na palik docelowy (C). Wykonujemy jeden ruch.
- Ostatecznie przenosimy $n - 1$ krążków z palika pomocniczego (B) na docelowy (C) przy pomocy palika (A). Wymaga to a_{n-1} ruchów.

Zatem ilość ruchów niezbędnych do przeniesienia n krążków można wyznaczyć ze wzoru:

$$\begin{cases} a_0 = 0 \\ a_n = 2a_{n-1} + 1 \text{ dla } n \in N_+ \end{cases}$$

```
(1) void Hanoi(int n, char A, char B, char C) {
(2)  if(n > 0) {
(3)    Hanoi(n – 1, A, C, B);
(4)    cout << A << "–>" << C << ", ";
(5)    Hanoi(n – 1, B, A, C);
(6)  }
(7) }
```

(3) Przełożenie $n - 1$ krążków z palika (A) na palik (B) przy pomocy palika (C).
(4) Wyświetla ruch przełożenia jednego krążka z palika (A) na (C).
(5) Przełożenie $n - 1$ krążków z palika (B) na palik (C) przy pomocy palika (A).
Rezultat dla wywołania Hanoi(4,'1','2','3');:

```
1->2, 1->3, 2->3, 1->2, 3->1, 3->2, 1->2, 1->3, 2->3, 2->1, 3->1, 2->3, 1->2,
1->3, 2->3,
```

Analizując algorytm przekładania krążków i uruchamiając program dla $n = 2, 3, 4, 5$ (otrzymujemy odpowiednio 3, 7, 15 i 31 przełożeń), łatwo odgadujemy postać jawną wzoru na n-ty wyraz ciągu (a_n): $a_n = 2^n - 1$ dla dowolnego $n \in N$. Wzór należy udowodnić indukcyjnie.

Zwykle odgadnięcie wzoru jawnego nie jest możliwe.

4.1. Metoda iteracyjna (por. [CLR], [RW])

Metoda polega na rozwijaniu (iterowaniu) rekurencji przy wykorzystaniu zależności rekurencyjnej do momentu, gdy jesteśmy w stanie odgadnąć (na podstawie warunku brzegowego) jawną postać wzoru.

Przykład 10b. Wyznacz wzór jawny na n-ty wyraz ciągu (a_n) zadanego następującą zależnością rekurencyjną:

$$\begin{cases} a_0 = 0 \\ a_n = 2a_{n-1} + 1 \text{ dla } n \in N_+ \end{cases}$$

$$\underline{a_n} = 2a_{n-1} + 1 = 2(2a_{n-2} + 1) + 1 = 2^2 a_{n-2} + 2 + 1 =$$

$$= 2^2(2a_{n-3} + 1) + 2 + 1 = 2^3 a_{n-3} + 2^2 + 2 + 1 =$$

$$= 2^3(2a_{n-4} + 1) + 2^2 + 2 + 1 = 2^4 a_{n-4} + 2^3 + 2^2 + 2 + 1 = \cdots =$$

$$= 2^n \underbrace{a_{n-n}}_{a_0=0} + 2^{n-1} + \cdots + 2^3 + 2^2 + 2 + 1 =$$

$$= 2^{n-1} + \cdots + 2^3 + 2^2 + 2 + 1 = \frac{1-2^n}{1-2} = \underline{2^n - 1}.$$

4.2. Metoda z wykorzystaniem równania charakterystycznego (por. [RW])

Metodę tę stosujemy do zależności rekurencyjnych postaci:

(1) $\begin{cases} a_0, \dots, a_{k-1} & \text{warunki brzegowe} \\ a_n = \mathrm{c}_{k-1} a_{n-1} + \cdots + \mathrm{c}_1 a_{n-k+1} + \mathrm{c}_0 a_{n-k} \end{cases}$

Twierdzenie 1 (por. [RW])
Rozważmy zależność rekurencyjną postaci:

(2) $\begin{cases} a_0, a_1 \\ a_n = \mathrm{c}_1 a_{n-1} + \mathrm{c}_0 a_{n-2} \end{cases}$
mającą równanie charakterystyczne:

(3) $x^2 - \mathrm{c}_1 x - \mathrm{c}_0 = 0$, gdzie c_0, c_1 są niezerowymi stałymi.

(A) Jeśli równanie charakterystyczne ma dwa rożne pierwiastki x_1 i x_2, to wzór jawny ma postać:

(4) $a_n = \mathrm{A}_0 (x_1)^n + \mathrm{B}_0 (x_2)^n$ dla pewnych stałych A_0 i B_0.
Ponieważ a_0 i a_1 są dane, więc wartości A_0 i B_0 wyznaczamy, podstawiając do wzoru (4) $n = 0$ i $n = 1$ i rozwiązując układ dwóch równań z dwiema niewiadomymi A_0 i B_0.

(B) Jeśli równanie charakterystyczne ma jedno rozwiązanie x ($\Delta = 0$), wówczas wzór jawny ma postać:

(5) $a_n = \mathrm{A}_0 \cdot x^n + \mathrm{A}_1 \cdot n \cdot x^n$ dla pewnych stałych A_0 i A_1, które wyznaczamy analogicznie jak w punkcie (4).

Przykład 10c. Wyznacz wzór jawny na n-ty wyraz ciągu (a_n) przedstawionego w przykładzie 10a. Zastosuj metodę z równaniem charakterystycznym.

$$\begin{cases} a_0 = 0 \\ a_n = 2a_{n-1} + 1 \text{ dla } n \in N_+ \end{cases}$$

Zapiszmy zależność rekurencyjną w wymaganej postaci:

$a_n = 2a_{n-1} + 1$, więc $a_{n+1} = 2a_n + 1$

$a_{n+1} - a_n = 2a_n + 1 - 2a_{n-1} - 1$, więc $a_{n+1} = 3a_n - 2a_{n-1}$, ostatecznie:

$$\begin{cases} a_0 = 0, a_1 = 1 \\ a_{n+1} = 3a_n - 2a_{n-1} \text{ dla } n > 1 \end{cases}$$

Z równania charakterystycznego $x^2 - 3x + 2 = 0$ wyznaczamy dwa pierwiastki: $x_1 = 1$ i $x_2 = 2$. Stąd wzór jawny na n-ty wyraz ciągu (a_n) ma postać

$a_n = \mathrm{A}_0(1)^n + \mathrm{B}_0(2)^n$.

Dla $n = 0$: $a_0 = 0 = \mathrm{A}_0 + \mathrm{B}_0$, $\Rightarrow \mathrm{A}_0 = -\mathrm{B}_0$,

Dla $n = 1$: $a_1 = 1 = \mathrm{A}_0 + 2\mathrm{B}_0 = \mathrm{B}_0$ $\Rightarrow \mathrm{B}_0 = 1, \mathrm{A}_0 = -1$

Ostatecznie $\underline{a_n = -1 + 2^n}$ dla dowolnej liczby naturalnej n.

4.3. Metoda z wykorzystaniem funkcji tworzącej (por. [RW])

Definicja 7 (funkcja tworząca ciągu)
Funkcją tworzącą ciągu $(a_n) = (a_0, a_1, a_2, \dots)$ liczb rzeczywistych nazywamy funkcję $A(x)$ określoną wzorem:

$A(x) = \sum_{n=0}^{\infty} a_n x^n$.

Jest to wygodne przedstawienie ciągu (a_n) jako szeregu potęgowego zmiennej x. Funkcje tworzące służą m.in. do rozwiązywania rekurencji.

Przykład 10d. Wyznacz wzór jawny na n-ty wyraz ciągu (a_n) przedstawionego w przykładzie 10a. Skorzystaj z metody z funkcją tworzącą.

$$\begin{cases} a_0 = 0 \\ a_n = 2a_{n-1} + 1 \text{ dla } n \in N_+ \end{cases}$$

a_n oznacza minimalną liczbę ruchów koniecznych do przeniesienia n krążków z jednego palika na drugi.

$\underline{f(x)} = \sum_{n=0}^{\infty} a_n x^n = 0 + \sum_{n=1}^{\infty} a_n x^n = \sum_{n=0}^{\infty} a_{n+1} x^{n+1} =$

$= \sum_{n=0}^{\infty} (2a_n + 1) x^{n+1} = 2x \sum_{n=0}^{\infty} a_n x^n + x \sum_{n=0}^{\infty} x^n =^{(a)} \underline{2xf(x) + \frac{x}{1-x}}$

$f(x)(1 - 2x) = \frac{x}{1-x}$

$f(x) = \frac{x}{(1-x)(1-2x)} =^{(b)} \frac{\mathrm{A}}{(1-x)} + \frac{\mathrm{B}}{(1-2x)} =^{(c)} \frac{-1}{(1-x)} + \frac{1}{(1-2x)} =^{(a)}$

$= -\sum_{n=0}^{\infty} x^n + \sum_{n=0}^{\infty} 2^n x^n$

Stąd wzór jawny na n-ty wyraz ciągu ma postać $\underline{a_n = -1 + 2^n}$ dla dowolnego naturalnego n.

(a) $\sum_{n=0}^{\infty}(ax)^n = 1 + ax + (ax)^2 + (ax)^3 + \cdots = \frac{1}{1-ax}$
(b) Rozkładamy ułamek na sumę ułamków prostych.
(c) $x = \mathrm{A}(1-2x) + \mathrm{B}(1-x) = \mathrm{A} + \mathrm{B} - (2\mathrm{A} + \mathrm{B})x$
$0 = \mathrm{A} + \mathrm{B} \quad \Rightarrow \mathrm{A} = -\mathrm{B} = -1$
$-1 = 2\mathrm{A} + \mathrm{B} \quad \Rightarrow \mathrm{B} = 1$

5. Sortowanie przez kopcowanie (HeapSort)

Jako przykład uzasadnionego zastosowania rekurencji przedstawiony zostanie algorytm sortowania przez kopcowanie HeapSort( ). Złożoność czasową tego algorytmu szacujemy przez $O(n\lg n)$. Efektywność sortowania HeapSort( ) wynika z wykorzystania struktury kopca, który spełnia tu zadanie kolejki priorytetowej. Własność kopca zupełnego pozwala reprezentować drzewo (kopiec) za pomocą spójnej tablicy.

HeapSort( ) jest *sortowaniem w miejscu* (tj. stała liczba elementów tablicy jest przechowywana poza tablicą wejściową w trakcie działania algorytmu).

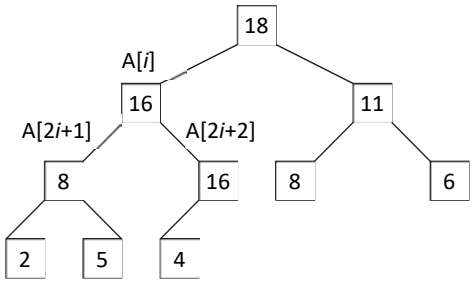
5.1. Definicja kopca (por. [BDR], [CLR], [DA], [DS])

Definicje i fakty (por. [CLR], [LW], [WR]):

- *graf niezorientowany* (graf) G – dowolna para $G = (V, E)$ (V – zbiór wierzchołków (węzłów), E – zbiór krawędzi), gdzie $E \subseteq \{\{u, v\}: u, v \in V \wedge u \neq v\}$ ($\{u, v\}$ oznacza nieuporządkowaną parę wierzchołków $u, v \in V$);
- *ścieżka* w grafie G – ciąg wierzchołków $(v_0, v_1, \ldots, v_k)$ $(k \geq 0)$ taki, że $\{v_i, v_{i+1}\} \in E$ jest krawędzią (dla $i = 0, 1, \ldots, k-1$) oraz wszystkie krawędzie są różne;
- ścieżka $(v_0, v_1, \ldots, v_k)$ tworzy *cykl*, jeśli $v_0 = v_k$ oraz zawiera co najmniej jedną krawędź;
- *spójny graf niezorientowany* – graf niezorientowany, w którym między wszystkimi parami wierzchołków istnieje ścieżka;
- *graf acykliczny* – graf bez cykli;
- *drzewo z korzeniem* – niezorientowany, acykliczny graf spójny z jednym wyróżnionym węzłem zwanym *korzeniem drzewa*;
- *drzewo binarne* – drzewo z korzeniem, które:
  - nie zawiera żadnych węzłów (*drzewo puste*) lub
  - zawiera trzy rozłączne zbiory węzłów: *korzeń*, drzewo binarne zwane *lewym poddrzewem* i drzewo binarne zwane *prawym poddrzewem.*

Definicja 8 (kopiec binarny) (por. [DS])
Kopcem binarnym nazywamy drzewo binarne o dwóch następujących własnościach:
(a) wartość w dowolnym jego węźle jest nie mniejsza niż wartości przechowywane w każdym z jego synów (tzw. *własność kopca*),
(b) wszystkie liście w drzewie leżą na co najwyżej dwóch ostatnich poziomach, a liście na ostatnim poziomie szczelnie wypełniają jego lewą część (tzw. *kopiec zupełny*).

- Wysokość kopca (tj. długość ścieżki od korzenia do liścia) wynosi $h = \lfloor \log_2 n \rfloor$, gdzie n jest liczbą wierzchołków kopca.
- Własność (b) pozwala reprezentować kopiec (w celu zaoszczędzenia pamięci) za pomocą spójnej tablicy, przyporządkowując każdemu węzłowi drzewa odpowiednie pole tablicy.



(a) kopiec na drzewie

18	16	11	8	16	8	6	2	5	4
0	1	2	3	4	5	6	7	8	9

(b) kopiec w tablicy

Rys. 2.5. Kopiec można przedstawić w postaci (a) drzewa binarnego, (b) kopca w tablicy

Definicja 9 (*kopiec binarny* – tablicowa reprezentacja kopca)
Tablica A długości n reprezentuje *kopiec binarny*, gdy jest w niej spełniona tzw. *własność kopca*:

$$\mathrm{A}[i] \geq \mathrm{A}[2i+1] \text{ i } \mathrm{A}[i] \geq \mathrm{A}[2i+2] \text{ dla } 0 \leq i \leq E((n-2)/2).$$

- Elementy w kopcu (tablicy) nie są całkowicie uporządkowane.
- W korzeniu umieszczony jest największy element kopca.
- Dla każdego wierzchołka wartości wszystkich jego potomków są mniejsze bądź równe od jego wartości.

5.2. Sortowanie HeapSort

Jako przykład uzasadnionej rekurencji przedstawiony zostanie algorytm sortowania przez kopcowanie.
Idea algorytmu (sortowanie przez kopcowanie HeapSort( )) (por. [BDR], [DS])
(1) Przekształcamy drzewo (tablicę) do postaci kopca, zaczynając jego budowę od ostatniego węzła wewnętrznego (zakładamy, że liście są 1-elementowymi kopcami), dodając kolejno po jednym węźle do rosnących kopców.
(2) Sortowanie polega na (n–1)-krotnym powtórzeniu dwóch kroków: zamianie korzenia (pierwszego elementu w tablicy) z ostatnim elementem aktualnego kopca (ostatnim elementem w nieposortowanej części tablicy) i przywróceniu drzewu (tablicy) własności kopca.

Schemat (sortowanie HeapSort( ))
(1) Build( ) – budowa kopca
(2) $(n-1)\cdot$ DelMax( ) – zamiana korzenia z ostatnim elementem kopca oraz przywrócenie własności kopca

(1) Build( ) – budowa kopca

Funkcje pomocnicze wykorzystane w Build( ):
- Max( ) – funkcja zwracająca indeks największego elementu spośród elementów: A[i], A[$2i+1$], A[$2i+2$]

```
int Sort::Max(int i, int j) {
 if(j < 2 * i + 1) return i;
 if(j == 2 * i + 1) return (A[i] >= A[2*i+1]) ? i : (2 * i + 1);
 if(A[i] >= A[2*i+1]) return (A[i] >= A[2*i+2]) ? i : (2 * i + 2);
 return (A[2*i+1] >= A[2*i+2]) ? (2 * i + 1) : (2 * i + 2);
}
```

- Heapify( ) – *rekurencyjny* algorytm przywracania własności kopca na ścieżce

```
void Sort::Heapify(int i, int j) {
 int k = Max(i, j), pom;          // znajdź indeks elementu max{ A[i], A[2i + 1],
                                  // A[2i + 2] }
 if(k != i) {
  pom = A[i]; A[i] = A[k]; A[k] = pom;
  Heapify(k, j);                  // przywróć własność kopca w drzewie A[k]
 }
}
void Sort::Build( ) {
 for(int i = ((n – 2) / 2); i >= 0; i––) Heapify(i, n – 1);
}
```

Przykład 11a. Budowa kopca w tablicy – rezultat działania metody Build( );

```
    0 1 2 3 4 5 6 7 8 9
    ------------------
A:  5 4 1 7 5 3 2 8 6 9
    5 4 1 7 9 3 2 8 6 5
    5 4 1 8 9 3 2 7 6 5
    5 4 3 8 9 1 2 7 6 5
    5 9 3 8 4 1 2 7 6 5
    5 9 3 8 5 1 2 7 6 4
    9 5 3 8 5 1 2 7 6 4
    9 8 3 5 5 1 2 7 6 4
    9 8 3 7 5 1 2 5 6 4 - kopiec
```

(2) DelMax( ) – zamiana korzenia z ostatnim elementem kopca oraz przywrócenie własności kopca

```
void Sort::DelMax(int i) {
 int pom = A[0]; A[0] = A[i]; A[i] = pom;   // zamień korzeń z ostatnim
                                            // elementem nieposortowanej
                                            // części tablicy,
 Heapify(0, i – 1);                         // przywróć własność kopca
                                            // w części A[0]..A[i − 1]
}
```

Przykład 11b. Rezultat działania metody DelMax( );

```
    0 1 2 3 4 5 6 7 8 9
    ------------------
A:  9 8 3 7 5 1 2 5 6 4 - kopiec
    4 8 3 7 5 1 2 5 6|9
    8 4 3 7 5 1 2 5 6|9
    8 7 3 4 5 1 2 5 6|9
    8 7 3 6 5 1 2 5 4|9 - kopiec
```

(3) HeapSort( ) – sortowanie przez kopcowanie

```
void Sort::HeapSort( ) {
 Build( );
 for(int i = n – 1; i >= 1; i––) DelMax(i);
}
```

Przykład 11c. Ciąg dalszy sortowania:

```
      0 1 2 3 4 5 6 7 8 9
      -------------------
A:    8 7 3 6 5 1 2 5 4|9 - kopiec
i=8: 4 7 3 6 5 1 2 5|8 9
      7 4 3 6 5 1 2 5|8 9
      7 6 3 4 5 1 2 5|8 9
      7 6 3 5 5 1 2 4|8 9 - kopiec
i=7: 4 6 3 5 5 1 2|7 8 9
      6 4 3 5 5 1 2|7 8 9
      6 5 3 4 5 1 2|7 8 9 - kopiec
i=6: 2 5 3 4 5 1|6 7 8 9
      5 2 3 4 5 1|6 7 8 9
      5 5 3 4 2 1|6 7 8 9 - kopiec
i=5: 1 5 3 4 2|5 6 7 8 9
      5 1 3 4 2|5 6 7 8 9
      5 4 3 1 2|5 6 7 8 9 - kopiec
i=4: 2 4 3 1|5 5 6 7 8 9
      4 2 3 1|5 5 6 7 8 9 - kopiec
i=3: 1 2 3|4 5 5 6 7 8 9
      3 2 1|4 5 5 6 7 8 9 - kopiec
i=2: 1 2|3 4 5 5 6 7 8 9
      2 1|3 4 5 5 6 7 8 9 - kopiec
i=1: 1|2 3 4 5 5 6 7 8 9 - kopiec
```

5.3. Analiza złożoności czasowej algorytmu HeapSort( )

[Złożoność pesymistyczna] (algorytm HeapSort( )) (por. [BDR])

- n oznacza rozmiar tablicy do posortowania.
- Przyjmijmy za operację elementarną porównanie elementów tablicy.
- Złożoność pesymistyczną algorytmu Build( ) szacujemy przez $O(n)$.

Uzasadnienie:

- Funkcja pomocnicza Max( ) wykonuje maksymalnie dwa porównania.
- Funkcja pomocnicza Heapify(i, j) wywołana dla węzła i z poziomu f $(0 \le f < h)$ wykonuje co najwyżej $2(h-f)$ porównań. Na poziomie f jest 2^f węzłów. Dla węzłów na poziomie h Heapify(i, j) nie wykonuje żadnych operacji elementarnych.

$$T_{\max}(n) = \sum_{f=0}^{h-1} 2^f 2(h-f) =^{(a)} \sum_{f=1}^{h} 2^f (h-f+1) =$$
$$= 2^{h+1} \sum_{f=1}^{h} \frac{h+1-f}{2^{(h+1-f)}} =^{(b)} 2^{h+1} \sum_{f=1}^{h} \frac{f}{2^f} =^{(a)}$$
$$= 2^{h+1} \sum_{f=1}^{h} \left(\sum_{j=f}^{h} \frac{1}{2^j} \right) \le^{(a)} 2^{h+1} \sum_{f=1}^{h} \left(\frac{1}{2^f} \sum_{j=0}^{\infty} \frac{1}{2^j} \right) =^{(c)} 2^{h+1} \sum_{f=1}^{h} \frac{2}{2^f} =$$
$$= 2^{h+1} \sum_{f=0}^{h-1} \frac{1}{2^f} \le 2^{h+1} \sum_{f=0}^{\infty} \frac{1}{2^f} =^{(c)} 2^{h+2} = 4 \cdot 2^h =^{(d)} 4n = O(n)$$

(a) Wystarczy rozpisać.

