Algorytmy i struktury danych (15h) – notatki do kursu

mgr. inż Dominik Filipiak, dr Piotr Arendarski

Rok akademicki 2020/2021

Spis treści

Dokument ten jest pomocą dla prowadzącego i nie zastępuje w żaden sposób podręcznika akademickiego. W szczególności nauka z tego dokumentu nie jest gwarantem zdania egzaminu. Dokument powstał głównie na podstawiem książki Cormena i in. *Wprowadzenie do algorytmów*. Autor nie odpowiada ze ewentualne błędy w tym dokumencie.

1 Wprowadzenie

1.1 O prowadzącym

dr Piotr Arendarski, Katedra Informatyki Ekonomicznej, `piotr.arendarski@dalejstandardowo` (proszę zaczynać tytuły wiadomości od [ASD]). Konsultacje : środa 18:30 – 19:30, czwartek 9:30 – 10:30 na platfromie MS Teams

1.2 Egzamin

Egzamin z przedmiotu (wykład + ćwiczenia) odbędzie się:

- I termin: 4 luty,
- II termin: 18 luty (ostatni czwartek sesji egz.),
- III termin: 4 marca (ostatni czwartek poprawkowej sesji egz.).

Progi punktowe określa prof. Abramowicz. Zaliczenie Obowiązują ogólne zasady zaliczenia w KIE. Obecność na ćwiczeniach jest obowiązkowa. Każda nieobecność nieusprawiedliwiona począwszy od trzeciej włącznie to ujemne punkty na egzaminie (-3%). Zgodnie z regulaminem studiów, przy ponad połowie nieobecności (usprawiedliwionej bądź nie) jestem zmuszony przedstawić taką osobę dyrektorce studiów do skreślenia z listy studentów. Skany usprawiedliwień proszę przesyłać mailem w terminie zgodnych z zasadami zaliczania w KIE.

1.3 Literatura

- Cormen Thomas H., Leiserson Charles E., Rivest Ronald L, Clifford Stein. *Wprowadzenie do algorytmów*. Wydawnictwo Naukowe PWN, Warszawa, 2000 - 2019.
- Lutz Mark. *Python. Wprowadzenie*. Wydanie IV, Helion, 2011.
- Cormen Thomas H. *Algorytmy bez tajemnic*. Helion, 2012-2018.

Większość użytych tu przykładów będzie pochodziła z książki Cormena WdA i reszty.

2 Proste algorytmy

Definicja algorytmu, kod a pseudokod.

2.1 Pierwiastki kwadratowe

Niech $a, b, c \in \mathbb{R}, a \neq 0$. Pierwiastki równania kwadratowego o postaci $y = ax + bx + c$ wyliczamy korzystając ze znanego ze szkoły średniej algorytmu.

Algorithm 1 Pierwiastki rzeczywiste równania kwadratowego

function QUADRATIC-ROOTS(a, b, c)
    $\Delta \leftarrow b^2 - 4ac$
    if $\Delta > 0$ **then**
        $x_1 \leftarrow \frac{-b-\sqrt{\Delta}}{2a}$
        $x_2 \leftarrow \frac{-b+\sqrt{\Delta}}{2a}$
        return $\{x_1, x_2\}$
    else if $\Delta = 0$ **then**
        $x \leftarrow \frac{-b}{2a}$
        return $\{x\}$
    else
        return $\{\varnothing\}$

2.2 Algorytm Euklidesa

Wprowadźmy najpierw operację dzielenia modulo.

$$a \bmod b = r \quad \Rightarrow \quad a = bn + r, \qquad |n| > r \geqslant 0$$

Przykład.

$$\begin{aligned} 7 \bmod 6 &= 1, && (\text{ponieważ } 7 = 6 \cdot 1 + 1) \\ 17 \bmod 7 &= 3, && (\text{ponieważ } 17 = 7 \cdot 2 + 3) \\ -14 \bmod 2 &= 0 \\ 9 \bmod 6 &= 3 \\ -17 \bmod 7 &= 4 \end{aligned}$$

Według algorytmu Euklidesa, NWD(a, b), gdzie $a, b \in$ wyliczymy w poniższy sposób:

$$\begin{aligned} a &= q_1 b + r_1 \\ b &= q_2 r_1 + r_2 \\ r_1 &= q_3 r_2 + r_3 \\ r_2 &= q_4 r_3 + r_4 \\ &\vdots \\ r_{n-2} &= q_n r_{n-1} + r_n \\ r_{n-1} &= q_{n+1} r_n + 0 \end{aligned}$$

Jeżeli $r_{n+1} = 0$, to NWD(a, b) jest równe r_n.