(b) $\forall n \in N_+ \sum_{i=1}^{n} \frac{n+1-i}{2^{(n+1-i)}} = \sum_{i=1}^{n} \frac{i}{2^i}$ (poprawność wzoru należy udowodnić indukcyjnie)

(c) $\sum_{i=0}^{\infty} \frac{1}{2^i} = \frac{1}{1-\frac{1}{2}} = 2$ (poprawność wzoru należy udowodnić)

(d) $h = \lfloor \log_2 n \rfloor$ – wysokość kopca

- Sumaryczną złożoność pesymistyczną dla $(n-1)$ wywołań funkcji DelMax( ) szacujemy przez $O(n\lg n)$.

Uzasadnienie:

- Funkcja DelMax( ) wywołana dla i-tego węzła z poziomu f $(0 < f \leq h)$ wykona maksymalnie $2f$ porównań. Na poziomach $f = 0, \ldots, (h-1)$ znajduje się po 2^f węzłów, a na poziomie h mamy $n - (2^h - 1)$ węzłów. Policzmy złożoność pesymistyczną dla wszystkich wierzchołków z poziomów $1, \ldots, (h-1)$:

$$T_{\max}(n) = \sum_{f=1}^{h-1} 2^f 2f = 2\sum_{f=1}^{h-1}\left(\sum_{j=1}^{f} 2^f\right) =^{(a)} 2\sum_{f=1}^{h-1}\left(\sum_{j=f}^{h-1} 2^j\right) =$$

$$= 2\sum_{f=1}^{h-1} 2^f\left(\sum_{j=0}^{h-1-f} 2^j\right) = 2\sum_{f=1}^{h-1} 2^f\left(\frac{1-2^{h-f}}{1-2}\right) =$$

$$= 2\sum_{f=1}^{h-1} 2^f\left(2^{h-f}-1\right) = 2\sum_{f=1}^{h-1}\left(2^h - 2^f\right) =$$

$$= 2\left(2^h(h-1) - \sum_{f=1}^{h-1} 2^f\right) = 2\left(2^h(h-1) - 2\left(\frac{1-2^{h-1}}{1-2}\right)\right) =$$

$$= 2\left(2^h(h-1) - 2^h + 2\right) = 2n(-1+\lg n) - 2n + 4 = O(n\lg n)$$

(a) Wystarczy rozpisać.

- Sumaryczna pesymistyczna złożoność wszystkich wywołań funkcji DelMax( ) dla wierzchołków z poziomu h (gdy na poziomie h mamy 2^h wierzchołków):

$$T_{\max}(n) = 2^h 2h = 2n\lg n = O(n\lg n)$$

- Stąd podczas $(n-1)$ wywołań funkcji DelMax( ) algorytm wykona $O(n\lg n)$ porównań.
- Ostatecznie złożoność pesymistyczną algorytmu HeapSort( ) szacujemy przez $O(n\lg n)$.

6. Zadania różne

[01] Zaproponuj algorytm (o minimalnej liczbie wywołań rekurencyjnych) wypisujący wszystkie liczby ciągu Fibonacciego mniejsze od ustalonego n.

Rozwiązanie:

```
void Fib(int n, long int k1, long int k2) {
 if(k2 < 2) { if(n < 1) cout << "0"; else cout << "0 1 1 "; }
 if(k1 + k2 < n) {
  cout << (k1 + k2) << " ";
  Fib(n, k2, k1 + k2);
 }
}
```

Rezultat dla wywołania Fib(1000,1,1);: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
Ilość wywołań rekurencyjnych: 15

[02] Zapisz rekurencyjną definicję oraz zaproponuj algorytm realizujący wyznaczanie:
(a) potęgi o podstawie $x \in R \setminus \{0\}$ i wykładniku naturalnym $n \in N$,
(b) wartości funkcji $f(n) = n!$,
(c) wartości funkcji symbolu Newtona SN(n, k).

Rozwiązanie:
(a) Rekurencyjna definicja potęgi o podstawie $x \in R \setminus \{0\}$ i wykładniku naturalnym $n \in N$:

$$x^n = \begin{cases} 1 & \text{dla } n = 0 \\ x \cdot x^{n-1} & \text{dla } n \in N_+ \end{cases}$$

```
double Powx(int n, double x) {
 if(n == 0) return 1;
 return Powx(n – 1, x) * x;
}
```

(b) Rekurencyjna definicja wartości funkcji $f(n) = n!$:

$$n! = \begin{cases} 1 & \text{dla } n = 0 \vee n = 1 \\ n \cdot (n-1)! & \text{dla } n > 1, n \in N \end{cases}$$

```
long int silnia(int n) {
 if(n == 0 || n == 1) return 1;
 return silnia(n – 1) * n;
}
```

(c) Rekurencyjna definicja symbolu Newtona:

$$\binom{n}{k} = \begin{cases} 1 & \text{dla } k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{dla pozostałych } n, k \in N \end{cases}$$

```
long int Newt(int n, int k) {
  if(k == 0 || k == n) return 1;
  return Newt(n – 1, k – 1) + Newt(n – 1, k);
}
```

[03] Uzasadnij następujący fakt. Wysokość kopca (tj. długość ścieżki od korzenia do liścia) wynosi $h = \lfloor \log_2 n \rfloor$, gdzie n jest liczbą wierzchołków kopca.

Rozwiązanie: Załóżmy, że kopiec mieści n elementów. Na każdym i-tym (wewnętrznym) poziomie znajduje się 2^i elementów (poziomy numerujemy od 0). Na ostatnim poziomie znajduje się $k = 1, \ldots, 2^h$ elementów, gdzie h jest długością najdłuższej ścieżki. Zachodzi zatem zależność:

$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + k = n$, stąd

$2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 1 \le n \le 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h$

$\frac{1-2^h}{1-2} + 1 \le n \le \frac{1-2^{h+1}}{1-2}$, więc $2^h \le n \le 2^{h+1} - 1$, stąd $2^h \le n < 2^{h+1}$

i ostatecznie $h = \lfloor \log_2 n \rfloor$.

[04] Czy poniżej wypełniona tablica T reprezentuje kopiec? Odpowiedź uzasadnij.

(a)

16	16	11	11	16	8	6	2	6	15
0	1	2	3	4	5	6	7	8	9

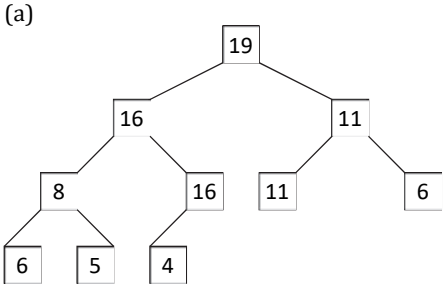
(b)

16	8	15	6	6	11	11	5	5	8
0	1	2	3	4	5	6	7	8	9

Rozwiązanie:
(a) Tablica reprezentuje kopiec. Własność kopca jest zachowana.
(b) To nie jest kopiec. Nie jest zachowana własność kopca (T[4] < T[9], tzn. 6 < 8).

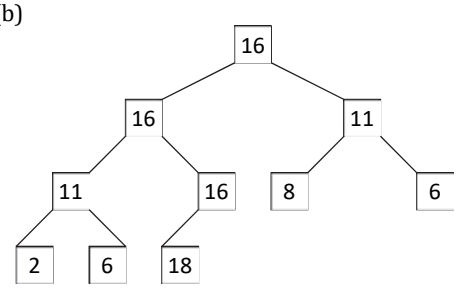
[05] Czy poniższe drzewo binarne reprezentuje kopiec? Odpowiedź uzasadnij.

(a)



Rys. 2.6a. Drzewo binarne do zadania [05] (a)

(b)



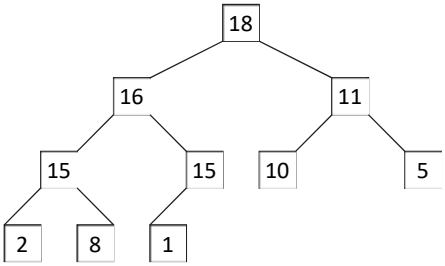
Rys. 2.6b. Drzewo binarne do zadania [05] (b)

Rozwiązanie:
(a) Drzewo jest kopcem. Zachowana jest struktura i własność kopca.
(b) Drzewo nie jest kopcem. Nie jest zachowana własność kopca.

[06] Utwórz kopiec (na drzewie), wkładając do niego kolejno następujące elementy:
(a) 10, 8, 16, 15, 1, 11, 5, 2, 15, 18 (b) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

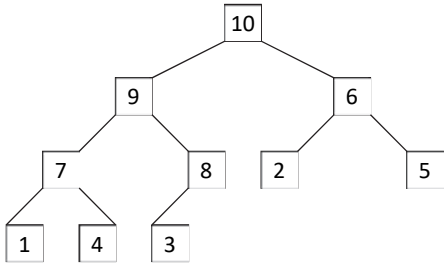
Rozwiązanie:

(a)



Rys. 2.7a. Kopiec do zadania [06] (a)

(b)



Rys. 2.7b. Kopiec do zadania [06] (b)

[07] Przykładem rekurencji zagnieżdżonej może być definicja funkcji Ackermanna (por. [DA]). Podaj implementację funkcji Ackermanna w języku C++.

$$A(m,n) = \begin{cases} n+1 & \text{dla } m = 0 \\ A(m-1,1) & \text{dla } m > 0, n = 0 \\ A\big(m-1, A(m,n-1)\big) & \text{dla } m > 0, n > 0 \end{cases}$$

Rozwiązanie:

```
long Ackermann(long m, long n) {
  if(m == 0) return n + 1;
  if(m > 0 && n == 0) return Ackermann(m – 1, 1);
  return Ackermann(m – 1, Ackermann(m, n – 1));
}
```

[08] Niech $\Sigma = \{a, b, c\}$. Niech t_n oznacza liczbę słów długości n nad alfabetem Σ, w których jest parzysta liczba liter a. (a) Oblicz pięć pierwszych wyrazów ciągu (t_n). (b) Znajdź wzór rekurencyjny i wzór jawny na t_n (n-ty wyraz ciągu).

Rozwiązanie: Niech A_n oznacza zbiór słów w Σ^n (zbiór słów długości n nad alfabetem Σ), w których jest parzysta liczba liter a.

$t_0 = 1, \quad A_0 = \{ \lambda \} \quad | \; t_4 = 41$

$t_1 = 2, \quad A_1 = \{ b, c \} \quad | \; t_5 = 122$

$t_2 = 5, \quad A_2 = \{ bb, cb, bc, cc, aa \}$

$t_3 = 14, \quad A_3 = \{ bbb, cbb, bcb, ccb, aab, bbc, cbc, bcc,$

$ccc, aac, baa, aba, caa, aca \}$

- Jeśli słowo w A_n kończy się literą b, to może być ona poprzedzona dowolnym słowem z A_{n-1}.
- Stąd t_{n-1} słów w A_n kończy się literą b. Podobnie t_{n-1} słów w A_n kończy się literą c.
- Jeśli słowo w A_n kończy się literą a, to musi być ona poprzedzona słowem z Σ^{n-1}, w którym jest nieparzysta liczba liter a.
- Ponieważ Σ^{n-1} ma 3^{n-1} słów, to $3^{n-1} - t_{n-1}$ z nich musi mieć nieparzystą liczbę liter a.
- Stąd $3^{n-1} - t_{n-1}$ słów w A_n kończy się literą a. Zatem

$t_n = 2t_{n-1} + 3^{n-1} - t_{n-1} = 3^{n-1} + t_{n-1}$ dla $n \geq 1$. Stąd

$$\begin{cases} t_0 = 1 \\ t_n = 3^{n-1} + t_{n-1} \text{ dla } n \geq 1 \end{cases}$$

Wyznaczamy wzór jawny:

$$\underline{t_n} = 3^{n-1} + t_{n-1} = 3^{n-1} + 3^{n-2} + t_{n-2} =$$

$$= 3^{n-1} + 3^{n-2} + 3^{n-3} + t_{n-3} = \cdots =$$

$$= 3^{n-1} + 3^{n-2} + 3^{n-3} + \cdots + 3^0 + t_0 =$$

$$= 1 + \sum_{k=0}^{n-1} 3^k = 1 + \frac{1-3^n}{1-3} = \underline{\frac{3^n+1}{2}} \text{ dla } n \geq 0$$

[09] Niech $\Sigma = \{a, b, c\}$ i niech t_n oznacza liczbę słów długości n, które można wygenerować z liter a, b, c tak, aby utworzone słowa nie zawierały ciągu znaków aa (tzn. kolejnych liter a). (a) Oblicz t_0, t_1, t_2. (b) Znajdź wzór rekurencyjny na t_n i oblicz t_3, t_4, t_5, a następnie znajdź wzór jawny na t_n i go udowodnij.

Rozwiązanie: Niech A_n oznacza zbiór słów w Σ^n niezawierających kolejnych liter a.

$t_0 = 1, \quad A_0 = \{\lambda\} \quad | \; t_4 = 60$
$t_1 = 3, \quad A_1 = \{a, b, c\} \quad | \; t_5 = 164$
$t_2 = 8, \quad A_2 = \{ab, bb, cb, ac, bc, cc, ba, ca\} = \Sigma^2 - \{aa\}$
$t_3 = 22, \quad A_3 = \Sigma^3 - \{aaa, aab, aac, baa, caa\}$

- Załóżmy, że $n \geq 2$ i spróbujmy zliczyć słowa w A_n za pomocą słów krótszych.
- Jeśli słowo w A_n kończy się literą b, to litera b może być poprzedzona dowolnym słowem ze zbioru A_{n-1}.
- Zatem t_{n-1} słów w zbiorze A_n kończy się literą b i t_{n-1} słów kończy się literą c.
- Jeśli słowo w A_n kończy się literą a, to dwiema ostatnimi literami muszą być ba lub ca i może je poprzedzać dowolne słowo ze zbioru A_{n-2}. Zatem $2t_{n-2}$ słów ze zbioru A_n kończy się literą a. Stąd

$$\begin{cases} t_0 = 1, t_1 = 3 \\ t_n = 2t_{n-1} + 2t_{n-2} \text{ dla } n \geq 2 \end{cases}$$

Wyznaczamy wzór jawny:
$f(x) = x^2 - 2x - 2 = 0$ – wielomian charakterystyczny

$\Delta = 4 + 8 = 12, \sqrt{\Delta} = 2\sqrt{3}, x_1 = \frac{2-2\sqrt{3}}{2} = 1 - \sqrt{3}, x_2 = \frac{2+2\sqrt{3}}{2} = 1 + \sqrt{3},$

$$t_n = \mathrm{A}_0(1 + \sqrt{3})^n + \mathrm{B}_0(1 - \sqrt{3})^n$$

$n = 0$: $t_0 = 1 = \mathrm{A}_0(1 + \sqrt{3})^0 + \mathrm{B}_0(1 - \sqrt{3})^0$, stąd $\mathrm{A}_0 = 1 - \mathrm{B}_0$

$n = 1$: $t_1 = 3 = \mathrm{A}_0(1 + \sqrt{3})^1 + \mathrm{B}_0(1 - \sqrt{3})^1$

$3 = (1 - \mathrm{B}_0)(1 + \sqrt{3}) + \mathrm{B}_0(1 - \sqrt{3}),$

stąd $2\sqrt{3}\mathrm{B}_0 = \sqrt{3} - 2, \mathrm{B}_0 = \frac{\sqrt{3}-2}{2\sqrt{3}}, \mathrm{A}_0 = \frac{\sqrt{3}+2}{2\sqrt{3}}$

Ostatecznie: $t_n = \frac{\sqrt{3}+2}{2\sqrt{3}}(1 + \sqrt{3})^n + \frac{\sqrt{3}-2}{2\sqrt{3}}(1 - \sqrt{3})^n$ dla $n \geq 0$
Udowodnimy, że oba wzory (rekurencyjny i jawny) definiują ten sam ciąg (t_n).
Dowód indukcyjny (II schemat indukcji)
(1) baza indukcji: $t_0 = 1, t_0 = \frac{\sqrt{3}+2}{2\sqrt{3}} + \frac{\sqrt{3}-2}{2\sqrt{3}} = 1$
(2) krok indukcyjny:

Zakładamy prawdziwość wzoru jawnego dla wszystkich liczb naturalnych mniejszych od pewnej wartości naturalnej k.

Pokażemy, że wówczas wzór jest również poprawny dla k.

$$\mathrm{L}_k = t_k \overset{\text{def.rek}}{=} 2t_{k-1} + 2t_{k-2} \overset{\text{zał.ind}}{=}$$

$$= 2\left[\tfrac{\sqrt{3}+2}{2\sqrt{3}}\left(1+\sqrt{3}\right)^{k-1} + \tfrac{\sqrt{3}-2}{2\sqrt{3}}\left(1-\sqrt{3}\right)^{k-1}\right] +$$

$$+2\left[\tfrac{\sqrt{3}+2}{2\sqrt{3}}(1+\sqrt{3})^{k-2} + \tfrac{\sqrt{3}-2}{2\sqrt{3}}(1-\sqrt{3})^{k-2}\right] =$$

$$= 2\left[\tfrac{\sqrt{3}+2}{2\sqrt{3}}\left(1+\sqrt{3}\right)^{k-2}(1+\sqrt{3}+1) + \tfrac{\sqrt{3}-2}{2\sqrt{3}}\left(1-\sqrt{3}\right)^{k-2}(1-\sqrt{3}+1)\right] =^{(a)}$$

$$= \tfrac{\sqrt{3}+2}{2\sqrt{3}}(1+\sqrt{3})^k + \tfrac{\sqrt{3}-2}{2\sqrt{3}}(1-\sqrt{3})^k = \mathrm{P}_k$$

(a) $\left(1+\sqrt{3}\right)^2 = 1 + 2\sqrt{3} + 3 = 2(2+\sqrt{3})$

$\left(1-\sqrt{3}\right)^2 = 1 - 2\sqrt{3} + 3 = 2(2-\sqrt{3})$

Na mocy II ZIM oba wyznaczone wzory (tj. wzór rekurencyjny i wzór jawny) definiują ten sam ciąg (t_n).

[10] Ciąg Fibonacciego określony jest zależnością $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + +F_{n-2}$ dla $n > 1$. Wyznacz wzór jawny na n-ty wyraz ciągu na podstawie wielomianu charakterystycznego oraz z pomocą funkcji tworzącej.

Rozwiązanie:
Wielomian charakterystyczny: $f(x) = x^2 - x - 1 = 0$
$\Delta = 1 + 4 = 5$, $\sqrt{\Delta} = \sqrt{5}$, $x_1 = \frac{1-\sqrt{5}}{2}$, $x_2 = \frac{1+\sqrt{5}}{2}$,

$$F_n = \mathrm{A}_0 \cdot \left(\tfrac{1-\sqrt{5}}{2}\right)^n + \mathrm{B}_0 \cdot \left(\tfrac{1+\sqrt{5}}{2}\right)^n$$

$n = 0$: $F_0 = 0 = \mathrm{A}_0\left(\tfrac{1-\sqrt{5}}{2}\right)^0 + \mathrm{B}_0\left(\tfrac{1+\sqrt{5}}{2}\right)^0 = \mathrm{A}_0 + \mathrm{B}_0$, stąd $\mathrm{A}_0 = -\mathrm{B}_0$

$n = 1$: $F_1 = 1 = \mathrm{A}_0\left(\tfrac{1-\sqrt{5}}{2}\right)^1 + \mathrm{B}_0\left(\tfrac{1+\sqrt{5}}{2}\right)^1 = \mathrm{B}_0\sqrt{5}$, stąd $\mathrm{B}_0 = \frac{1}{\sqrt{5}}$, $\mathrm{A}_0 = -\frac{1}{\sqrt{5}}$

Ostatecznie: $F_n = -\frac{1}{\sqrt{5}} \cdot \left(\tfrac{1-\sqrt{5}}{2}\right)^n + \frac{1}{\sqrt{5}} \cdot \left(\tfrac{1+\sqrt{5}}{2}\right)^n \ \forall n \in N$.
Udowodnimy, że oba wzory (rekurencyjny i jawny) definiują ten sam ciąg (F_n).
Dowód indukcyjny (II schemat indukcji)
Jeśli wyrazy ciągu (F_n) spełniają warunek

$$\begin{cases} F_0 = 0, F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ dla } n > 1 \end{cases} \text{ to } \forall n \in N \; F_n = \tfrac{1}{\sqrt{5}} \cdot \left(\tfrac{1+\sqrt{5}}{2}\right)^n - \tfrac{1}{\sqrt{5}} \cdot \left(\tfrac{1-\sqrt{5}}{2}\right)^n$$

(1) baza indukcji: pokażemy prawdziwość wzoru jawnego dla $n = 0$

$$F_0 = 0, F_0 = \tfrac{1}{\sqrt{5}} \cdot \left(\tfrac{1+\sqrt{5}}{2}\right)^0 - \tfrac{1}{\sqrt{5}} \cdot \left(\tfrac{1-\sqrt{5}}{2}\right)^0 = \tfrac{1}{\sqrt{5}}(1-1) = 0$$

(2) krok indukcyjny:

Zakładamy prawdziwość wzoru jawnego dla wszystkich liczb naturalnych mniejszych od $k+2$, gdzie k jest pewną liczbą naturalną. Pokażemy, że wówczas wzór jawny jest słuszny również dla $k+2$.

Dla uproszczenia zapisu wprowadzamy oznaczenia:

$$\mathrm{w} = \frac{1}{\sqrt{5}}, \mathrm{a} = \frac{1+\sqrt{5}}{2}, \mathrm{b} = \frac{1-\sqrt{5}}{2}$$

Wówczas założenie indukcyjne ma postać: $F_i = \mathrm{w}(\mathrm{a}^{\,i} - \mathrm{b}^{i})\ \forall_{i<k+2}$

Należy pokazać, że: $F_{k+2} = \mathrm{w}(\mathrm{a}^{\,k+2} - \mathrm{b}^{k+2})$

$$\mathrm{L}_{k+2} = F_{k+2} \overset{\text{def.rek}}{=} F_{k+1} + F_k \overset{\text{zał.ind}}{=} \mathrm{w}(\mathrm{a}^{\,k+1} - \mathrm{b}^{k+1}) + \mathrm{w}(\mathrm{a}^{\,k} - \mathrm{b}^{k}) =$$

$$= \mathrm{w}(\mathrm{a}^{\,k}(\mathrm{a}+1) - \mathrm{b}^{k}(\mathrm{b}+1)) =^{(\mathrm{a})} \mathrm{w}(\mathrm{a}^{\,k}\mathrm{a}^{\,2} - \mathrm{b}^{k}\mathrm{b}^{\,2}) =$$

$$= \mathrm{w}(\mathrm{a}^{\,k+2} - \mathrm{b}^{k+2}) = \mathrm{P}_{k+2}$$

$$1 + \mathrm{a} = 1 + \frac{1+\sqrt{5}}{2} = \frac{3+\sqrt{5}}{2}, \qquad 1 + \mathrm{b} = 1 + \frac{1-\sqrt{5}}{2} = \frac{3-\sqrt{5}}{2}$$

$$\mathrm{a}^2 = \frac{1+2\sqrt{5}+5}{4} = \frac{3+\sqrt{5}}{2}, \qquad \mathrm{b}^2 = \frac{1-2\sqrt{5}+5}{4} = \frac{3-\sqrt{5}}{2}$$

Stąd (a) $(1+\mathrm{a}) = \mathrm{a}^2$, $(1+\mathrm{b}) = \mathrm{b}^2$.

Na mocy II ZIM podana zależność rekurencyjna oraz wyznaczony wzór jawny definiują ten sam ciąg (F_n).

Wyznaczamy wzór jawny z pomocą funkcji tworzącej:

$$f(x) = \sum_{n=0}^{\infty} a_n x^n = x + \sum_{n=2}^{\infty} a_n x^n = x + \sum_{n=0}^{\infty} a_{n+2} x^{n+2} =$$

$$= x + \sum_{n=0}^{\infty} (a_{n+1} + a_n) x^{n+2} = x + x\sum_{n=0}^{\infty} a_{n+1} x^{n+1} + x^2 \sum_{n=0}^{\infty} a_n x^n =$$

$$= x + xf(x) + x^2 f(x)$$

$$f(x)(1 - x - x^2) = x$$

$$f(x) = \frac{x}{1-x-x^2} =^{(\mathrm{a})} \frac{\mathrm{A}}{1-\alpha x} + \frac{\mathrm{B}}{1-\beta x} =^{(\mathrm{b})} \frac{\mathrm{A}}{1-\frac{1+\sqrt{5}}{2}x} + \frac{\mathrm{B}}{1-\frac{1-\sqrt{5}}{2}x} =^{(\mathrm{c})}$$

$$= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\frac{1+\sqrt{5}}{2}x} - \frac{1}{1-\frac{1-\sqrt{5}}{2}x}\right)$$

(a) Rozkładam ułamek na sumę ułamków prostych.

(b) $1 - x - x^2 = (1-\alpha x)(1-\beta x) = 1 - (\alpha+\beta)x + \alpha\beta x^2$

$$1 = \alpha + \beta \Rightarrow \alpha = 1 - \beta \qquad \Rightarrow \alpha_1 = \frac{1+\sqrt{5}}{2}, \alpha_2 = \frac{1-\sqrt{5}}{2},$$

$$-1 = \alpha\beta \Rightarrow \beta^2 - \beta - 1 = 0, \Delta = 1 + 4 = 5, \sqrt{\Delta} = \sqrt{5} \Rightarrow \beta_1 = \frac{1-\sqrt{5}}{2}, \beta_2 = \frac{1+\sqrt{5}}{2},$$

(c) $x = \mathrm{A}\left(1 - \frac{1-\sqrt{5}}{2}x\right) + \mathrm{B}\left(1 - \frac{1+\sqrt{5}}{2}x\right)$

$2x = (-\mathrm{A} + \sqrt{5}\mathrm{A} - \mathrm{B} - \sqrt{5}\mathrm{B})x \Rightarrow 2 = \mathrm{B} - \sqrt{5}\mathrm{B} - \mathrm{B} - \sqrt{5}\mathrm{B} \Rightarrow \mathrm{B} = \frac{-1}{\sqrt{5}}$

$0 = 2\mathrm{A} + 2\mathrm{B} \Rightarrow \mathrm{A} = -\mathrm{B} \Rightarrow \mathrm{A} = \frac{1}{\sqrt{5}}$

Wzór jawny na n-ty wyraz ciągu: $a_n = F_n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n$

6.1. Zadania programistyczne

[11] (Przeglądanie grafu w głąb) Zaimplementuj algorytm sprawdzający, czy zadany graf nieskierowany, bez pętli (tj. krawędzi postaci (x, x)), reprezentowany tablicą list incydencji jest spójny metodą przeglądania w głąb. Wypisz kolejno odwiedzone wierzchołki. Wskazówka: (por. [LW], s. 78).

Przykład: Dla grafu zadanego poniższą tablicą list incydencji kolejność odwiedzania wierzchołków jest następująca: 1, 2, 3, 8, 4, 6, 5, 7. Graf jest spójny.
[1] 2 4 5 6
[2] 1 3 4 8
[3] 2 8
[4] 1 2 6 8
[5] 1 6
[6] 1 4 5 7 8
[7] 6 8
[8] 2 3 4 6 7

[12] Napisz rekurencyjną funkcję, która rysuje (zaczynając od górnego poziomu) przedstawiony na rysunku trójkąt o podstawie n ($n \in N$, $1 \le n \le 30$), zbudowany ze znaków '*'. Trójkąt jest symetryczny względem jego wysokości.

```
      *
     * * *
    * * * * *
   * * * * * * *
```

Rys. 2.8. Trójkąt dla $n = 7$

[13] (Magiczna kostka) Wypełnij kostkę sześcienną o rozmiarze $n \times n \times n$ tekstem T (podanym przez użytkownika) w następujący sposób.

Sześcian składa się z n tablic o rozmiarze $n \times n$, ustawionych jedna za drugą. Każda tablica stanowi magiczny kwadrat wypełniony (w odpowiedni sposób) literami tekstu T. Tekst T jest powielany, tzn. dla T = ABCDEFGH! generowany jest ciąg:

T = ABCDEFGH!ABCDEFGH!ABCDEFGH!...,

którego kolejne znaki są wykorzystywane w nieprzerwany sposób do wypełniania wszystkich (od 0 do (n–1)-tej) tablic sześcianu. Tablice o numerach parzystych (dla i = 0, 2, 4, 6, ...) uzupełniane są literami ciągu T zgodnie z podanym w przykładzie poniżej wzorcem (tj. po każdorazowym rekurencyjnym wywołaniu funkcji wypełniającej ramkę, elementy ciągu T powinny być umieszczone na obwodzie kwadratu w następującym porządku: górna krawędź (od lewej do prawej), prawa krawędź (od góry do dołu), dolna krawędź (od prawej do lewej), lewa krawędź (od dołu do góry)). Tablice o numerach nieparzystych (tj. dla i = 1, 3, 5, ...) są wypełniane przy wykorzystaniu tej samej drogi, ale kolejne litery ciągu umieszczane są na niej w odwrotnej kolejności, tj. zaczynamy wypełnianie kwadratu od jego środka (patrz przykład poniżej). Wszystkie kwadraty należy wypełnić literami w taki sposób, aby ciąg T nie został przerwany w żadnym miejscu (tj. jeśli generując tablicę 0, zakończyliśmy na znaku G, to tablicę 1 rozpoczynamy zapełniać od znaku H oraz litery H i G muszą sąsiadować ze sobą w sześcianie).

Przykład:

```
T = ABCDEFGH!, n = 5

Tablica 0       Tablica 1       Tablica 2       Tablica 3       Tablica 4
[A, B, C, D, E] [E, D, C, B, A] [F, G, H, !, A] [A, !, H, G, F] [B, C, D, E, F]
[G, H, !, A, F] [H, G, F, E, !] [C, D, E, F, B] [D, C, B, A, E] [H, !, A, B, G]
[F, F, G, B, G] [!, !, H, D, H] [B, B, C, G, C] [E, E, D, !, D] [G, G, H, C, H]
[E, E, D, C, H] [A, A, B, C, G] [A, A, !, H, D] [F, F, G, H, C] [F, F, E, D, !]
[D, C, B, A, !] [B, C, D, E, F] [!, H, G, F, E] [G, H, !, A, B] [E, D, C, B, A]
```

Rys. 2.9. Magiczna kostka dla $n = 5$

Rozdział 3. Technika projektowania algorytmów „dziel i zwyciężaj”

„*Dziel i zwyciężaj*” to najbardziej popularna technika efektywnego projektowania algorytmów. Polega na podzieleniu problemu P na podproblemy tego samego typu, ale o mniejszych rozmiarach. Te mniejsze podproblemy są dalej dzielone na mniejsze i mniejsze, aż osiągną rozmiar tak mały, że ich rozwiązanie jest oczywiste lub mało kosztowne. Następnie wyznaczone rozwiązania podbroblemów są łączone w celu uzyskania rozwiązania pierwotnego problemu P.

1. Technika „dziel i zwyciężaj” (divide and conquer) (por. [WP])

Technika „*dziel i zwyciężaj*” jest realizowana w dwóch etapach i polega na:
(1) dekompozycji problemu n-wymiarowego na skończoną liczbę problemów tego samego rodzaju, ale o mniejszych rozmiarach,
(2) „scaleniu” otrzymanych wyników częściowych w celu uzyskania rozwiązania problemu pierwotnego.

Schemat („dziel i zwyciężaj”), PB(n) – problem rozmiaru n

```
PB(n) {
 if(n jest wystarczająco małe) return przypadek_elementarny(n);
 else {
  dekompozycja PB(n) na PB(n1), ... , PB(nk);
  for(int i = 1; i <= k; i++) wi = PB(i);
```

```
    Łącz(w1, ..., wk);
  }
}
```

W przypadku pewnej klasy problemów zastosowanie strategii „dziel i zwyciężaj" obniża złożoność algorytmu z $O(n)$ do $O(\lg n)$.

2. Wyszukiwanie binarne „BinarySearch" (por. [BDR])

Problem 1. Dany jest n-elementowy uporządkowany ciąg (a_n) liczb naturalnych oraz wartość $x \in N$. Zaproponuj dwa algorytmy (w wersji iteracyjnej i rekurencyjnej) zwracające pozycję wartości x w zadanym ciągu lub wartość –1, gdy x w ciągu nie występuje. Wykorzystaj technikę „dziel i zwyciężaj". Wyznacz złożoność pesymistyczną czasową obu funkcji. Przyjmij za operację elementarną porównanie elementu ciągu i wartości x.

```
int BSIt(int x, int *A, int n) {
 int m = 0, left = 0, right = n – 1;
 while(left <= right) {
  m = (left + right) / 2;
  if(A[m] == x) return m;
  if(A[m] > x) right = m – 1;
  else left = m + 1;
 }
 return –1;
}
```

```
int BSRek(int left, int right, int x, int *A) {
 int m = 0;
 if(left <= right) {
  m = (left + right) / 2;
  if(A[m] == x) return m;
  if(A[m] > x) return BSRek(left, m – 1, x, A);
  return BSRek(m + 1, right, x, A);
 }
 return –1;
}
```

Przykład 1. Realizacja algorytmu dla 12-elementowego ciągu liczb i wartości $x = 11$.

```
i      0  1  2  3  4  5  6  7  8  9  10 11
A[i]:  1  3  5  6  6  8  11 14 16 18 19 24
```

Ciąg kolejno porównywanych z liczbą $x = 11$ wartości tablicy A to: A[5]=8, A[8]=16, A[6]=**11**.

[Złożoność pesymistyczna] (algorytm iteracyjny BSIt( ) i rekurencyjny BSRek( ))

- Przyjmijmy za operację elementarną porównanie elementu ciągu i wartości x.
- Załóżmy, że $n = 2^k$ dla $k \in N$.

Złożoność algorytmu iteracyjnego BSIt( ):

- Algorytm w przypadku pesymistycznym wykona iteracje dla następujących długości podciągów: n, $n/2$, $n/4$, $n/8$, $n/16$, ..., 2, 1, tzn. wykona $1 + \lg_2 n$ iteracji (por. [05], rozdział 1).
- W każdej iteracji algorytm wykona dwa porównania.

$$T_{\max}(\text{BSIt}, n) = 2(1 + \lg_2 n) = \Theta(\lg_2 n)$$

Złożoność algorytmu rekurencyjnego BSRek( ):

$$T_{\max}(n) = \begin{cases} 2 & \text{dla } n = 1 \\ T_{\max}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 & \text{dla } n > 1 \end{cases}$$

Ponieważ $n = 2^k$ dla $k \in N$, więc

$$T_{\max}(n) = T_{\max}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2 =$$
$$= T_{\max}\left(2^{k-1}\right) + 2 =$$
$$= T_{\max}\left(2^{k-2}\right) + 2 + 2 = \cdots =$$
$$= T_{\max}\left(2^{k-k}\right) + \underbrace{2 + \cdots + 2}_{k \text{ razy}} =$$
$$= 2 + 2k = 2(1 + \lg_2 n) = \Theta(\lg_2 n).$$

- Złożoność pesymistyczna algorytmu wyszukiwania elementu x w ciągu (a_n) metodą sekwencyjną (naiwną) jest liniowa w stosunku do długości n ciągu, tj. $T_{\max}(n) = \Theta(n)$.

Wniosek. „Dziel i zwyciężaj" usprawnia wyszukiwanie elementu w ciągu uporządkowanym.

3. Metoda bisekcji

Problem 2. Wiadomo, że funkcja f jest ciągła i ściśle monotoniczna na przedziale $[a, b]$ oraz, że wartości funkcji f na końcach przedziału mają różne znaki. Zaproponuj algorytm wyznaczający (metodą bisekcji) przybliżony pierwiastek równania $f(x) = 0$ w przedziale $[a, b]$ z dokładnością E.

Twierdzenie 1 (*Darboux*)
Jeśli funkcja f jest ciągła na przedziale $[a, b]$ oraz spełnia warunek $f(a)f(b) < 0$ (tzn. wartości funkcji na końcach przedziału mają różne znaki), to istnieje punkt $c \in (a, b)$ taki, że $f(c) = 0$. Jeśli dodatkowo funkcja f jest ściśle monotoniczna (tzn. rosnąca albo malejąca) w $[a, b]$, to punkt c jest jedyny.

Idea (metoda bisekcji)
Metoda bisekcji wykorzystuje strategię „dziel i zwyciężaj".
Zakładamy, że w $[a, b]$ istnieje dokładnie jeden pierwiastek równania $f(x) = 0$, tzn. spełnione są założenia twierdzenia Darboux.
Podczas gdy długość przedziału $[a, b]$ jest większa niż E, powtarzaj kroki (a)–(d):
(a) Wyznacz środek przedziału $[a, b]$: $x_1 = \frac{a+b}{2}$.
(b) Jeśli x_1 jest pierwiastkiem równania $f(x) = 0$, to go zwróć.

(c) Jeśli pierwiastek równania $f(x) = 0$ jest mniejszy niż x_1 (tzn. $f(a)f(x_1) < 0$), to za b przyjmij x_1.
(d) W przeciwnym razie (tzn. gdy $f(x_1)f(b) < 0$) za a przyjmij x_1.

```
double f(double x) { return pow(x, 2) – 2; }

double Bisekcja(double a, double b, double E) {
 double x = (a + b) / 2.0;
 while(fabs(b – a) > E) {
  x = (a + b) / 2.0;
  if(f(x) == 0) return x;
  if(f(a) * f(x) < 0) b = x;
  else a = x;
 }
 return x;
}
```

Przykład 2. Realizacja algorytmu Bisekcja( ) dla danych $[a, b] = [-1, 5]$, $\mathrm{E} = 0.25$ oraz funkcji $f(x) = x^2 - 2$.

	$a = -1.0$,	$b = 5.0$,	$\lvert b - a\rvert = 6$,
$x = 2.0$,	$a = -1.0$,	$\underline{b = 2.0}$,	$\lvert b - a\rvert = 3$,
$x = 0.5$,	$\underline{a = 0.5}$,	$b = 2.0$,	$\lvert b - a\rvert = 1.5$,
$x = 1.25$,	$\underline{a = 1.25}$,	$b = 2.0$,	$\lvert b - a\rvert = 0.75$,
$x = 1.625$,	$a = 1.25$,	$\underline{b = 1.625}$,	$\lvert b - a\rvert = 0.375$,
$x = 1.4375$,	$a = 1.25$,	$\underline{b = 1.4375}$,	$\lvert b - a\rvert = 0.1875 < \mathrm{E} = 0.25$,
$x = 1.4375$			

4. Wyszukiwanie minimum i maksimum

Problem 3. (Wyszukiwanie minimum i maksimum w tablicy liczb) (por. [WP]) Zaproponuj dwa algorytmy: sekwencyjny (naiwny) i z wykorzystaniem techniki „dziel i zwyciężaj” (w wersji rekurencyjnej) wyszukiwania najmniejszego i największego elementu w tablicy liczb. Wyznacz złożoność pesymistyczną czasową obu algorytmów. Przyjmij za operację elementarną porównanie elementów ciągu. Określ rzędy wielkości wyznaczonych funkcji złożoności oraz na ich podstawie wyciągnij odpowiednie wnioski.

(a) Algorytm MinMax1( ) „sekwencyjny”:

```
class MM {
 public:
 int min, max;
};

void MinMax1(MM &M, int *A, int n) {
 M.min = M.max = A[0];
 for(int i = 1; i < n; i++) {
  if(M.min > A[i]) M.min = A[i];
  if(M.max < A[i]) M.max = A[i];
 }
}
```

(b) Algorytm MinMax2( ) „dziel i zwyciężaj”:

```
void MinMax2(int left, int right, MM &M, int *A) {
 if(left == right) M.min = M.max = A[left];
 else
  if(left + 1 == right)
   if(A[left] < A[right]) {
    M.min = A[left]; M.max = A[right];
   } else { M.min = A[right]; M.max = A[left]; }
  else {
   MM *M1 = new MM( );
   MM *M2 = new MM( );
   int m = (left + right) / 2;
   MinMax2(left, m, *M1, A);
   MinMax2(m + 1, right, *M2, A);
   M.min = (M1->min < M2->min) ? M1->min : M2->min;
   M.max = (M1->max > M2->max) ? M1->max : M2->max;
  }
}
```

[Złożoność pesymistyczna] (algorytm MinMax1( ) i MinMax2( ))

(a) MinMax1( )

$$T_{\max}(n) = 2(n-1) = \Theta(n)$$

(b) MinMax2( )

$$T_{\max}(n) = \begin{cases} 0 & \text{dla } n = 1 \\ 1 & \text{dla } n = 2 \\ T_{\max}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{\max}\left(\left\lceil \frac{n}{2} \right\rceil\right) + 2 & \text{dla } n > 2 \end{cases}$$

Załóżmy, że $n = 2^k$ dla $k \in N$.

$$T_{\max}(n) = 2T_{\max}\left(2^{k-1}\right) + 2 = 2\left[2T_{\max}\left(2^{k-2}\right) + 2\right] + 2 =$$
$$= 2^2 T_{\max}\left(2^{k-2}\right) + 2^2 + 2 = 2^2\left[2T_{\max}\left(2^{k-3}\right) + 2\right] + 2^2 + 2 =$$
$$= 2^3 T_{\max}\left(2^{k-3}\right) + 2^3 + 2^2 + 2 = \cdots =$$
$$= 2^{k-1} T_{\max}\left(2^{k-(k-1)}\right) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 =$$
$$= 2^{k-1} + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 =$$
$$= 2^{k-1} + 2\frac{1-2^{k-1}}{1-2} = 2^{k-1} + 2^k - 2 = \frac{3n-4}{2} = \Theta(n)$$

Wniosek. W przypadku niektórych problemów algorytmicznych technika „dziel i zwyciężaj" nie poprawia złożoności rozwiązań.

5. Sortowanie przez scalanie (MergeSort( )) (por. [BDR], [DS])

Idea algorytmu (sortowanie MergeSort( ))
Algorytm MergeSort( ) jest oparty na strategii „dziel i zwyciężaj".

- W funkcji MergeSort( ) dzielimy ciąg (tablicę) na dwa równe podciągi, oba sortujemy oddzielnie (rekurencyjnie wywołując MergeSort( ) dla każdego z nich), a następnie łączymy dwa uporządkowane podciągi (w funkcji Merge( )) w jeden posortowany ciąg.
- Proces dzielenia tablicy na połówki kończy się, gdy zawiera ona mniej niż dwa elementy.

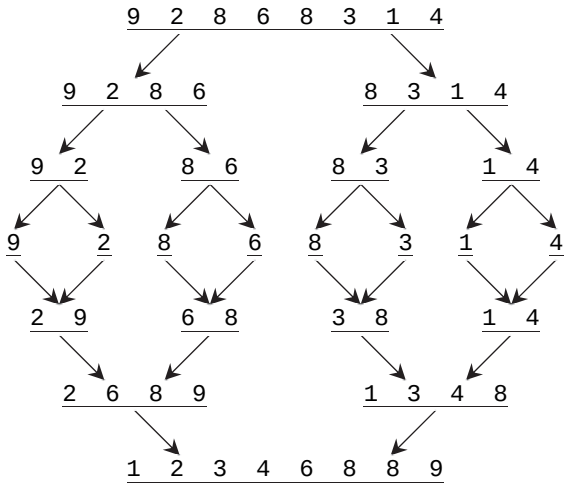
```
void Merge(int left, int m, int right, int *A) {
 int i = left, j = m + 1, k = 0;
 int *P = new int[MAX];
 while(i <= m && j <= right)
  P[k++] = (A[i] <= A[j]) ? A[i++] : A[j++];
 if(i <= m)
  for(int z = i; z <= m; z++) P[k++] = A[z];
 if(j <= right)
  for(int z = j; z <= right; z++) P[k++] = A[z];
 for(int z = left; z <= right; z++) A[z] = P[z–left];
 delete [ ]P;
}
```

```
void MergeSort(int left, int right, int *A) {
 if(left < right) {
  int m = (left + right) / 2;
  MergeSort(left, m, A);
  MergeSort(m + 1, right, A);
  Merge(left, m, right, A);
 }
}
```

```
->MS(0,8)
  ->MS(0,4)
    ->MS(0,2)
      ->MS(0,1)
        ->MS(0,0)
        ->MS(1,1)
        ->M(0,0,1)
      ->MS(2,2)
      ->M(0,1,2)
    ->MS(3,4)
        ->MS(3,3)
        ->MS(4,4)
        ->M(3,3,4)
    ->M(0,2,4)
  ->MS(5,8)
    ->MS(5,6)
        ->MS(5,5)
        ->MS(6,6)
        ->M(5,5,6)
    ->MS(7,8)
        ->MS(7,7)
        ->MS(8,8)
        ->M(7,7,8)
    ->M(5,6,8)
  ->M(0,4,8)
```

Rys. 3.1. Łańcuch wywołań rekurencyjnych dla ciągu długości $n = 9$

Przykład 3. Realizacja algorytmu MergeSort( ) dla ciągu liczb 9, 2, 8, 6, 8, 3, 1, 4.



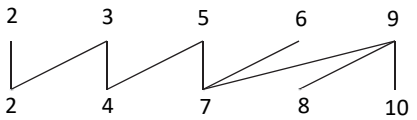
Rys. 3.2. Ilustracja działania algorytmu MergeSort( )

[Złożoność pesymistyczna] (algorytm MergeSort( )) (por. [DA], [DS])

- Przyjmijmy za operację elementarną porównanie elementów tablicy.
- n oznacza rozmiar danych (rozmiar tablicy).
- Pesymistyczną złożoność funkcji Merge(*left*, *m*, *right*, A) szacujemy przez $O(n)$.

Uzasadnienie:

- Efektem działania funkcji Merge(*left*, *m*, *right*, A) jest uporządkowany fragment tablicy A[*left..right*] utworzony z posortowanych połówek A[*left..m*] i A[*m*+1..*right*] długości odpowiednio $n_1 = m - left + 1$ i $n_2 = right - m$.
- Pesymistyczna liczba porównań przy scalaniu dwóch uporządkowanych fragmentów tablicy ma miejsce w każdej z dwóch następujących sytuacji: (1) gdy wyjście z pętli następuje dla $i = m + 1$, podczas gdy $j = right$ bądź (2) gdy wyjście z pętli następuje dla $j = right + 1$, podczas gdy $i = m$, tzn., gdy jedna z tablic została już wykorzystana, a w drugiej został dokładnie jeden element.



Kreski oznaczają porównania.
Dla ciągów długości $\lfloor \frac{n}{2} \rfloor$ i $\lceil \frac{n}{2} \rceil$ jest ich maksymalnie $n - 1$.

$$T_{\max}(n_1, n_2) = n_1 + n_2 - 1 = m - \text{left} + 1 + \text{right} - m - 1 = \text{right} - \text{left}$$

Dla $left = 0$ i $right = n - 1$ mamy $T_{\max}(n_1, n_2) = n - 1$.

- Pesymistyczną złożoność algorytmu MergeSort( ) szacujemy przez $O(n\lg n)$.

Uzasadnienie:

$$T_{\max}(n) = \begin{cases} 0 & \text{dla } n = 1 \\ T_{\max}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T_{\max}\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underbrace{n-1}_{\text{scalanie}} & \text{dla } n > 1 \end{cases}$$

Załóżmy, że $n = 2^k$ dla $k \in N$.

$$T_{\max}(n) = 2T_{\max}\left(\frac{n}{2}\right) + n - 1 = 2T_{\max}\left(2^{k-1}\right) + 2^k - 1 =$$
$$= 2\left[2T_{\max}\left(2^{k-2}\right) + 2^{k-1} - 1\right] + 2^k - 1 =$$
$$= 2^2 T_{\max}\left(2^{k-2}\right) + 2^k - 2 + 2^k - 1 =$$
$$= 2^2\left[2T_{\max}\left(2^{k-3}\right) + 2^{k-2} - 1\right] + 2^k - 2 + 2^k - 1 =$$
$$= 2^3 T_{\max}\left(2^{k-3}\right) + 2^k - 2^2 + 2^k - 2 + 2^k - 1 =$$
$$= 2^3\left[2T_{\max}\left(2^{k-4}\right) + 2^{k-3} - 1\right] + 2^k - 2^2 + 2^k - 2 + 2^k - 1 =$$
$$= 2^4 T_{\max}\left(2^{k-4}\right) + 2^k - 2^3 + 2^k - 2^2 + 2^k - 2 + 2^k - 1 = \cdots =$$
$$= 2^k \underbrace{T_{\max}\left(2^{k-k}\right)}_{0} + 2^k - 2^{k-1} + \cdots + 2^k - 2^2 + 2^k - 2 + 2^k - 1 =$$
$$= k2^k - \left(2^{k-1} + \cdots + 2^2 + 2^1 + 2^0\right) = k2^k - \left(\frac{1-2^k}{1-2}\right) = k2^k - 2^k + 1 =$$
$$= n\lg n - n + 1 = O(n\lg n)$$

6. Sortowanie szybkie (QuickSort( )) (por. [BDR], [CLR], [DA], [DS])

Algorytm QuickSort( ) jest jednym z najszybszych algorytmów sortowania dużych, „losowych" tablic.

Idea (algorytm QuickSort( ))
Algorytm QuickSort( ) jest oparty na strategii „dziel i zwyciężaj".

- Wektor wejściowy A zostaje odpowiednio przygotowany i rozdzielony na dwie części, które są niezależnie sortowane poprzez rekurencyjne wywołanie QuickSort( ) dla każdej z nich.
- Główne zadanie sortowania odbywa się w funkcji partition( ), po której zakończeniu podwektor A[*left*..*right*] spełnia następujące własności:
  - *v* – element rozdzielający,
  - A[*j*] = *v* – element zajmujący właściwą pozycję w posortowanym wektorze,
  - A[*left*..*j*–1] – elementy mniejsze bądź równe *v*,
  - A[*j*+1..*right*] – elementy większe bądź równe *v*.

Działanie funkcji partition(*left*, *right*) (*left* – lewy indeks wektora A[*left*..*right*], *right* – prawy indeks wektora A[*left*..*right*])

(1) v = A[*left*] – Ustalamy element rozdzielający (v może być wybrany w dowolny sposób).
(2) Przeglądamy indeksem i wektor A[*left*+1..*right*] od strony lewej do prawej w celu znalezienia elementu większego lub równego v.
(3) Przeglądamy indeksem j wektor A[*left*..*right*] od strony prawej do lewej w celu znalezienia elementu mniejszego lub równego v.
(4) Jeżeli ($i < j$), zamieniamy miejscami A[i] z A[j].
(5) Powtarzamy kroki (2)–(4) do momentu, aż uzyskamy $i \geq j$.
(6) Zamieniamy element A[*left*] = v z elementem A[j].

Przykład 4. Realizacja funkcji partition( ) dla $n = 9$.

```
    i               j
6 4 8 5 9 7 2 3 1
          i     j
6 4 1 5 9 7 2 3 8
            i j
6 4 1 5 3 7 2 9 8
            j i
6 4 1 5 3 2 7 9 8

2 4 1 5 3 6 7 9 8

2 4 1 5 3 6 7 9 8
```

```
void QuickSort(int left, int right, int *A) {
 if(left < right) {
   int v = A[left], i = left, j = right + 1, pom;          // początek partition( )
   do {
     do { i++; } while(A[i] < v && i < right);
     do { j--; } while(A[j] > v);
     if(i < j) { pom = A[i]; A[i] = A[j]; A[j] = pom; }
   } while(i < j);
   A[left] = A[j]; A[j] = v;                               // koniec partition( )
   QuickSort(left, j – 1, A);
   QuickSort(j + 1, right, A);
 }
}
```

- Efektywność algorytmu QuickSort( ) zależy od wyboru elementu rozdzielającego, który ma wpływ na podział ciągu na dwa podciągi.

[Złożoność pesymistyczna] (algorytm QuickSort( ))

- Pesymistyczny przypadek podziałów następuje, gdy za każdym razem na element rozdzielający wybierana jest wartość najmniejsza albo największa. Taka sytuacja ma miejsce, gdy np. sortowany jest ciąg uporządkowany [1, 2, 3, 4, 5, 6, 7, 8] oraz *v* = A[*left*]. Wówczas funkcja partition( ) za każdym razem dzieli wektor o długości $n_i = right_i - left_i + 1$ na podwektory o długości 0 i $n_i - 1$, wykonując przy tym $n_i + 1$ porównań. Wówczas:

  $$T_{\max}(n) = \begin{cases} 0 & \text{dla } n = 1 \\ 3 & \text{dla } n = 2 \\ T_{\max}(n-1) + (n+1) & \text{dla } n > 2 \end{cases}$$

  Załóżmy, że $n = 2^k$ dla $k \in N$.

  $T_{\max}(n) = (n+1) + n + (n-1) + \cdots + 3 = \frac{(n+4)(n-1)}{2} = \Theta(n^2)$

[Złożoność średnia] (algorytm QuickSort( ))

- Optymistyczny przypadek podziałów zachodzi wtedy, gdy funkcja partition( ) za każdym razem dzieli ciąg długości $n_i = right_i - left_i + 1$ na dwa podciągi o rozmiarach $\left\lfloor \frac{n_i - 1}{2} \right\rfloor$ i $\left\lceil \frac{n_i - 1}{2} \right\rceil$ elementów. Wówczas we wszystkich podziałach razem dokonuje się około

  $$T(n) \approx n + 2\left(\frac{n}{2}\right) + 4\left(\frac{n}{4}\right) + 8\left(\frac{n}{8}\right) + \cdots + n\left(\frac{n}{n}\right) = n\left(\underbrace{1 + \cdots + 1}_{\approx 1 + \lg_2 n}\right) = \mathrm{O}(n \lg n)$$

  porównań.

- Obliczenia wykazują, że w przypadku średnim również trzeba wykonać około $\mathrm{O}(n \lg n)$ porównań.
- Aby zoptymalizować algorytm, tj. aby funkcja partition( ) zwracała podwektory mniej więcej tego samego rozmiaru, sugeruje się, aby przyjąć za element rozdzielający (1) losowy element z wektora A[*left..right*] lub (2) medianę trzech elementów (pierwszego, środkowego i ostatniego).

  Nie należy też stosować algorytmu QuickSort( ) do tablic o małych rozmiarach.

7. *k*-ty największy element (algorytm Hoare'a) (por. [AHU], [BDR])

Problem 4. Należy znaleźć k-ty największy element ciągu (a_n).
Idea (algorytm Hoare'a)

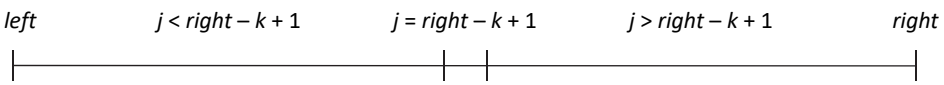
- Algorytm Hoare( ) wyznacza k-ty największy element ciągu (a_n) i korzysta z funkcji partition( ) identycznej jak w algorytmie QuickSort( ).
- Funkcja partition( ) ustawia element rozdzielający *v* na właściwą (ostateczną) *j*-tą pozycję (H[*j*] = *v*) w posortowanym ciągu. Następnie algorytm sprawdza,

w której części tablicy (lewej czy prawej) znajduje się szukany k-ty największy element i dla tej części tablicy wywoływana jest rekurencyjnie funkcja Hoare( ). Uściślając, jeśli $k = right - j + 1$, to H[j] jest k-tym największym elementem ciągu, jeśli $k < right - j + 1$, to element k-ty znajduje się po jego prawej stronie, w przeciwnym przypadku k-ty element znajduje się po lewej stronie elementu H[j].

```
int Hoare(int k, int left, int right, int *H) {
 if(left < right) {
   int v = H[left], i = left, j = right + 1;          // początek partition( )
   do {
     do { i++; } while(H[i] < v && i < right);
     do { j--; } while(H[j] > v);
     if(i < j) swap(H[i], H[j]);
   } while(i < j);
   H[left] = H[j]; H[j] = v;                          // koniec partition( )
   if(k == right – j + 1) return j;
   if(k < right – j + 1) return Hoare(k, j + 1, right, H);
   return Hoare(k – (right – j + 1), left, j – 1, H);
 }
 return left;
}
```

Przykład 5. Realizacja algorytmu Hoare( ) dla $k = 9$ i ciągu długości 12. Elementem rozdzielającym jest H[*left*] – pierwszy element podciągu H[*left..right*].

```
  i    0  1  2  3  4  5  6  7  8  9  10 11
H[i]:  9  3  15 11 6  8  18 12 16 7  29 4
H[i]:  8  3  4  7  6  9  18 12 16 11 29 15, k=9>11-5+1=7, więc k=9-7=2
H[i]:  6  3  4  7  8  9  18 12 16 11 29 15, k=2>4-4+1=1,  więc k=2-1=1
H[i]:  4  3  6  7  8  9  18 12 16 11 29 15, k=1<3-2+1=2,  więc k=1
H[i]:  4  3  6  7  8  9  18 12 16 11 29 15, k=1=3-3+1=1,  więc H[3]=7
                                           H[3]=7 – 9-ty największy element
```



Rys. 3.3. Ilustracja (podział rekurencyjny w algorytmie Hoare'a)

- Efektywność algorytmu zależy od wyboru elementu rozdzielającego.

[Złożoność] (algorytm Hoare'a) (por. [AHU])

- Najgorszy przypadek ma miejsce, gdy np. za każdym razem na element rozdzielający wybierana jest wartość najmniejsza (v = H[*left*] dla ciągu uporządkowanego [1, 2, 3, 4, 5, 6, 7, 8]) w sytuacji, gdy szukamy największego elementu ($k = 1$). Wówczas:

$$T_{\max}(n) = \begin{cases} 0 & \text{dla } n = 1 \\ 3 & \text{dla } n = 2 \\ T_{\max}(n-1) + (n+1) & \text{dla } n > 2 \end{cases}$$

  Załóżmy, że $n = 2^k$ dla $k \in N$, wówczas:

$$T_{\max}(n) = (n+1) + n + (n-1) + \cdots + 3 = \frac{(n+4)(n-1)}{2} = \Theta(n^2)$$

- W optymistycznym przypadku zostanie wykonanych n porównań.
- Oczekiwaną złożoność czasową algorytmu Hoare( ) szacuje się przez $O(n)$. W odróżnieniu od funkcji QuickSort( ), która rekurencyjnie wywołuje samą siebie dwukrotnie (tj. dla dwóch podciągów), funkcja Hoare( ) wywołuje samą siebie tylko raz (tzn. tylko dla jednego podciągu).
- Algorytm Hoare'a jest szybki dla danych losowych i podobnie jak QuickSort( ) ma złożoność pesymistyczną czasową $\Theta(n^2)$.

8. Zadania różne

[01] Dany jest wektor A liczb naturalnych uporządkowanych niemalejąco. W wektorze tym szukamy wartości $x = 14$. Podaj kolejne wartości tablicy A porównywane z liczbą x, jeśli zastosowano metodę binarną.

A[i]	2	4	4	5	6	8	10	11	16
i	0	1	2	3	4	5	6	7	8

Rozwiązanie: A[4] = 6, A[6] = 10, A[7] = 11, A[8] = 16, brak liczby $x = 14$ w wektorze.

[02] W algorytmie QuickSort( ) przyjęto za element rozdzielający najbardziej lewy element podwektora. Dla którego z poniższych ciągów QuickSort( ) wykona więcej porównań (i dlaczego)?

(a) 4 1 6 2 7 4
(b) 6 5 4 3 2 1

Rozwiązanie: QuickSort( ) wykona więcej porównań dla ciągu (b). Ciąg ten jest uporządkowany, więc za każdym razem będzie dzielony na dwa podciągi (długości 0 i $n_i - 1$).

[03] Przeanalizuj funkcję partition( ) dla ciągu liczb 6, 4, 8, 5, 9, 7, 2, 3, 1. Za element rozdzielający przyjmij lewy koniec tablicy.

Rozwiązanie: Uzyskany ciąg to: 2, 4, 1, 5, 3, 6, 7, 9, 8.

[04] Wyznacz złożoność algorytmu QuickSort( ) zastosowanego do takiego ciągu liczb, dla którego element rozdzielający za każdym razem trafi w środek sortowanego podciągu.

Rozwiązanie: Załóżmy, że funkcja partition( ) za każdym razem podzieli ciąg długości $n_i = right_i - left_i + 1$ na dwa podciągi o rozmiarach $\left\lfloor \frac{n_i-1}{2} \right\rfloor$ i $\left\lceil \frac{n_i-1}{2} \right\rceil$ elementów. Wówczas:

$$T(n) = \begin{cases} 0 & \text{dla } n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n + 1 & \text{dla } n > 1 \end{cases}$$

Załóżmy, że $n = 2^k$ dla $k \in N$.

$$T(n) = 2T\left(2^{k-1}\right) + 2^k + 1 = 2\left(2T\left(2^{k-2}\right) + 2^{k-1} + 1\right) + 2^k + 1 =$$

$$= 2^2 T\left(2^{k-2}\right) + 2^k + 2 + 2^k + 1 = \cdots =$$

$$= 2^k \underbrace{T\left(2^{k-k}\right)}_{0} + \underbrace{2^k + \cdots + 2^k}_{k \text{ razy}} + 2^{k-1} + 2^{k-2} + \cdots + 1 =$$

$$= k2^k + \frac{(1-2^k)}{1-2} = n\lg n + n - 1 = \Theta(n\lg n)$$

[05] Wyznacz pesymistyczną złożoność czasową poniższego algorytmu (operacja elementarna – porównania elementu x z końcami przedziałów).

```
int Zad1(double x, int left, int right) {
 if(left <= right) {
  int mid = (left + right) / 2;
  if((r[mid].x <= x) && (x <= r[mid].y)) return mid;
  else
   if(r[mid].x < x) return Zad1(x, mid + 1, right);
   else return Zad1(x, left, mid – 1);
 }
 return –1;
}
```

Dane:

- wektor r[ ] przedziałów domkniętych, wzajemnie rozłącznych i uporządkowanych względem początków;
- r[i].x – początek przedziału, r[i].y – koniec przedziału ($i = 0, \ldots, (n-1)$);
- x – liczba rzeczywista.

Wyjście:

- Taki indeks i ($i = 0, \dots, (n-1)$), dla którego spełniony jest warunek $x \in$ <r[i].x, r[i].y> lub wartość –1 w przypadku, gdy powyższy warunek nie jest spełniony dla żadnego i.

Rozwiązanie:

$$T(n) = \begin{cases} 3 & \text{dla } n = 1 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 3 & \text{dla } n > 1 \end{cases}$$

Załóżmy, że $n = 2^k$ dla $k \in N$.

$$T(n) = T\left(2^{k-1}\right) + 3 = T\left(2^{k-2}\right) + 3 + 3 = T\left(2^{k-3}\right) + 3 + 3 + 3 = \dots =$$

$$= T\left(2^{k-k}\right) + 3k = T(1) + 3k = 3 + 3k = 3 + 3\lg n = \Theta(\lg n).$$

[06] Wyznacz pesymistyczną złożoność czasową poniższego algorytmu oraz podaj rząd wielkości wyznaczonej funkcji złożoności. Operacja elementarna – porównania elementu x z elementami tablicy A, rozmiar zadania – n (liczba elementów tablicy A).

```
int Zad2(int left, int right, float x, int *A) {
 if(left == right) {
  if(A[left] <= x) return left;
  return left – 1;
 }
 if(left + 1 == right) {
  if(A[left] > x) return left – 1;
  return Zad2(right, right, x, A);
 }
 int m = (left + right) / 2;
 if(A[m] > x) return Zad2(left, m, x, A);
 return Zad2(m, right, x, A);
}
```

Dane:

- rosnący ciąg liczb naturalnych $(a_0, \dots, a_{n-1})$ $(n \in N)$,
- liczba rzeczywista x.

Wyjście:

$$\text{out} = \begin{cases} -1 & \text{gdy } x < a_0 \\ i & \text{gdy } a_i \le x < a_{i+1} \\ n-1 & \text{gdy } a_{n-1} \le x \end{cases}$$

Rozwiązanie:

$$T(n) = \begin{cases} 1 & \text{dla } n = 1 \\ 2 & \text{dla } n = 2 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 & \text{dla } n > 2 \end{cases}$$

Załóżmy, że $n = 2^k$ dla $k \in N$.

$$T(n) = T\left(2^{k-1}\right) + 1 = T\left(2^{k-2}\right) + 1 + 1 = T\left(2^{k-3}\right) + 1 + 1 + 1 = \cdots =$$

$$= T\left(2^{k-(k-1)}\right) + k - 1 = T(2) + k - 1 = k + 1 = 1 + \lg n = \Theta(\lg n).$$

[07] Podaj możliwie najefektywniejszy czasowo algorytm (tj. o złożoności $O(n \lg n)$) wyznaczający przecięcie zbiorów *A* i *B* ($A \cap B$). Zakładamy, że zbiory *A* i *B* zawierają wartości całkowite z zakresu od 1 do 100000 i reprezentowane są odpowiednio tablicami A i B oraz że elementy zbioru *B* (tablica B) są uporządkowane rosnąco.

Wskazówka: Zastosuj algorytm wyszukiwania binarnego.

Rozdział 4. Technika projektowania algorytmów: programowanie dynamiczne

Programowanie dynamiczne to technika projektowania algorytmów mająca na celu zoptymalizowanie ich złożoności czasowej. Polega na tym, że *problem* P *zostaje podzielony na mniejsze problemy*. Następnie w pierwszej kolejności rozwiązywane są mniejsze podproblemy, a ich *wyniki są zapamiętywane w celu ponownego wykorzystania* do rozwiązania podproblemów wyższego rzędu.

1. Programowanie dynamiczne (por. [AHU])

Programowanie dynamiczne – technika projektowania algorytmów polegająca na tym, że:

- *problem zostaje podzielony na mniejsze problemy* (podobieństwo do „dziel i zwyciężaj");
- najpierw rozwiązywane są mniejsze podproblemy, a ich *wyniki są zapamiętywane w celu ponownego wykorzystania* do rozwiązania podproblemów wyższego rzędu, których wyniki również są zapamiętywane.

Programowanie dynamiczne ma sens, gdy np. rozwiązywany problem da się podzielić na wielomianową liczbę podproblemów i każdy z nich (poza przypadkami elementarnymi) można rozwiązać, korzystając z zapisanych w jakiejś strukturze (np. tablicy) wyników uzyskanych dla wcześniej rozwiązanych podproblemów. Taka możliwość pojawia się, gdy rozwiązywane podproblemy nie są niezależne, tzn. gdy podproblemy zawierają podproblemy tego samego typu i bez zastosowania programowania dynamicznego pewne podproblemy mogłyby być wykonywane wielokrotnie. Stąd programowanie dynamiczne jest techniką chętnie stosowaną przy zadaniach optymalizacyjnych.

2. Wyznaczanie *n*-tego wyrazu ciągu Fibonacciego

Definicja 1 (rekurencyjna definicja ciągu Fibonacciego)

$$F(n) = \begin{cases} n & \text{dla } n \leq 1 \\ F(n-2) + F(n-1) & \text{dla } n > 1 \end{cases}$$

Kilka pierwszych wyrazów ciągu: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

Algorytm rekurencyjny fib1( ):

```
int fib1(int n) {
 if(n <= 1) return n;
 return fib1(n – 2) + fib1(n – 1);
}
```

[Złożoność] Algorytm fib1( ) ma złożoność wykładniczą jeśli za operację elementarną przyjmiemy wywołanie funkcji rekurencyjnej.

- Dla $n \geq 2$ $T(n) > 2^{\frac{n}{2}}$,
  np. $T(2) = 3 > 2^{\frac{2}{2}}$, $T(3) = 5 > 2^{\frac{3}{2}}$.
- Aby wyznaczyć n-ty wyraz ciągu, trzeba określić więcej niż $2^{\frac{n}{2}}$ jego wyrazów.
- Określenie każdego wyrazu (oprócz fib1(0) i fib1(1)) kosztuje jedną operację dodawania.

Algorytm fib2( ) z zastosowaniem programowania dynamicznego:

```
int fib2(int n) {
 int *F = new int[n+1];
 F[0] = 0;
 if(n > 0) {
  F[1] = 1;
  for(int i = 2; i <= n; i++)
   F[i] = F[i–2] + F[i–1];
  return F[n];
 }
 delete [ ]F;
 return 0;
}
```

[Złożoność] Algorytm fib2( ) ma złożoność liniową.

- Aby wyznaczyć n-ty wyraz ciągu, należy określić $n - 1$ jego wyrazów.

```
fib1(5)
  ->fib1(3)
    ->fib1(1)=1
    ->fib1(2)
      ->fib1(0)=0
      ->fib1(1)=1
  ->fib1(4)
    ->fib1(2)
      ->fib1(0)=0
      ->fib1(1)=1
    ->fib1(3)
      ->fib1(1)=1
      ->fib1(2)
        ->fib1(0)=0
        ->fib1(1)=1
```

Rys. 4.1. Drzewo wywołań rekurencyjnych dla wywołania fib1(5)

- Aby wyznaczyć piąty wyraz ciągu Fibonacciego algorytm fib1( ) dwa razy wyznacza fib1(3). Algorytm fib2( ) zapamiętuje wyznaczone wartości w tablicy, co powoduje zoptymalizowanie złożoności z wykładniczej do liniowej.

3. Najdłuższy wspólny podciąg (por. [CLR])

Definicja 2 (podciąg ciągu znaków)
Dane są dwa ciągi $A = (a_1, \dots, a_m)$ i $B = (b_1, \dots, b_k)$.
B jest *podciągiem* A, jeśli istnieje taki rosnący ciąg indeksów $(i_1, \dots, i_m)$, że dla wszystkich $j = 1, \dots, k$ mamy $a_{i_j} = b_j$.

Przykład 1.

A = (a,	**b**,	r,	**a**,	k,	**a**,	d,	**a**,	**b**,	r,	**a**)
B = (	**b**,		**a**,		**a**,		**a**,	**b**,		**a**)

Problem 1. (Najdłuższy wspólny podciąg (NWP))
Dane są dwa ciągi: $X = (x_1, \dots, x_n)$ i $Y = (y_1, \dots, y_m)$. Należy znaleźć ich najdłuższy (tj. o możliwie największej liczbie znaków) wspólny podciąg.

Schemat rozwiązania (generowanie tablicy)
- Niech S będzie macierzą o rozmiarze $(n+1)\times(m+1)$, gdzie n i m są długościami odpowiednio ciągu X i Y.

$$\mathrm{S}[i][j] = \begin{cases} 0 & \text{dla } i = 0 \text{ lub } j = 0 \\ \mathrm{S}[i-1][j-1] + 1 & \text{dla } i,j > 0 \text{ i } x_i = y_j \\ \max\{\mathrm{S}[i][j-1], \mathrm{S}[i-1][j]\} & \text{dla } i,j > 0 \text{ i } x_i \neq y_j \end{cases}$$

```
int** TAB_NWP(string s1, string s2) {
 int r1 = s1.length( ), r2 = s2.length( );
 int **S = new int*[r1+1];
 for(int i = 0; i <= r1; i++) S[i] = new int[r2+1];

 for(int i = 0; i <= r1; i++) S[i][0] = 0;
 for(int i = 0; i <= r2; i++) S[0][i] = 0;
 for(int i = 1; i <= r1; i++)
  for(int j = 1; j <= r2; j++)
   if(s1[i–1] == s2[j–1]) S[i][j] = S[i–1][j–1] + 1;
   else S[i][j] = (S[i–1][j] >= S[i][j–1]) ? S[i–1][j] : S[i][j–1];
 return S;
}
```

Przykład 2. Utwórz tablicę S dla ciągów *A* = (banaraba) i *B* = (abrakadabra) oraz znajdź NWP(*A*, *B*).

Tabela 4.1. Ilustracja algorytmu NWP( ) dla ciągów *A* = (banaraba) i *B* = (abrakadabra)

		a	**b**	**r**	**a**	**k**	**a**	**d**	**a**	**b**	**r**	**a**
	0	0	0	0	0	0	0	0	0	0	0	0
b	0	0	**1**	1	1	1	1	1	1	1	1	1
a	0	1	1	1	**2**	2	2	2	2	2	2	2
n	0	1	1	1	2	2	2	2	2	2	2	2
a	0	1	1	1	2	2	**3**	3	3	3	3	3
r	0	1	1	2	2	2	3	3	3	3	4	4
a	0	1	1	2	3	3	3	3	**4**	4	4	5
b	0	1	2	2	3	3	3	3	4	**5**	5	5
a	0	1	2	2	3	3	4	4	4	5	5	**6**

- NWP(banaraba, abrakadabra) = baaaba
- Jak odczytać najdłuższy wspólny podciąg z tablicy? Kierując się strzałkami.

```
string NWP(string x, string y) {
 int **S = TAB_NWP(x, y);
 string nwp = "";
 int i = x.length( ), j = y.length( );
 do {
  if(x[i–1] == y[j–1]) { nwp += x[i–1]; i––; j––; }
  else
   if(S[i–1][j] >= S[i][j–1]) i––; else j––;
 } while(S[i][j]);
 delete [ ]S;
 return nwp;
}
```

[Złożoność] (algorytm NWP( ))
- utworzenie tablicy $O(nm)$,
- odnalezienie szukanego słowa w utworzonej tablicy $O(n+m)$.

4. Zero-jedynkowe zagadnienie plecakowe (por. [SDK])

Problem 2. Należy tak wypełnić plecak różnymi przedmiotami o wartościach p_i i wagach w_i $(i = 1, \dots, n)$, aby nie przekroczyć łącznego udźwigu W plecaka i jednocześnie wybrać takie przedmioty, by ich sumaryczna wartość (cena) była możliwie największa. To znaczy należy znaleźć $\max\{\sum_{i=1}^{n} p_i x_i\}$ przy warunkach $\sum_{i=1}^{n} w_i x_i \le W$, gdzie $x_i \in \{0, 1\}$, $p_i, x_i, W \in Z_+ \cup \{0\}$, $(i = 1, \dots, n)$. Algorytm powinien zwrócić wszystkie możliwe rozwiązania.

Schemat rozwiązania (generowanie tablicy S)
- Niech S będzie macierzą o rozmiarze $(W+1)\times(n+1)$, gdzie W jest maksymalnym udźwigiem, a n oznacza ilość elementów.

$$S[w][i] = \begin{cases} 0 & \text{dla } w = 0 \text{ lub } i = 0 \\ S[w][i-1] & \text{dla } w < w_i, i \neq 0 \\ \max\{S[w][i-1], S[w-w_i][i-1] + p_i\} & \text{dla } w \geq w_i, i \neq 0 \end{cases}$$

- $S[w][i]$ oznacza optymalną wartość (cenę) plecaka uzyskaną przy rozpatrzeniu i pierwszych elementów (przy ograniczeniu, że całkowita waga plecaka nie może przekroczyć wartości w).
- $S[W][n]$ oznacza maksymalną wartość (cenę) plecaka przy rozpatrzeniu n elementów (przy ograniczeniu, że całkowita waga nie może przekroczyć wartości W).
- Rozwiązanie uzyskujemy, używając dwóch 2-wymiarowych tablic (S i S2 indeksowanych wierszowo od 0 do W i kolumnowo od 0 do n).

```
void Plecak(int *p, int *w, int n, int W) {
int pom = 0;
int **S = new int*[W+1];
int **S2 = new int*[W+1];
for(int i = 0; i <= W; i++) { S[i] = new int[n+1]; S2[i] = new int[n+1]; }
for(int i = 0; i <= W; i++) { S[i][0] = S2[i][0] = 0; }
for(int i =0; i <= n; i++) { S[0][i] = S2[0][i] = 0; }

for(int i = 1; i <= W; i++)
 for(int j = 1; j <= n; j++)
  if(i – w[j] >= 0) {
   pom = S[i–w[j]][j–1] + p[j];
   if(pom < S[i][j–1]) { S[i][j] = S[i][j–1]; S2[i][j] = 0; }
```

```
        else { S[i][j] = pom; S2[i][j] = 1; }
      } else{ S[i][j] = S[i][j−1]; S2[i][j] = 0; }
    }
```

Przykład 3. Rozwiąż zero-jedynkowy problem plecakowy dla poniższych danych.
$n = 4$, $W = 6$ // n – liczba przedmiotów, W – maksymalna waga (udźwig) plecaka
$p_1 = 2$, $w_1 = 2$ // p_i – cena i-tego przedmiotu, w_i – waga i-tego przedmiotu
$p_2 = 4$, $w_2 = 3$
$p_3 = 2$, $w_3 = 1$
$p_4 = 2$, $w_4 = 2$

Tabela 4.2. Ilustracja algorytmu rozwiązującego 0-1 problem plecakowy

w	S[w][0]	S[w][1]	S[w][2]	S[w][3]	S[w][4]
0	0	0	0	0	0
1	0	0	0	**2**	2
2	0	**2**	2	**2**	**2**
3	0	**2**	**4**	**4**	**4**
4	0	**2**	**4**	**6**	6
5	0	**2**	**6**	**6**	**6**
6	0	**2**	**6**	**8**	**8**

w	S2[w][0]	S2[w][1]	S2[w][2]	S2[w][3]	S2[w][4]
0	0	0	0	0	0
1	0	0	0	**1**	0
2	0	**1**	0	0/**1**	0/**1**
3	0	**1**	**1**	0/**1**	0/**1**
4	0	**1**	**1**	**1**	0
5	0	**1**	**1**	0/**1**	0/**1**
6	0	**1**	**1**	**1**	0/**1**

- Powyższe tablice S i S2 uzyskamy, wpisując instrukcję Plecak(p, w, 4, 6);, gdzie tablice p[ ] i w[ ] są zdefiniowane następująco: p[ ] = {0, 2, 4, 2, 2} i w[ ] = {0, 2, 3, 1, 2}.
- Jak odczytać możliwości załadunku plecaka z tablicy? Kierując się strzałkami. Z powyższych tablic wynika, że możliwe są następujące rozwiązania {2, 3, 4} i {1, 2, 3}.
- S2[w][i] = 0 / **1** oznacza, że nie opłaca się/opłaca się załadować do plecaka przedmiot i-ty.

[Złożoność] (algorytm dla zero-jedynkowego problemu plecakowego)
- Przyjmijmy za operację elementarną wpis do tablicy.
- Złożoność algorytmu szacujemy przez $\Theta(nW)$.
- Między n i W nie istnieje żadna zależność.
- Dla danego n można wybrać dowolnie duże W, np. $W = 4^n$.
  Wówczas powyższe rozwiązanie będzie mniej optymalne od algorytmu naiwnego (sprawdzającego wszystkie możliwości), którego złożoność wynosi $\Theta(2^n)$.
- Podany algorytm ma złożoność wyższą niż *wielomianowa*. Nie jest znane rozwiązanie problemu o złożoności wielomianowej.
- Problem należy do *klasy problemów NP-zupełnych*.

5. Zadania różne

[01] Wyznacz najdłuższy wspólny podciąg dla słów:
(a) ”KROLKAROL” i ”KOLOROWEKORALE”,
(b) ”KORALEGOSI” i ”MISKORALGOL”.

Rozwiązanie:
(a) ”KROLKAROL” i ”KOLOROWEKORALE”

```
     K  O  L  O  R  O  W  E  K  O  R  A  L  E
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
K[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
R[0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2]
O[0, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
L[0, 1, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4]
K[0, 1, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4]
A[0, 1, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 5, 5, 5]
R[0, 1, 2, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5]
O[0, 1, 2, 3, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5]
L[0, 1, 2, 3, 4, 4, 5, 5, 5, 5, 5, 5, 5, 6, 6]
NWP(”KROLKAROL”,”KOLOROWEKORALE”) = KROKAL
```

(a) ”KORALEGOSI” i ”MISKORALGOL”

```
     M  I  S  K  O  R  A  L  G  O  L
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
K[0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
O[0, 0, 0, 0, 1, 2, 2, 2, 2, 2, 2, 2]
R[0, 0, 0, 0, 1, 2, 3, 3, 3, 3, 3, 3]
A[0, 0, 0, 0, 1, 2, 3, 4, 4, 4, 4, 4]
L[0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5]
E[0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5]
G[0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 6, 6]
O[0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 7]
S[0, 0, 0, 1, 1, 2, 3, 4, 5, 6, 7, 7]
I[0, 0, 1, 1, 1, 2, 3, 4, 5, 6, 7, 7]
NWP(”KORALEGOSI”,”MISKORALGOL”) = KORALGO
```

[02] Rozwiąż zero-jedynkowe zagadnienie plecakowe dla poniższych danych.
(a) $n = 4, W = 8, p1 = 1, w1 = 2, p2 = 4, w2 = 3, p3 = 11, w3 = 4, p4 = 2, w4 = 1,$
(b) $n = 4, W = 6, p1 = 2, w1 = 2, p2 = 4, w2 = 3, p3 = 2, w3 = 1, p4 = 2, w4 = 2,$

Rozwiązanie:
(a) $n = 4, W = 8, p1 = 1, w1 = 2, p2 = 4, w2 = 3, p3 = 11, w3 = 4, p4 = 2, w4 = 1$

```
S1   1  2   3   4 (przedmioty)     S2   1  2  3  4 (przedmioty)
0[0, 0, 0,  0,  0]                0[0, 0, 0, 0, 0]
1[0, 0, 0,  0,  2]                1[0, 0, 0, 0, 1]
2[0, 1, 1,  1,  2]                2[0, 1, 0, 0, 1]
3[0, 1, 4,  4,  4]                3[0, 1, 1, 0, 0]
4[0, 1, 4, 11, 11]                4[0, 1, 1, 1, 0]
5[0, 1, 5, 11, 13]                5[0, 1, 1, 1, 1]
6[0, 1, 5, 12, 13]                6[0, 1, 1, 1, 1]
7[0, 1, 5, 15, 15]                7[0, 1, 1, 1, 0]
8[0, 1, 5, 15, 17]                8[0, 1, 1, 1, 1]
```

Jedna możliwość {4, 3, 2}

(b) $n = 4, W = 6, p1 = 2, w1 = 2, p2 = 4, w2 = 3, p3 = 2, w3 = 1, p4 = 2, w4 = 2$

```
S1   1  2  3  4 (przedmioty)      S2   1  2  3  4 (przedmioty)
0[0, 0, 0, 0, 0]                 0[0, 0, 0, 0, 0]
1[0, 0, 0, 2, 2]                 1[0, 0, 0, 1, 0]
2[0, 2, 2, 2, 2]                 2[0, 1, 0, 1, 1]
3[0, 2, 4, 4, 4]                 3[0, 1, 1, 1, 1]
4[0, 2, 4, 6, 6]                 4[0, 1, 1, 1, 0]
5[0, 2, 6, 6, 6]                 5[0, 1, 1, 1, 1]
6[0, 2, 6, 8, 8]                 6[0, 1, 1, 1, 1]
```

Dwie możliwości {4, 3, 2} i {3, 2, 1}

[03] Które z następujących problemów opłaca się zaimplementować z użyciem programowania dynamicznego?
(a) wyznaczenie n-tego elementu ciągu Fibonacciego,
(b) wyznaczenie symbolu Newtona,
(c) rozwiązanie zero-jedynkowego problemu plecakowego,
(d) wyznaczenie najdłuższego wspólnego podciągu,
(e) rozwiązanie problemu wież z Hanoi,
(f) wyszukanie najmniejszej wartości w tablicy liczb.

Rozwiązanie:
(a) [+], (b) [+], (c) [+], (d) [+], (e) [–], (f) [–]

Rozdział 5. Technika projektowania algorytmów: metoda zachłanna

1. Algorytm zachłanny
2. Problem wydawania reszty o minimalnej liczbie monet
3. Problem znajdowania najkrótszych ścieżek z ustalonego źródła w grafie *G* (algorytm Dijkstry)
4. Problem wyznaczania minimalnego drzewa rozpinającego (algorytm Kruskala)
5. Kompresja Huffmana
6. Zadania różne

Algorytmy zachłanne są stosowane m.in. do rozwiązywania problemów optymalizacyjnych. *Metoda zachłanna* polega na konsekwentnym (tj. w każdym kroku) podejmowaniu takich decyzji, które są w chwili wyboru najbardziej korzystne, w nadziei uzyskania optymalnego rozwiązania globalnego. Algorytmy zachłanne nie zawsze generują optymalne wyniki. W przypadku każdego konkretnego problemu należy ustalić, czy jego rozwiązanie metodą zachłanną jest optymalne w każdym przypadku.

1. Algorytm zachłanny (*Greedy algorithm*) (por. [AHU], [CLR])

- *Algorytm zachłanny* polega na podjęciu ciągu takich decyzji (po jednej decyzji w każdym kroku), które są w chwili wyboru najbardziej korzystne, tzn. w każdym kroku wybieramy możliwość lokalnie optymalną w nadziei, że rozwiązanie globalne będzie również optymalne.
- Algorytm zachłanny nie zawsze generuje optymalne wyniki. W przypadku każdego konkretnego problemu należy ustalić, czy jego rozwiązanie metodą zachłanną jest rozwiązaniem dokładnym w każdym przypadku. Nie istnieje ogólna metoda rozstrzygająca, czy w przypadku zadanego problemu algorytm zachłanny zwróci poprawny rezultat.
- Algorytmy zachłanne są chętnie wykorzystywane m.in. do rozwiązywania problemów optymalizacyjnych. Dla praktycznego problemu P, jeśli nawet w pewnych przypadkach nie uzyskamy metodą zachłanną rozwiązania optymalnego,

to wyniki bliskie optymalnym są często wystarczające (tzn. praktycznie akceptowalne). Takie zachłanne podejście jest opłacalne, gdy np. jedynym algorytmem pozwalającym na uzyskanie wyników dokładnych jest przeszukiwanie wyczerpujące (tj. sprawdzenie wszystkich możliwości).

- Problemy, dla których istnieją optymalne algorytmy zachłanne to (por. [CLR], [DS], [LW]):
  - problem znajdowania najkrótszych ścieżek z ustalonego wierzchołka w grafie (algorytm Dijkstry),
  - problem wyznaczania minimalnego drzewa rozpinającego (algorytm Kruskala),
  - problem kompresji danych (metoda Huffmana).

2. Problem wydawania reszty o minimalnej liczbie monet (por. [AHU])

Problem 1. (Wydawanie reszty przy użyciu minimalnej liczby monet)
Dane są następujące monety:
(a) 20gr, 10gr, 10gr, 5gr, 1gr, 1gr
(b) 11gr, 11gr, 10gr, 5gr, 2gr, 1gr, 1gr, 1gr

Użyj możliwie najmniejszej liczby monet w celu wydania reszty równej: (a) 31gr, (b) 26gr.

Rozwiązanie zachłanne:
Wydajemy możliwie największe (mieszczące się w reszcie) monety, aż do zwrócenia dokładnej kwoty (gdy rozwiązanie istnieje) lub do momentu, gdy skończą się pieniądze (gdy brak rozwiązania).

Wynik algorytmu:
(a) 20gr, 10gr, 1gr, 3 monety (rozwiązanie optymalne)
(b) 11gr, 11gr, 2gr, 1gr, 1gr, 5 monet (rozwiązaniem optymalnym są 3 monety)

W systemie monetarnym (a) algorytm zachłanny zawsze zwróci optymalne rozwiązanie. W przypadku systemu monetarnego (b) nie należy stosować metody zachłannej do rozwiązania problemu wydawania reszty przy użyciu minimalnej liczby monet.

3. Problem znajdowania najkrótszych ścieżek z ustalonego wierzchołka w grafie *G* (algorytm Dijkstry) (por. [CLR], [LW], [SDK])

Przykład 1. Harcerze uczestniczący w biegu przełajowym otrzymali odręcznie sporządzoną mapę, na której zaznaczono krzyżykami rozgałęzienia leśnych ścieżek oraz odległości (mierzone wzdłuż zaznaczonych ścieżek) między każdą parą sąsiadujących (oznaczonych krzyżykami) „skrzyżowań". Zadaniem harcerzy jest przemaszerowanie (najkrótszą trasą) z punktu startowego (we wschodnim fragmencie puszczy) do punktu kontrolnego usytuowanego w jednym z takich rozgałęzień (w zachodniej części puszczy). Harcerzom wolno poruszać się jedynie oznakowanymi drogami zaznaczonymi na mapie. Ponieważ duży obszar lasu pokrywają bagna, pomysł, by konsekwentnie kierować się ze wschodu na zachód, nie jest możliwy do zrealizowania.

Jak efektywnie rozwiązać powyższy problem?

Definicje i fakty (por. [CLR], [LW], [WR]):

- *graf skierowany G* – dowolna para uporządkowana $G = (V, E)$ (V – zbiór wierzchołków, E – zbiór krawędzi), taka że $E \subseteq \{(u, v)\colon u, v \in V\}$ ((u, v) oznacza parę uporządkowaną wierzchołków $u, v \in V$);
- *graf prosty* – graf niezawierający pętli (tj. krawędzi (u, u), gdzie $u \in V$) oraz krawędzi wielokrotnych;
- *droga* w grafie G – taki ciąg wierzchołków $(v_0, v_1, \ldots, v_k)$ $(k \geq 0)$, że $(v_i, v_{i+1}) \in E$ (lub $\{v_i, v_{i+1}\} \in E$ dla grafu niezorientowanego) jest krawędzią $(i = 0, 1, \ldots, k-1)$ oraz $(v_j \neq v_ł)$ (z wyjątkiem być może równości $v_0 = v_k$) (dla $j, ł = 0, 1, \ldots, k$);
- *waga ścieżki* $p = (v_0, v_1, \ldots, v_k)$ – suma wag krawędzi tworzących daną ścieżkę $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$;
- *odległość* (*wagę najkrótszej ścieżki*) z wierzchołka u do wierzchołka v (ozn. $\delta(u, v)$) definiujemy jako:

$$\delta(u, v) = \begin{cases} \min\left\{w(p)\colon u \overset{p}{\rightarrow} v\right\} & \text{jeśli istnieje ścieżka z } u \text{ do } v \\ \infty & \text{w przeciwnym przypadku} \end{cases}$$

- *najkrótsza ścieżka* z wierzchołka u do wierzchołka v – każda ścieżka p z u do v, dla której $w(p) = \delta(u, v)$;
- *metoda relaksacji* – wielokrotne zmniejszanie górnego ograniczenia na rzeczywiste wagi najkrótszych ścieżek wykonywane do momentu, gdy te ograniczenia staną się rzeczywistymi wagami najkrótszych ścieżek.

Problem 2. (Najkrótsze ścieżki z jednym źródłem)
Dany jest graf skierowany $G = (V, E)$, ważony (tj. z funkcją wagową $w: E \to R_+ \cup \{0\}$ przyporządkowującą krawędziom grafu wagi o wartościach ze zbioru $R_+ \cup \{0\}$) oraz ustalony wierzchołek $s \in V$ (źródło) grafu G. Należy znaleźć najkrótsze ścieżki od ustalonego wierzchołka s do pozostałych wierzchołków grafu.

Idea (algorytm Dijkstry) (por. [CLR], [WL])
- Służy do rozwiązywania problemu najkrótszych ścieżek z jednym źródłem w ważonym grafie skierowanym $G = (V, E)$ w przypadku, gdy wagi wszystkich krawędzi są nieujemne.
- W algorytmie Dijkstry pamiętany jest zbiór T zawierający te wierzchołki, dla których wagi najkrótszych ścieżek ze źródła s zostały już obliczone (i już nie ulegną zmianie), tzn. dla każdego wierzchołka $v \in T$ mamy wyznaczoną wartość dist[v] = $\delta(s, v)$.
- Algorytm Dijkstry polega na (n–1)-krotnym powtórzeniu następujących operacji:
  - wybraniu nowego wierzchołka *last* $\in$ V$\setminus T$ o najmniejszym aktualnym oszacowaniu najkrótszej ścieżki,
  - dodaniu wierzchołka *last* do zbioru T,
  - relaksacji krawędzi przechodzących przez wierzchołek *last*.

Schemat (algorytm Dijkstry) (por. [LW])
Dane:
- graf zorientowany $G = (V, E)$, ważony (tj. z funkcją wagową $w: E \to R_+ \cup \{0\}$ przyporządkowującą krawędziom grafu wagi o wartościach ze zbioru $R_+ \cup \{0\}$) z wyróżnionym źródłem $s \in V$, reprezentowany tablicą list incydencji LI[ ].

Wynik:
- dist[n] – tablica zawierająca odległości od źródła s do wszystkich wierzchołków $v \in V$ grafu (dist[v] = $\delta(s, v)$ dla $v \in V$);
- pred[n] – tablica poprzedników (na najkrótszych drogach), za pomocą której można wyznaczyć najkrótsze drogi.

```
(1) T = ∅;
(2) for(v ∈ V) { dist[v] = ∞; pred[v] = –1; }
(3) for(v ∈ LI[s]) { dist[v] = LI[s]–>[v](w); pred[v] = s; }
(4) dist[s] = 0; pred[s] = s; T = T ∪ {s};
(5) while(V \ T ≠ ∅) {
(6)   last = dowolny wierzchołek v ∈ V \ T taki, że dist[v] == min{ dist[u]: u ∈ V \ T };
(7)   T = T ∪ {last};
(8)   for(v ∈ LI[last])
(9)     if(T ∩ {v} = ∅ && dist[v] > dist[last] + LI[last]–>[v](w)) {
```

```
(10)     dist[v] = dist[last] + LI[last]–>[v](w);
(11)     pred[v] = last;
(12)   }
(13)}
```

[Złożoność czasowa] (algorytm Dijkstry)

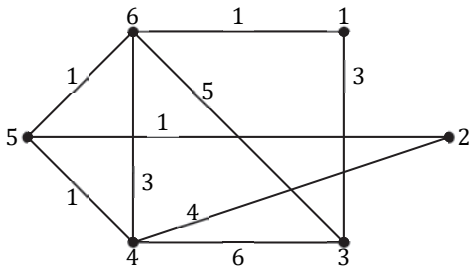
- Pętla (5) jest wykonywana $n-1$ razy, przy czym zarówno instrukcje w wierszu (6), jak i pętla (8) wymagają po $O(n)$ kroków. Złożoność algorytmu szacujemy więc przez $O(n^2)$.
- Powyższy algorytm można zmodyfikować tak, by jego złożoność wynosiła $O(m\lg n)$.

Idea (zmodyfikowany algorytm Dijkstry) (por. [LW])

- Do reprezentowania zbioru $V\backslash T$ należy wykorzystać strukturę kopca (patrz rozdział 5.2) z minimalną wartością klucza w korzeniu. Niech kluczem węzłów kopca będzie aktualna waga ścieżki od źródła s do wierzchołka v. Budowa kopca z n wierzchołków kosztuje $O(n)$. Usunięcie wierzchołka ze zbioru $V\backslash T$ to usunięcie korzenia kopca (kosztuje $O(\lg n)$).
- Aby przeprowadzić relaksację potrzebny jest dostęp do aktualnych wag ścieżek (od źródła s do pozostałych wierzchołków) zapamiętanych jako klucze wierzchołków kopca. Dostęp ten można uzyskać, definiując tablicę wskaźników do węzłów kopca.

W pętli (8) poprawiane są wagi ścieżek dla wierzchołków znajdujących się na liście LI[*last*]. Jeśli waga ścieżki z s do v ulegnie zmniejszeniu, należy poprawić wagę w odpowiednim węźle kopca, a następnie poprawić własność kopca, tzn. przesunąć węzeł ze zmodyfikowaną wagą ścieżki (od s do v) w górę, zamieniając ten węzeł z węzłami znajdującymi się w kopcu bezpośrednio nad nim. Ilość kroków takiego przesunięcia szacujemy przez $O(\lg n)$. Ponieważ w zmodyfikowanym algorytmie każda krawędź grafu jest analizowana dokładnie raz, to takich przesunięć węzłów w górę będzie co najwyżej m. Daje to w sumie $O(m\lg n)$ kroków. Do tego należy doliczyć $O(n)$ kroków potrzebnych do zbudowania kopca i $O(n\lg n)$ kroków potrzebnych do $(n-1)$-krotnego usunięcia z kopca korzenia. Całkowita złożoność zmodyfikowanego algorytmu to $O(m\lg n)$.

Przykład 2. Dla zadanego grafu $G = (V, E)$ i źródła $s = 1$ wyznacz najkrótsze ścieżki (i ich wagi) prowadzące od źródła s do wszystkich pozostałych wierzchołków. Zastosuj algorytm Dijkstry. Graf reprezentowany jest tablicą list incydencji LI[ ].



[1] 3(3), 6(1)
[2] 4(4), 5(1)
[3] 1(3), 4(6), 6(5)
[4] 2(4), 3(6), 5(1), 6(3)
[5] 2(1), 4(1), 6(1)
[6] 1(1), 3(5), 4(3), 5(1)

Rys. 5.1. Graf *G* i tablica list incydencji LI[ ] grafu *G*

Tabela 5.1. Ilustracja algorytmu Dijkstry

i\dist[ ]	1	2	3	4	5	6
1	0	∝	3	∝	∝	**1**
2	|	∝	3	1+3	**1+1**	|
3	|	**2+1**	3	2+1	|	|
4	|	|	**3**	3	|	|
5	|	|	|	**3**	|	|
dist[ ]	0	3	3	3	2	1
pred[ ]	1	5	1	6, 5	6	1

Ścieżki ustalone na podstawie tablicy pred[ ]:

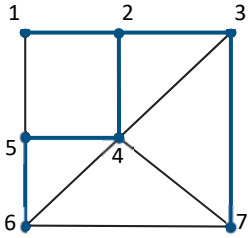
1
2 ← 5 ← 6 ← 1
3 ← 1
4 ← 5 ← 6 ← 1
5 ← 6 ← 1
6 ← 1

- Dijkstry jest algorytmem zachłannym, ponieważ w każdym kroku do zbioru T jest dodawany najbliższy wierzchołek ze zbioru V\T.
- Dijkstry zawsze zwraca optymalny wynik.

4. Problem wyznaczania minimalnego drzewa rozpinającego (algorytm Kruskala) (por. [AHU])

Definicje i fakty (por. [AHU], [CLR], [LW]):

- *las* – acykliczny graf niezorientowany (nieskierowany);
- *drzewo* – las spójny (acykliczny, spójny graf nieskierowany);
- *drzewo rozpinające* spójnego grafu $G = (V, E)$ – drzewo $G^* = (V, F)$, będące takim podgrafem grafu G ($F \subseteq E$), którego krawędzie łączą wszystkie wierzchołki grafu, przy czym $|F| = n - 1$ dla $|V| = n$; przykładowe drzewo rozpinające zaznaczono na poniższym grafie pogrubioną, niebieską linią;



Rys. 5.2. Drzewo rozpinające grafu G

- *minimalne drzewo rozpinające* grafu G – drzewo rozpinające $G^* = (V, F)$ grafu G, dla którego suma wag krawędzi $\sum_{e \in F} w(e)$ jest najmniejsza z możliwych; dla niektórych grafów można wskazać wiele drzew rozpinających spełniających tę własność;
- *podział zbioru* n-elementowego X na k bloków – dowolna rodzina $B = (B_1, \ldots, B_k)$ taka, że $B_1 \cup \ldots \cup B_k = X, B_i \cap B_j = \emptyset$ (dla $1 \le i < j \le k$), $B_i \neq \emptyset$ (dla $1 \le i \le k$).

Przykład 3. Las składający się z trzech składowych będących drzewami.

Rys. 5.3. Las składający się z trzech składowych

Problem 3. Niech $G = (V, E)$ (V – zbiór wierzchołków, $|V| = n$, E – zbiór krawędzi, $|E| = m$) będzie grafem nieskierowanym, spójnym, ważonym i niech $w: E \to R_+ \cup \{0\}$ oznacza funkcję kosztu określoną na krawędziach tego grafu. Dla zadanego grafu G oraz danej funkcji kosztu w należy znaleźć minimalne drzewo rozpinające.

Idea (algorytm Kruskala) (por. [MG])
Zachłanny algorytm dla problemu minimalnego drzewa rozpinającego.

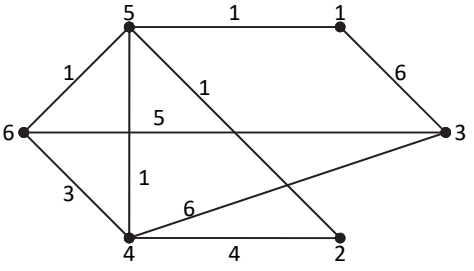
(1) Posortuj krawędzie grafu G ze względu na wartości wag w porządku niemalejącym. Zapamiętaj je w tablicy E.
(2) Utwórz początkowy podział P zbioru V jako rodzinę jednoelementowych zbiorów $\{v_i\}$ (gdzie $v_i \in V$, $i = 1, \ldots, n$), reprezentującą las początkowy $(G_i = (V_i, E_i), V_i = \{v_i\}, E_i = \emptyset, i = 1, \ldots, n)$.
(3) $F = \emptyset$ – zainicjalizuj zbiór krawędzi minimalnego drzewa rozpinającego.
(4) Wybierz spośród nieanalizowanych krawędzi tablicy E krawędź o najmniejszej wadze.
(5) Jeśli wybrana krawędź łączy dwa wierzchołki z różnych zbiorów podziału:
    (5a) dodaj ją do zbioru krawędzi tworzonego minimalnego drzewa rozpinającego F,
    (5b) połącz podzbiory zawierające wierzchołki dodanej krawędzi.
(6) Powtarzaj kroki (4), (5) dopóki zbiór F nie będzie zawierał $n - 1$ krawędzi.

- Algorytm Kruskala to algorytm zachłanny, ponieważ w każdym kroku do lasu dodawana jest krawędź o najmniejszej możliwej wadze.
- Algorytm Kruskala zawsze zwraca optymalny wynik.

[Złożoność czasowa] (algorytm Kruskala)

- Czas działania algorytmu Kruskala zależy od sposobu implementacji struktury danych dla zbiorów rozłącznych.
- Najszybsza znana implementacja (tj. z użyciem lasu zbiorów rozłącznych z łączeniem według rangi i z kompresją ścieżek (por. [CLR])) gwarantuje pesymistyczny czas wyznaczenia minimalnego drzewa rozpinającego tą metodą równy $\Theta(m \lg m)$, gdzie $m = |E|$.

Przykład 4. Wypisz krawędzie i sumę wag krawędzi minimalnego drzewa rozpinającego (MDK) wyznaczonego algorytmem Kruskala dla zadanego grafu G reprezentowanego tablicą list incydencji LI[ ].



[1] 3(6), 5(1)
[2] 4(4), 5(1)
[3] 1(6), 4(6), 6(5)
[4] 2(4), 3(6), 5(1), 6(3)
[5] 1(1), 2(1), 4(1), 6(1)
[6] 3(5), 4(3), 5(1)

Rys. 5.4. Graf G i tablica list incydencji LI[ ] grafu G

Rozwiązanie:

Krawędzie posortowane niemalejąco ze względu na wagi:	Proces generowania minimalnego drzewa rozpinającego dla grafu *G*:
w = 1: (1, 5), (2, 5), (4, 5), (5, 6),	{1}, {2}, {3}, {4}, {5}, {6}
w = 3: (4, 6),	{1, 5}, {2}, {3}, {4}, {6}
w = 4: (2, 4),	{1, 2, 5}, {3}, {4}, {6}
w = 5: (3, 6),	{1, 2, 4, 5}, {3}, {6}
w = 6: (1, 3), (3, 4)	{1, 2, 4, 5, 6}, {3}
	{1, 2, 3, 4, 5, 6}

Krawędzie MDK: (1, 5), (2, 5), (4, 5), (5, 6), (3, 6)
Suma wag krawędzi: ∑=1+1+1+1+5=9

5. Kompresja Huffmana

Definicje i fakty (por. [AHU], [DA], [SD]):

Definicja 1 (entropia)
Niech *X* będzie zmienną losową przyjmującą wartości ze skończonego zbioru zgodnie z rozkładem prawdopodobieństwa pr(*X*). *Entropię* tego rozkładu definiujemy wzorem:

$$H(X) = -\sum_{i=1}^{n} \mathrm{pr}_i \cdot \log_2 \mathrm{pr}_i$$

Jeśli x_i (dla $1 \le i \le n$) są wszystkimi możliwymi wartościami zmiennej *X*, to

$$H(X) = -\sum_{i=1}^{n} \mathrm{pr}(X = x_i) \cdot \log_2 \mathrm{pr}(X = x_i)$$

Definicja 2 (kodowanie f zmiennej *X*)
Niech *X* oznacza zmienną losową przyjmującą wartości ze skończonego zbioru zgodnie z rozkładem prawdopodobieństwa pr(*X*). *Kodowaniem* zmiennej losowej *X* nazywamy dowolne przekształcenie $f: X \to \{0, 1\}^*$, gdzie $\{0, 1\}^*$ jest zbiorem wszystkich skończonych ciągów 0–1.

Własności (*kodowania* f):

- Funkcja f powinna być *przekształceniem różnowartościowym*, ponieważ tylko w takim przypadku zaszyfrowany (zakodowany) tekst da się odszyfrować (zdekodować).
- Funkcja f musi być *funkcją prefiksową* (inaczej – *funkcją wolną od przedrostków*), tzn. nie może zdarzyć się sytuacja, że dla dwóch elementów $a, b \in X$ i ciągu $x \in \{0, 1\}^*$ zachodzi $f(a) = f(b)||x$ (symbol || oznacza konkatenację ciągów). Takie sytuacje powodowałyby niejasności przy dekodowaniu.

Definicja 3 (ważona średnia długości kodowania elementu X (ozn. $\ell(f)$)
Jest to miara skuteczności kodowania f zdefiniowana następująco:

$$\ell(f) = \sum_{x \in X} \mathrm{pr}(x) \cdot |f(x)|,$$

gdzie $|y|$ ozn. długość ciągu y.

Dąży się do wyszukania takiego różnowartościowego kodowania f, wolnego od przedrostków, które minimalizuje wartość $\ell(f)$. Metoda tworzenia optymalnego kodu została opracowana przez D. Huffmana. Zachłanny algorytm kompresji i dekompresji Huffmana generuje kodowanie f, które

- jest różnowartościowe,
- jest prefiksowe,
- spełnia nierówności $H(X) \leq \ell(f) < H(X) + 1$ (entropia przybliża średnią długość kodowania Huffmana).

Idea algorytmu (kompresja Huffmana)
Zachłanny algorytm generowania kodowania f, które jest różnowartościowe i wolne od przedrostków (przykładowa realizacja).

(1) Wyznacz tablicę częstości wystąpień poszczególnych znaków w tekście i na jej podstawie określ rozkład prawdopodobieństwa na zbiorze X możliwych znaków.
(2) Dla każdego znaku utwórz jednowęzłowe drzewo (węzły drzewa powinny mieć następujące pola: klucz (prawdopodobieństwo pr wystąpienia znaku), znak (lub '_' w przypadku węzłów wewnętrznych), referencje do lewego i prawego poddrzewa). Listę L jednowęzłowych drzew uporządkuj niemalejąco ze względu na klucz.
(3) W następujący sposób utwórz drzewo Huffmana na podstawie listy L:
Podczas gdy lista L zawiera więcej niż jedno drzewo, wykonuj:
Weź z początku listy L dwa drzewa T1 i T2 (o najmniejszych prawdopodobieństwach pr1 i pr2 w korzeniach) i połącz je w jedno drzewo tak, aby drzewa T1 i T2 były odpowiednio lewym i prawym poddrzewem korzenia, w którego kluczu zapiszesz wartość pr1+pr2. Nowe drzewo umieść na liście L w taki sposób, aby nadal była uporządkowana niemalejąco.
(4) Wygeneruj kody Huffmana w następujący sposób:
Każdą lewą krawędź drzewa oznacz znakiem '0', a prawą – '1'. Dla każdego znaku (znajdującego się w liściu drzewa) utwórz kod, spisując zera i jedynki ze ścieżki prowadzącej od korzenia drzewa do liścia odpowiadającego danemu znakowi. Znaki i ich kody zapisz w strukturze (której elementy mają dwa pola: znak w roli klucza oraz jego kod) z szybkim dostępem do elementu o zadanym kluczu.
(5) Zakoduj tekst T, korzystając z wygenerowanych kodów, a następnie każdy zero-jedynkowy 8-elementowy podciąg zamień na znak (zgodnie z tablicą kodów ASCII). Jeśli ostatni ciąg składa się z mniejszej niż 8 liczby znaków, dopełnij go zerami.

(6) Do pliku wyjściowego wklej tablicę częstości wystąpień poszczególnych znaków oraz tekst w zakodowanej postaci.

Idea algorytmu (dekompresja Huffmana)

(1) Odczytaj ze skompresowanego pliku tablicę częstości wystąpień znaków.
(2) Utwórz drzewo Huffmana dokładnie w taki sam sposób, jak podczas kompresji (podpunkty (2), (3)).
(3) Zamień zakodowany tekst na ciąg zero-jedynkowy (każdy znak tekstu to wartość z zakresu 0-255, którą należy zapisać w postaci binarnej).
(4) Określ (na podstawie tablicy częstości), ile znaków należy rozkodować. Czytaj kolejne zera i jedynki ciągu i na tej podstawie poruszaj się odpowiednio w lewo lub w prawo po drzewie Huffmana od jego korzenia do liścia, w którym znajduje się zakodowany znak. Po rozkodowaniu fragmentu ciągu zero-jedynkowego (tj. po przypisaniu mu odpowiedniego znaku znajdującego się w liściu drzewa) rozpocznij dekodowanie kolejnego znaku (startując na nowo od korzenia drzewa).

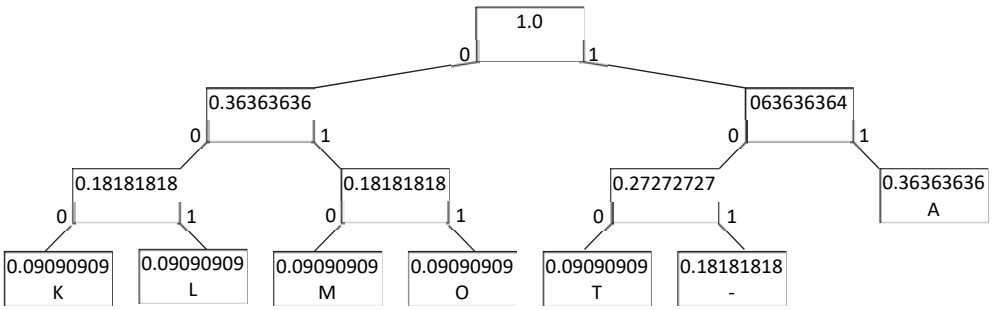
Przykład 5. Zastosuj opisany powyżej algorytm kompresji Huffmana dla tekstu *T*.
T = "ALA_MA_KOTA".

(1) tablica częstości wystąpień poszczególnych znaków w tekście

znak	A	L	_	M	K	O	T
częstość	4	1	2	1	1	1	1

(2) uporządkowana niemalejąco lista drzew jednowęzłowych
[K, 0.090909]→[L, 0.090909]→[M, 0.090909]→[O, 0.090909]→
[T, 0.090909]→[_, 0.181818]→[A, 0.363636]

(3) drzewo kodów Huffmana



Rys. 5.5. Drzewo kodów Huffmana

(4) kody Huffmana
{K = 000, L = 001, M = 010, O = 011, T = 100, _ = 101, A = 11}

(5) zakodowany tekst z podziałem na podciągi 8-znakowe
A L A _ M A _ K O T A
11001111 01010111 01000011 10011**000** – podkreślone zera dopełniają ostatnią ósemkę
(6) zawartość skompresowanego pliku (tablica częstości i zakodowany tekst)
A 4 L 1 _ 2 M 1 K 1 O 1 T 1 ±WCÿ

6. Zadania różne

[01] Dla zadanego grafu $G = (V, E)$ i źródła $s = 1$ wyznacz najkrótsze ścieżki (i ich wagi) prowadzące od źródła s do wszystkich pozostałych wierzchołków. Zastosuj algorytm Dijkstry. Graf reprezentowany jest tablicą list incydencji LI[ ].

(a) tablica list incydencji LI[ ]:
[1] 2(1), 4(1)
[2] 3(1), 5(3)
[3] 6(6)
[4] 3(4), 5(5)
[5] 6(1)
[6]

(b) tablica list incydencji LI[ ]:
[1] 2(1), 3(5), 4(1), 5(3)
[2] 3(1)
[3] 6(1)
[4] 5(1)
[5] 3(1), 6(3)
[6]

Tabela 5.2a. Ilustracja algorytmu Dijkstry

i\dist[ ]	1	2	3	4	5	6
1	0	**1**	∝	1	∝	∝
2	|	|	1+1	**1**	1+3	∝
3	|	|	**2**	|	4	∝
4	|	|	|	|	**4**	2+6
5	|	|	|	|	|	**4+1**
dist[ ]	0	1	2	1	4	5
pred[ ]	1	1	2	1	2	3, 5

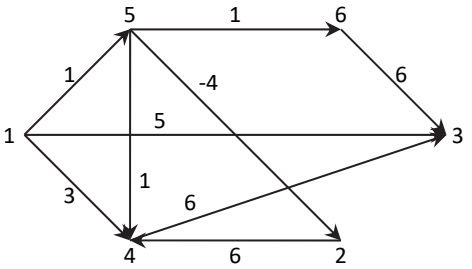
Ścieżki ustalone na podstawie tablicy pred[ ]:
1
2 ← 1
3 ← 2 ← 1
4 ← 1
5 ← 2 ← 1
6 ← 5 ← 2 ← 1

Tabela 5.2b. Ilustracja algorytmu Dijkstry

i\dist[ ]	1	2	3	4	5	6
1	0	**1**	5	1	3	∝
2	|	|	1+1	**1**	3	∝
3	|	|	**2**	|	1+1	∝
4	|	|	|	|	**2**	2+1
5	|	|	|	|	|	**3**
dist[ ]	0	1	2	1	2	3
pred[ ]	1	1	1, 2	1	1, 4	3

Ścieżki ustalone na podstawie tablicy pred[ ]:
1
2 ← 1
3 ← 2 ← 1
4 ← 1
5 ← 4 ← 1
6 ← 3 ← 2 ← 1

[02] Dlaczego algorytm Dijkstry błędnie określi w poniższym grafie najkrótsze ścieżki ze źródła $s = 1$?



[1] 3(5), 4(3), 5(1)
[2] 4(6)
[3]
[4] 3(6)
[5] 2(-4), 4(1), 6(1)
[6] 3(6)

Rys. 5.6. Graf G i tablica list incydencji LI[ ] grafu G

Tabela 5.3. Błędny wynik uzyskany algorytmem Dijkstry

i\dist[ ]	1	2	3	4	5	6
1	0	∝	5	3	**1**	∝
2	|	**1–4**	5	1+1	|	1+1
3	|	|	5	–3+6	|	**2**
4	|	|	5	**3**	|	|
5	|	|	5	|	|	|
dist[ ]	0	–3	5	3	1	2
pred[ ]	1	5	1	1, 5, 2	1	5

Ścieżki ustalone na podstawie tablicy pred[ ]

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

Poprawny wynik:

dist[ ]	0	–3	5	2	1	2
pred[ ]	1	5	1	5	1	5

Uzyskano błędny rezultat, ponieważ algorytmu Dijkstry nie stosuje się do grafów z ujemnymi wagami.

[03] Dla zadanego grafu $G = (V, E)$ i ujścia $t = 6$ wyznacz najkrótsze ścieżki (i ich wagi) prowadzące od każdego z wierzchołków do ujścia *t*. Zastosuj algorytm Dijkstry. Graf reprezentowany jest tablicą list incydencji LI[ ].

(a) tablica list incydencji LI[ ]:
[1] 2(1), 4(1)
[2] 3(1), 5(3)
[3] 6(6)
[4] 3(4), 5(5)
[5] 6(1)
[6]
transpozycja grafu:
[1]
[2] 1(1),
[3] 2(1), 4(4),
[4] 1(1),
[5] 2(3), 4(5),
[6] 3(6), 5(1),

(b) tablica list incydencji LI[ ]:
[1] 2(1), 3(5), 4(1), 5(3)
[2] 3(1)
[3] 6(1)
[4] 5(1)
[5] 3(1), 6(3)
[6]
transpozycja grafu:
[1]
[2] 1(1)
[3] 1(5), 2(1), 5(1)
[4] 1(1)
[5] 1(3), 4(1)
[6] 3(1), 5(3)

Tabela 5.4a. Ilustracja algorytmu Dijkstry

i\dist[ ]	1	2	3	4	5	6
1	∝	∝	6	∝	**1**	0
2	∝	**1+3**	6	1+5	|	|
3	**4+1**	|	6	6	|	|
4	|	|	**6**	6	|	|
5	|	|	|	6	|	|
dist[ ]	5	4	6	6	1	0
pred[ ]	2	5	6	5	6	6

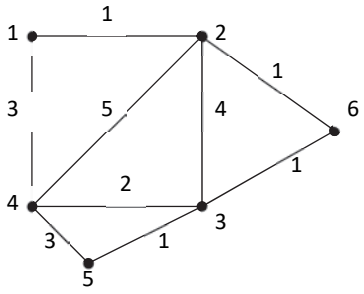
Ścieżki ustalone na podstawie tablicy pred[ ]:
$1 \to 2 \to 5 \to 6$
$2 \to 5 \to 6$
$3 \to 6$
$4 \to 5 \to 6$
$5 \to 6$
6

Tabela 5.4b. Ilustracja algorytmu Dijkstry

i\dist[ ]	1	2	3	4	5	6
1	∝	∝	**1**	∝	3	0
2	1+5	**1+1**	|	∝	1+1	|
3	2+1	|	|	∝	**2**	|
4	**3**	|	|	2+1	|	|
5	|	|	|	**3**	|	|
dist[ ]	3	2	1	3	2	0
pred[ ]	3, 2	3	6	5	6, 3	6

Ścieżki ustalone na podstawie tablicy pred[ ]:
$1 \to 2 \to 3 \to 6$
$2 \to 3 \to 6$
$3 \to 6$
$4 \to 5 \to 3 \to 6$
$5 \to 3 \to 6$
6

[04] Podaj krawędzie i sumę wag krawędzi minimalnego drzewa rozpinającego grafu *G* wyznaczonego algorytmem Kruskala.



[1] 2(1), 4(3)
[2] 1(1), 3(4), 4(5), 6(1)
[3] 2(4), 4(2), 5(1), 6(1)
[4] 1(3), 2(5), 3(2), 5(3)
[5] 3(1), 4(3)
[6] 2(1), 3(1)

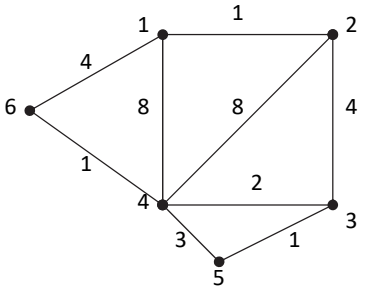
Rys. 5.7. Graf *G* i tablica list incydencji LI[ ] grafu *G*

Krawędzie posortowane niemalejąco ze względu na wagi:
w=1: (1, 2), (2, 6), (3, 5), (3, 6),
w=2: (3, 4),
w=3: (1, 4), (4, 5),
w=4: (2, 3),
w=5: (2, 4),

Proces generowania minimalnego drzewa rozpinającego dla grafu *G*:
{1}, {2}, {3}, {4}, {5}, {6}
{1, 2}, {3}, {4}, {5}, {6}
{1, 2, 6}, {3}, {4}, {5}
{1, 2, 6}, {3, 5}, {4}
{1, 2, 3, 5, 6}, {4}
{1, 2, 3, 4, 5, 6}

Krawędzie MDK: (1, 2), (2, 6), (3, 5), (3, 6), (3, 4)
Suma wag krawędzi: $\sum$=1+1+1+1+2=6

[05] Podaj krawędzie i sumę wag krawędzi minimalnego drzewa rozpinającego grafu *G* wyznaczonego algorytmem Kruskala.



[1] 2(1), 4(8), 6(4)
[2] 1(1), 3(4), 4(8)
[3] 2(4), 4(2), 5(1)
[4] 1(8), 2(8), 3(2), 5(3), 6(1)
[5] 3(1), 4(3)
[6] 1(4), 4(1)

Rys. 5.8. Graf *G* i tablica list incydencji LI[ ] grafu *G*

Krawędzie posortowane niemalejąco ze względu na wagi:	Proces generowania minimalnego drzewa rozpinającego dla grafu G:
w=1: (1, 2), (3, 5), (4, 6),	{1}, {2}, {3}, {4}, {5}, {6}
w=2: (3, 4),	{1, 2}, {3}, {4}, {5}, {6}
w=3: (4, 5),	{1, 2}, {3, 5}, {4}, {6}
w=4: (1, 6), (2, 3),	{1, 2}, {3, 5}, {4, 6}
w=8: (1, 4), (2, 4),	{1, 2}, {3, 4, 5, 6}
	{1, 2, 3, 4, 5, 6}

Krawędzie MDK: (1, 2), (3, 5), (4, 6), (3, 4), (1, 6)
Suma wag krawędzi: ∑=1+1+1+2+4=9

[06] Eksperyment polega na n-krotnym rzucie monetą. Niech $X = \{\mathrm{O}, \mathrm{R}\}$ (O – orzeł, R – reszka). Czy kodowania f, g są funkcjami prefiksowymi?
(a) $f(\mathrm{O}) = 01, f(\mathrm{R}) = 0110$
(b) $g(\mathrm{O}) = 01, g(\mathrm{R}) = 11$

Rozwiązanie:
(a) f nie jest funkcją prefiksową. W tym przypadku (np. podczas rozkodowania ciągu 01101...) nie ma jasności, jaki jest pierwszy znak (O czy R).
(b) g jest funkcją prefiksową. W tym przypadku zawsze mamy jednoznaczną odpowiedź, jaki jest kolejny znak (O czy R).

[07] X jest zmienną losową przyjmującą wartości ze zbioru $A = \{a, b, c, d\}$ z prawdopodobieństwami odpowiednio: $\mathrm{pr}(a) = 1/4$, $\mathrm{pr}(b) = 1/2$, $\mathrm{pr}(c) = 1/8$ i $\mathrm{pr}(d) = 1/8$. Policz entropię związaną z wylosowaniem jednej wartości ze zbioru A.

Rozwiązanie:

$$H(X) = -\left(\tfrac{1}{4}\log_2\tfrac{1}{4} + \tfrac{1}{2}\log_2\tfrac{1}{2} + \tfrac{1}{8}\log_2\tfrac{1}{8} + \tfrac{1}{8}\log_2\tfrac{1}{8}\right) =$$

$$= -\left(\tfrac{1}{4}(-2) + \tfrac{1}{2}(-1) + 2\cdot\tfrac{1}{8}(-3)\right) = \tfrac{7}{4}$$

Entropia wylosowania jednej wartości ze zbioru A wynosi $7/4$.

[08] Niech $A = \{a, b, c, d\}$. Które z następujących kodowań nie jest różnowartościowe?

(a)	$f(a) = 1$	$f(b) = 01$	$f(c) = 001$	$f(d) = 0001$
(b)	$g(a) = 110$	$g(b) = 111$	$g(c) = 10$	$g(d) = 0$
(c)	$h(a) = 0$	$h(b) = 00$	$h(c) = 01$	$h(d) = 10$

Rozwiązanie:
(a) Kodowanie f jest różnowartościowe. Ciąg zakodowany funkcją f można jednoznacznie rozkodować, ponieważ kod każdego symbolu kończy się znakiem 1.

(b) Kodowanie g jest różnowartościowe. Ciąg zakodowany funkcją g można jednoznacznie rozkodować, ponieważ przy czytaniu zakodowanego ciągu, np. 110100111…, zawsze mamy jednoznaczną odpowiedź, jaki kolejny symbol jest zakodowany. Następnie odcinamy odkodowany podciąg i czytamy dalej. Tu zakodowany ciąg symboli to *acdb*.
(c) Kodowanie h nie jest różnowartościowe, np. $h(ad) = 010 = h(ca)$.

[09] Podaj kody Huffmana wygenerowane dla tekstu *T*. Zakładamy, że podczas generowania drzewa, jeśli częstości dwóch znaków okażą się identyczne, jako pierwszy należy wybrać ten, który w alfabecie polskim ma mniejszy numer. Znak podkreślenia ma najwyższy numer. Dla ułatwienia zamiast prawdopodobieństwa umieść w kluczu ilość wystąpień znaku.
(a) *T* = "KROKODYLEK_KAROLA" (b) *T* = "KARTOTEKA"

Rozwiązanie:
(a) *T* = "KROKODYLEK_KAROLA"
(1) tablica ilości wystąpień poszczególnych znaków w tekście

znak	K	R	O	D	Y	L	E	_	A
ilość	4	2	3	1	1	2	1	1	2

(2) uporządkowana niemalejąco lista drzew jednowęzłowych

[D, 1]→[E, 1]→[Y, 1]→[_, 1]→[A, 2]→[L, 2]→[R, 2]→[O, 3]→[K, 4]

(3) drzewo Huffmana (wykonaj jako ćwiczenie)
(4) kody Huffmana

{K – 00, A – 010, L – 011, R – 100, D – 1010, E – 1011, Y – 1100, _ – 1101, O – 111}

(b) *T* = "KARTOTEKA"
(1) tablica ilości wystąpień poszczególnych znaków w tekście

znak	K	A	R	T	O	E
ilość	2	2	1	2	1	1

(2) uporządkowana niemalejąco lista drzew jednowęzłowych

[E, 1]→[O, 1]→[R, 1]→[A, 2]→[K, 2]→[T, 2]

(3) drzewo Huffmana (wykonaj jako ćwiczenie)
(4) kody Huffmana

{K – 00, T – 01, E – 100, O – 101, R – 110, A – 111}

[10] Na wejściu danych jest n ($1 \le n \le 100$, $n \in N$) różnych liczb naturalnych z przedziału $[1, k]$ ($1 \le k \le 150$). Zaproponuj algorytm o możliwie najniższej złożoności czasowej, który wykorzystując zadane liczby, generuje zbiory 2-elementowe $\{x, y\}$ spełniające własność $x + y \le k$. Każdą liczbę należy użyć dokładnie jeden raz. W przypadku, gdy dla zadanej wartości x nie można

już znaleźć takiej liczby y, by spełniona była powyższą własność, należy utworzyć zbiór 1-elementowy $\{x\}$. Zadanie polega na znalezieniu możliwie najmniejszej ilości tego typu zbiorów.

Przykład: Dla $k = 140$ i ciągu liczb: 60, 70, 80, 56, 67, 78, 81, 68 algorytm powinien zwrócić pięć zbiorów, np. {56, 81}, {60, 80}, {78}, {67, 70}, {68}.

Wskazówka: Uporządkuj liczby, wykorzystując (liniowe) sortowanie przez zliczanie, a następnie łącz liczby w pary, biorąc jedną z początku, a drugą z końca ciągu. Jeśli suma liczb jest większa niż k, utwórz zbiór 1-elementowy, zawierający wartość z końca ciągu.

Rozdział 6. Algorytmy tekstowe – wyszukiwanie wzorca w tekście

Podstawowym problemem dotyczącym tekstów jest wyszukiwanie wszystkich wystąpień ustalonego wzorca w tekście *T*. Efektywne *algorytmy wyszukiwania wzorca* są stosowane głównie w programach do edycji tekstu, ale również w systemach służących do znajdowania określonych wzorców w sekwencji DNA czy w systemach kryptograficznych.

1. Problem wyszukiwania wzorca (por. [BDR], [CLR])

Problem *wyszukiwania wzorca* w tekście (ozn. WW) polega na znalezieniu wszystkich wystąpień napisu *P*, zwanego wzorcem w tekście *T*. Będziemy zakładali, że $|P| = m$, $|T| = n$ oraz $m \leq n$.

Notacja, definicje i fakty:

- Niech $\sum$ oznacza skończony zbiór symboli zwany *alfabetem*. Ciągi znaków utworzone z symboli alfabetu $\sum$ nazywamy *tekstami* (lub *słowami*). *Długość słowa P* (ozn. $|P|$) to liczba znaków, z których *P* się składa. Słowo złożone z zerowej liczby znaków nazywamy *słowem pustym* i oznaczamy λ.
- Dla ułatwienia będziemy zakładać, że zarówno tekst *T*, jak i wzorzec *P* są ciągami znaków typu string (pozycje znaków w stringu są indeksowane od 0).
- Wystąpienia wzorca *P* w tekście *T* podaje się, wskazując pozycje s $(0 \leq s \leq n - m)$ w tekście, dla których: $T[s] = P[0]$ i $T[s+1] = P[1]$ i ... i $T[s+m-1] = P[m-1]$.
- Gdy $T[s..s+m-1] = P[0..m-1]$ dla $0 \leq s \leq n-m$, to mówimy, że *wzorzec P występuje z przesunięciem s* w tekście *T* albo że wzorzec *P* występuje w tekście *T* od pozycji $s+1$.

- Jeżeli P występuje w tekście T z przesunięciem s, to s nazywamy *poprawnym przesunięciem*, w przeciwnym razie s nazywamy *niepoprawnym przesunięciem*.
- $\sum^*$ – zbiór wszystkich tekstów (słów) utworzonych z symboli alfabetu $\sum$.
- $w \subset x$ – w jest *prefiksem* słowa x, gdy $x = wy$ dla pewnego słowa $y \in \sum^*$.
  $w \supset x$ – w jest *sufiksem* słowa x, gdy $x = yw$ dla pewnego słowa $y \in \sum^*$.
- $X[s..k]$ – fragment tekstu X od znaku o indeksie s do znaku o indeksie k.
- X_k – k-znakowy prefiks tekstu X. Jeżeli $X = X[0..t-1]$, to $X_k = X[0..k-1]$.

Przykład 1.
(a) Niech T: aabb**cad**bbba**cad**bdcbba**cad**ba będzie tekstem i P: **cad** – wzorcem. Mamy trzy wystąpienia wzorca P w tekście T: od pozycji $s = 5, s = 12, s = 21$.
(b) **ab**$\subset$**ab**cdca
ca$\supset$abcd**ca**

2. Algorytm naiwny (por. [BDR], [CLR])

Idea (algorytm naiwny)
Algorytm naiwny polega na sekwencyjnym przeglądaniu tekstu T. Przesuwamy się kolejno po jednym znaku tekstu T od strony lewej do prawej (iteracje od $s = 0$ do $n - m$) i w każdej iteracji sprawdzamy, czy kolejne znaki wzorca P pokrywają się z kolejnymi znakami tekstu T.

```
void naiwny(string T, string P) {
 int s = 0, j, n = T.length( ), m = P.length( );
 while(s <= n – m) {
  j = 0;
  while(j < m)
   if(P[j] == T[j+s]) j++; else break;
  if(j == m) cout << P << ” występuje w ” << T << ” od pozycji ” << (s + 1) << ”\n”;
  s++;
 }
}
```

Przykład 2. Niech T: **abr**akad**abrabr** będzie tekstem i P: **abr** – wzorcem.
Wystąpienia wzorca P w tekście T: od pozycji $s = 1, s = 8, s = 11$.

[Złożoność czasowa] (algorytm naiwny( ))
- Operacją elementarną są porównania między znakami wzorca i tekstu.
- Minimalną liczbę porównań, równą $n - m + 1$, mamy wtedy, gdy wzorzec nie występuje w tekście i stwierdzamy to po porównaniu pierwszego znaku wzorca z tekstem (przy każdym położeniu wzorca względem tekstu).

- Jeżeli wzorzec występuje na każdej pozycji tekstu (np., gdy T: a^n, P: a^m), to wykonujemy maksymalną liczbę porównań równą $T_{\max}(n,m) = = \Theta((n-m+1)m)$.

3. Algorytm Knutha–Morrisa–Pratta (KMP) (por. [BDR], [CLR])

Idea (algorytm Knutha-Morrisa-Pratta)

- Algorytm KMP( ) korzysta z funkcji pomocniczej Π (zwanej *funkcją prefiksową*), którą wyznacza się dla wzorca niezależnie od tekstu. Funkcja ta pozwala na zwiększenie (w określonych sytuacjach) przesunięcia wzorca względem tekstu, które w algorytmie naiwnym jest zawsze równe 1.
- $\Pi[k]$ – maksymalna długość prefiksu wzorca P, który jest równocześnie sufiksem P_k.
  Formalnie funkcja prefiksowa Π: $\{0, ..., m-1\} \to \{0, 1, ..., m-1\}$ jest określona następująco: $\Pi[k] = \max\{w: w < k, P_w \supset P_k\}$.
- Jeśli wiemy, że k znaków wzorca pasuje przy przesunięciu o s, to następne potencjalnie poprawne przesunięcie s' możemy wyliczyć jako $s' = s + +(k - \Pi[k-1])$.
- W najlepszym przypadku $s' = s + k$ i pomijamy wtedy przesunięcia $s+1$, $s+2$, $s+3$, ..., $s+k-1$.

Przykład 3. Niech P: ababaaa będzie wzorcem. Wówczas funkcja Π przyjmuje wartości:

$\Pi(0) = 0, \Pi(1) = 0, \Pi(2) = 1, \Pi(3) = 2, \Pi(4) = 3, \Pi(5) = 1, \Pi(6) = 1.$

```
ababaaa   ababaaa   ababaaa   ababaaa   ababaaa   ababaaa   ababaaa
ababaaa    ababaaa    ababaaa    ababaaa    ababaaa    ababaaa    ababaaa
Π(0)=0    Π(1)=0    Π(2)=1    Π(3)=2    Π(4)=3    Π(5)=1    Π(6)=1
```

```
int* pref(string P) {
  int m = P.length( ), k = 0;
  int* pi = new int[m];
  pi[0] = 0;
  for(int i = 1; i < m; i++) {
    while(k && P[k] != P[i]) k = pi[k–1];
    if(P[k] == P[i]) k++;
    pi[i] = k;
  }
  return pi;
}
```

```
void KMP(string T, string P) {
 int n = T.length( ), m = P.length( ), k = 0;
 int* pi = pref(P);
 for(int i = 0; i < n; i++) {
  while(k && P[k] != T[i]) k = pi[k–1];
  if(P[k] == T[i]) k++;
  if(k == m) {
   cout << ”wzorzec od pozycji ” << (i – m + 2) << ”\n”;
   k = pi[k–1];
  }
 } delete [ ]pi;
}
```

Przykład 4. Niech P: ababaaa będzie wzorcem i T: babababaaababaaacababaaa tekstem. Znajdź wszystkie wystąpienia wzorca P w tekście T. Skorzystaj z funkcji prefiksowej Π i algorytmu KMP( ).

```
       0 1 2 3 4 5 6
P:     a b a b a a a
Π      0 0 1 2 3 1 1

i:     0 1 2 3 4 5 6 7 8 9 10111213141516171819202 1
T:     b a b a b a a a b a b a a a c a b a b a a a
k:     0 1 2 3 4 5 6 7 2 3 4 5 6 7 0 1 2 3 4 5 6 7
                     1=Π[7-1]    1=Π[7-1]        1=Π[7-1]
                                   0=Π[1-1]
```

Wzorzec P występuje w tekście T od pozycji $s = 2$, $s = 8$ i $s = 16$.

[Złożoność czasowa] (algorytm KMP( ))

- Złożoność pesymistyczna obliczenia wszystkich wartości funkcji Π wynosi $O(m)$, gdyż:
  (1) warunek $P[k] = P[i]$ (po pętli) może być sprawdzany co najwyżej m razy oraz
  (2) warunek $P[k] \neq P[i]$ (w pętli) też może być sprawdzany co najwyżej m razy.
  Razem mamy co najwyżej $2m$ porównań.
- Analogicznie można uzasadnić, że złożoność pesymistyczna zasadniczego algorytmu KMP( ) jest szacowana przez $O(n)$.
- W związku z powyższym całkowita złożoność pesymistyczna algorytmu KMP( ) wynosi $O(n + m)$.

4. Algorytm Rabina–Karpa (RK) (por. [BDR], [CLR])

Idea (algorytm Rabina-Karpa)

- W algorytmie stosuje się metodę tzw. „odcisku palca". Zamiast porównywać wzorzec *P* z tekstem *T* znak po znaku, używa się specjalnej funkcji (zwanej „odciskiem palca"), która wiąże z każdym m-elementowym ciągiem (blokiem) znaków pewną liczbę, która staje się jego identyfikatorem. Zamiast porównywać ciągi znaków, porównuje się reprezentujące je liczby („odciski").
- Ogólnie przyjmuje się, że każdy znak jest cyfrą w systemie pozycyjnym o podstawie d, gdzie $d = |\Sigma|$. Każdy ciąg m kolejnych znaków traktuje się jako m-cyfrową liczbę w systemie pozycyjnym o podstawie d. Dla ułatwienia założymy, że alfabet Σ składa się tylko z cyfr ($\Sigma = \{0, 1, \dots, 9\}$, $|\Sigma| = 10$).
- Niech p oznacza liczbę odpowiadającą wzorcowi *P*, a t_s – liczbę odpowiadającą ciągowi znaków $T[s + 0..s + m - 1]$. Prawdziwa jest następująca własność:

  $t_s = p$ wtw $T[s + 0..s + m - 1] = P[0..m - 1]$.

- Liczbę p oraz t_0 wyznaczamy, korzystając z algorytmu Hornera (algorytm oblicza wartości wielomianu o współczynnikach *P*[0], *P*[1], ..., *P*[$m - 1$] dla argumentu $d = 10$):

  $$p = P[m-1] + 10\cdot(P[m-2] + 10\cdot(P[m-3] + \cdots + 10\cdot(P[1] + 10\cdot P[0]) \dots))$$

- Aby policzyć $t_1, ..., t_{n-m}$, wystarczy zauważyć, że do obliczenia t_{s+1} można użyć wyliczonej w poprzednim kroku wartości t_s (wartość 10^{m-1} wyznaczamy jednokrotnie):

  $t_{s+1} = 10\cdot(t_s - 10^{m-1}\cdot \mathrm{T}[s]) + \mathrm{T}[s + m]$

- Ponieważ zarówno moc zbioru Σ, jak i długość wzorca mogą być stosunkowo dużymi liczbami, algorytm korzysta z arytmetyki modulo q, gdzie q jest liczbą pierwszą (patrz problem 1).

Przykład 5. Niech *T*: 34587656743256989 będzie tekstem i *P*: 56983 – wzorcem. Mając odcisk $p = 56983$ oraz wartość $t_0 = 34587$, wyznacz wartości t_1, t_2, t_3.
$t_1 = 10\cdot(t_0 - 10^4\cdot 3) + 6 = 10\cdot 4587 + 6 = 45876$
$t_2 = 10\cdot(t_1 - 10^4\cdot 4) + 5 = 10\cdot 5876 + 5 = 58765$
$t_3 = 10\cdot(t_2 - 10^4\cdot 5) + 6 = 10\cdot 8765 + 6 = 87656$

```
int Hornerq(string P, int d, int m, int q) {
  int p = 0;
  for(int i = 0; i < m; i++) p = (d * p + (P[i] – '0')) % q;
  return p;
}
```

```
int powq(int d, int m, int q) {
 int h = 1;
 for(int i = 0; i < m; i++) h = (h * d) % q;
 return h;
}

void RK(string T, string P, int d, int q) {
 int n = T.length( ), m = P.length( );
 int p = Hornerq(P, d, m, q), t = Hornerq(T, d, m, q), h = powq(d, m – 1, q);
 for(int s = 0; s <= n – m; s++) {
  if(p == t) { int j;
   for(j = 0; j < m; j++) if(T[s+j] != P[j]) break;
   if(j == m) cout << "wzorzec na pozycji " << (s + 1) << "\n";
  }
  if(s < n – m) {
   t = (d * (t – (T[s] – '0') * h) + (T[s+m] – '0')) % q;
   if(t < 0) t += q;
  }
 }
}
```

Problem 1. Wartości $p, t_0, t_1, ..., t_{n-m}$ mogą być bardzo duże, a wtedy nie można stwierdzić, że wykonanie każdej operacji arytmetycznej dla p czy t_0 zabiera tyle samo czasu.

Rozwiązanie:
- Algorytm zamiast zwykłej arytmetyki stosuje arytmetyką modulo q (q – pewna liczba pierwsza). Wartość q jest zwykle taka, by $d{\cdot}q$ było nie większe niż jedno słowo maszynowe.
- Obliczanie następnego „odcisku palca" dla tekstu T wygląda następująco:

$$t_{s+1} = (d{\cdot}(t_s - d^{m-1}{\cdot}T[s]) + T[s+m]) \bmod q$$

Problem 2. Pojawia się problem niejednoznaczności, tzn. prawdziwość warunku $t_s \equiv p \bmod q$ nie oznacza, że na pewno $P[0..m-1] = T[s+0..s+m-1]$.

Rozwiązanie:
- Aby algorytm nie zwracał niepoprawnych wyników, należy w każdym przypadku, gdy $t_s \equiv p \bmod q$, dodatkowo sprawdzić równość odpowiednich ciągów, badając je znak po znaku.
- Powoduje to, że złożoność algorytmu RK( ), w najgorszym przypadku, szacuje się przez $\Theta((n-m+1)m)$.

Przykład: Dla $T = a^n$ i $P = a^m$ każdy blok tekstu o długości m daje ten sam odcisk równy odciskowi wzorca i konieczne jest sprawdzenie ciągów znak po znaku.

[Złożoność czasowa] (algorytm RK( ))

- Za operacje elementarne przyjmujemy tu operacje arytmetyczne realizowane przy obliczaniu odcisku wzorca i tekstu.
- Zarówno odcisk wzorca p, jak i odcisk t_0 możemy wyznaczyć, stosując algorytm Hornera, którego złożoność szacujemy przez $O(m)$.
- Mając wyznaczoną wartość t_0, pozostałe odciski t_1, ..., t_{n-m} obliczamy przy użyciu wzoru:

  $t_{s+1} = (d \cdot (t_s - d^{m-1} \cdot T[s]) + T[s+m]) \bmod q$

  Wówczas złożoność algorytmu obliczającego wszystkie wartości t_i (tj. dla i = 1, ..., $(n-m)$) szacujemy przez $O(n-m)$, ponieważ wartość 10^{m-1} (można tę wartość obliczyć w czasie $O(\log m)$ (por. [CLR])) wyznaczamy jednorazowo.
- Ponieważ w wielu zastosowaniach spodziewamy się małej liczby wystąpień wzorca w tekście, zakładamy, że oczekiwany czas realizacji całego algorytmu wynosi $O(n+m)$ plus czas potrzebny do weryfikacji błędnych „strzałów". Złożoność pesymistyczna algorytmu wynosi $\Theta((n-m+1)m)$.

5. Zadania różne

[01] Znajdź wszystkie wystąpienia wzorca *P* w tekście *T*. Zastosuj algorytm naiwny( ).

P: dfy, *T*: dfasdfuisdafyiusdfyiysdfyusdfiyu

Rozwiązanie: $s = 17$, $s = 23$.

[02] Niech *P* będzie wzorcem. Wyznacz funkcję prefiksową Π wykorzystywaną w algorytmie KMP( ).

(a) *P*: abrakadabra (b) *P*: banaraba (c) *P*: abaabbbabaa

Rozwiązanie:

(a) $\Pi(0) = 0, \Pi(1) = 0, \Pi(2) = 0, \Pi(3) = 1, \Pi(4) = 0, \Pi(5) = 1, \Pi(6) = 0, \Pi(7) = 1, \Pi(8) = 2, \Pi(9) = 3, \Pi(10) = 4$.

(b) $\Pi(0) = 0, \Pi(1) = 0, \Pi(2) = 0, \Pi(3) = 0, \Pi(4) = 0, \Pi(5) = 0, \Pi(6) = 1, \Pi(7) = 2$.

(c) $\Pi(0) = 0, \Pi(1) = 0, \Pi(2) = 1, \Pi(3) = 1, \Pi(4) = 2, \Pi(5) = 0, \Pi(6) = 0, \Pi(7) = 1, \Pi(8) = 2, \Pi(9) = 3, \Pi(10) = 4$.

[03] Znajdź wszystkie wystąpienia wzorca *P* w tekście *T*. Zastosuj algorytm KMP( ). Wyznacz funkcję prefiksową Π.

(a) *P*: ababaca, *T*: babababaaababaaacababaca,

(b) *P*: abcabcabbac, *T*: abcabcabcabcabbacaabab,

(c) *P*: abrakadabra, *T*: babababaaabrakadbabrabra.

Rozwiązanie:

(a)
```
      0 1 2 3 4 5 6
P:    a b a b a c a
Π     0 0 1 2 3 0 1
i:    0 1 2 3 4 5 6 7 8 9 10111213141516171819202l
T:    b a b a b a a a b a b a a a c a b a b a c a
k:    0 1 2 3 4 5 1 1 2 3 4 5 1 1 0 1 2 3 4 5 6 ~~7~~
                                                1
```

Wzorzec *P* występuje w tekście *T* od pozycji $s = 16$.

(b)
```
      0 1 2 3 4 5 6 7 8 9 10
P:    a b c a b c a b b a c
Π     0 0 0 1 2 3 4 5 0 1 0
i:    0 1 2 3 4 5 6 7 8 9 10111213141516171819202l
T:    a b c a b c a b c a b c a b b a c a a b a b
k:    1 2 3 4 5 6 7 8 6 7 8 6 7 8 9 10~~11~~1 1 2 1 2
                                      0
```

Wzorzec *P* występuje w tekście *T* od pozycji $s = 7$.

(c)
```
      0 1 2 3 4 5 6 7 8 9 10
P:    a b r a k a d a b r a
Π     0 0 0 1 0 1 0 1 2 3 4
i:    0 1 2 3 4 5 6 7 8 9 10111213141516171819202l
T:    b a b a b a a a b r a k a d b a b r a b r a
k:    0 1 2 1 2 1 1 1 2 3 4 5 6 7 0 1 2 3 4 2 3 4
```

Wzorzec *P* nie występuje w tekście *T*.

[04] Wyznacz odcisk *p* wzorca *P* dla algorytmu Rabina-Karpa, jeśli znaki tekstu są wybierane z alfabetu Σ.

(a) *P*: 84553, Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8},

(b) *P*: 1010, Σ = {0, 1, 2}.

Rozwiązanie: Przyjęto $q = 13$.

(a) |Σ| = 9,
p = (9 * 0 + 8) mod 13 = 8, *p* = (9 * 8 + 4) mod 13 = 11,
p = (9 * 11 + 5) mod 13 = 0, *p* = (9 * 0 + 5) mod 13 = 5,
p = (9 * 5 + 3) mod 13 = **9**

(b) |Σ| = 3,
p =(3 * 0 + 1) mod 13 = 1, *p* = (3 * 1 + 0) mod 13 = 3, *p* = (3 * 3 + 1) mod 13 = 10,
p = (3 * 10 + 0) mod 13 = **4**

[05] Znajdź wszystkie „strzały” i poprawne wystąpienia wzorca *P* w tekście *T*. Zastosuj algorytm RK( ).

P: 84553, *T*: 1617584553416, Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8 }

Rozwiązanie: Przyjęto $q = 13$.
$p = 9$,
16175, $t_0 = 8$
61758, $t_1 = 12$
17584, $t_2 = 3$
75845, $t_3 = 3$
58455, $t_4 = 11$
84553, $t_5 = 9 = p$, trzeba sprawdzić znaki
45534, $t_6 = 9 = p$, trzeba sprawdzić znaki
55341, $t_7 = 5$
53416, $t_8 = 10$

Wykaz skrótów i oznaczeń

$\varnothing$ – zbiór pusty
$E(x)$ – część całkowita liczby x
$x \mid y$ – liczba całkowita y *dzieli się przez* liczbę całkowitą x
$x||y$ – konkatenacja (złożenie) ciągów x i y
LI[ ] – tablica list incydencji grafu $G = (V, E)$
lg$n$ – $\log_2 n$
λ – słowo puste
N_+ – zbiór liczb naturalnych dodatnich
N – zbiór liczb naturalnych, $N = N_+ \cup \{0\}$
pr(X) – prawdopodobieństwo zdarzenia X
R_+ – zbiór liczb rzeczywistych dodatnich
sup{} – kres górny zbioru
wtw – „wtedy i tylko wtedy, gdy”
$=^{(a)}$ – równość ta zachodzi na mocy podanego poniżej faktu (a)
$w \subset x$ – w jest prefiksem słowa x
$w \supset x$ – w jest sufiksem słowa x
$|X|$ – moc zbioru X
Z_+ – zbiór liczb całkowitych dodatnich
$\ell(f)$ – ważona średnia długości kodowania f
ZIM – skrót dla „zasada indukcji matematycznej”

Literatura

[AHU] Aho A.V., Hopcroft J.E., Ullman J.D., *Algorytmy i struktury danych*, Helion, Gliwice 2003.

[BDR] Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, Wydawnictwo Naukowo-Techniczne, Warszawa 1999.

[CLR] Cormen T.H., Leiserson Ch.E., Rivest R.L., *Wprowadzenie do algorytmów*, Wydawnictwo Naukowo-Techniczne, Warszawa 2000.

[DA] Drozdek A., *C++, Algorytmy i struktury danych*, Helion, Gliwice 2004.

[DS] Drozdek A., Simon D.L., *Struktury danych w języku C*, Wydawnictwo Naukowo-Techniczne, Warszawa 1996.

[EB] Eckel B., *Thinking in C++*, Helion, Gliwice 2002.

[GŚ] Gerstenkorn T., Śródka T., *Kombinatoryka i rachunek prawdopodobieństwa*, Państwowe Wydawnictwo Naukowe, Warszawa 1976.

[GM] Grabowski M., *Ćwiczenia z analizy matematycznej dla nauczycieli*, Państwowe Wydawnictwo Naukowe, Warszawa 1980.

[KW] Krysicki W., Włodarski L., *Analiza matematyczna w zadaniach. Część I*, Państwowe Wydawnictwo Naukowe, Warszawa 1974.

[LW] Lipski W., *Kombinatoryka dla programistów*, Wydawnictwo Naukowo-Techniczne, Warszawa 1982.

[MG] Mirkowska-Salwicka G., *Wykłady z „Algorytmów i struktur danych"*, Politechnika Białostocka.

[RW] Ross K.A., Wright Ch.R.B., *Matematyka dyskretna*, Wydawnictwo Naukowe PWN, Warszawa 1996.

[SD] Stinson D.R., *Kryptografia w teorii i praktyce*, Wydawnictwo Naukowo-Techniczne, Warszawa 2005.

[SDK] Sysło M.M., Deo N., Kowalik J.S., *Algorytmy optymalizacji dyskretnej*, Wydawnictwo Naukowe PWN, Warszawa 1999.

[WR] Wilson R.J., *Wprowadzenie do teorii grafów*, Wydawnictwo Naukowe PWN, Warszawa 1998.

[WN] Wirth N., *Algorytmy + struktury danych = programy*, Wydawnictwo Naukowo-Techniczne, Warszawa 2002.

[WP] Wróblewski P., *Algorytmy, struktury danych i techniki programowania*, Helion, Gliwice 1997.

SKRYPT ZAWIERA:

- wykład do przedmiotu *algorytmy i struktury danych* (*Część I. Analiza i techniki projektowania algorytmów*) przeznaczony dla studentów studiów informatycznych;
- techniki projektowania algorytmów oraz znane algorytmy przedstawione za pomocą prostych, intuicyjnych schematów;
- przykłady i ćwiczenia z pełnymi rozwiązaniami do samodzielnej nauki;
- znane algorytmy zapisane w języku C++.

BOOK CONTENTS:

- lecture notes on *Algorithms and Data Structures* (*Part I. Algorithmic analysis and algorithm design techniques*) for students of computer science;
- algorithm design techniques and known algorithms presented through the use of simple, intuitive patterns;
- examples and exercises with complete solutions for self-study;
- known algorithms written in C++.

Politechnika Białostocka