Przykład. Przykład: NWD dla 1071 oraz 462.

$$\begin{aligned} 1071 &= 2 \cdot 462 + 147 \\ 462 &= 3 \cdot 147 + 21 \\ 147 &= 7 \cdot 21 + 0 \end{aligned}$$

Wynikiem jest 21.

2.3 Silnia (iteracyjnie)

Silnię definiujemy w następujący sposób:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n = \prod_{k=1}^{n} k \qquad \text{dla } n \in \mathbb{N} \tag{1}$$

Już 11! to więcej, niż jest ludzi w Polsce.

Algorithm 2 Algorytm Euklidesa

```
function EUCLID(a, b)
    r ← a mod b
    while r ≠ 0
        a ← b
        b ← r
        r ← a mod b
    return b
```

3 Listy i iteracje

Tablica jest uporządkowaną kolekcją, w której każdy element ma swój indeks (dostęp bezpośredni po indeksie w stałym czasie). Tablica jednowymiarowa jako wektor, dwuwymiarowa jako macierz, n-wymiarowa jako coś w rodzaju tensora. Lista (jednokierunkowa) jest zbiorem elementów uporządkowanym liniowo (dostęp sekwencyjny, czas zależny od długości listy). Niestety w języku Python pojęcia listy i tablicy są nieco pomieszane w stosunku do kanonu informatyki, tj. standardową strukturą danych jest coś na wzór ich hybrydy (o nazwie listy). Różnicę między listą a tablicą dobrze widać w C++.

3.1 Tworzenie i wydruk listy w Pythonie

```python
list = [1, 2, 3, 4]

for i in list:
    print(i)
```

3.2 Tworzenie macierzy kwadratowej

```python
X = [[12,7,3],
     [4 ,5,6],
     [7 ,8,9]]

a = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
for row in a:
    for elem in row:
        print(elem, end=' ')
    print()
```

3.3 Proste operacje na wektorze i macierzy

Powiemy, że $C = AB$ dla macierzy A o wymiarach $n \times m$ oraz macierzy B o wymiarach $m \times p$, gdzie $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$. Złożoność obliczeniowa rzędu $\Theta(n^3)$ (dla macierzy $n \times n$, nieformalnie - bo trzy pętle) lub $\Theta(nmp)$ (dla macierzy $n \times m$ oraz $m \times p$)

Algorithm 3 Mnożenie macierzy

```
function MATMUL(A, B)
    C ← nowa macierz o wymiarach n × p
    for i from 1 to n
        for j from 1 to p
            sum ← 0
            for k from 1 to m
                sum ← sum + A_ik · B_kj
            C_ij ← sum
    return C
```

```python
# source: https://www.programiz.com/python-programming/examples/multiply-matrix
# 3x3 matrix
X = [[12,7,3],
     [4 ,5,6],
     [7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],
     [6,7,3,0],
     [4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],
          [0,0,0,0],
          [0,0,0,0]]

for i in range(len(X)):
   for j in range(len(Y[0])):
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]

for r in result:
   print(r)
```

4 Rekurencja

Podprogramy, rekurencja

4.1 Silnia (rekurencyjnie)

Niech $n \in \mathbb{N}$.

$$n! = \begin{cases} 1 & \text{jeżeli } n = 1 \\ n \cdot (n-1)! & \text{w każdym innym przypadku} \end{cases}$$

Przykład.

$$\begin{aligned} 5! &= 5 \cdot 4! \\ &= 5 \cdot 4 \cdot 3! \\ &= 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 120 \end{aligned}$$

Algorithm 4 Silnia (rekurencyjnie)

1: **function** FACTORIAL(n)
2: **if** $n = 1$ **then**
3: **return** n
4: **else**
5: **return** $n\cdot$FACTORIAL$(n-1)$

```python
# source https://www.programiz.com/python-programming/examples/factorial-recursion
def factorial(n):
   if n == 1:
       return n
   else:
       return n*factorial(n-1)
```

4.2 Ciag Fibonacciego, min, max

Ciąg Fibonacciego to ciąg, w którym każdy element począwszy od trzeciego jest sumą dwóch poprzednich elementów. Niech $n \in \mathbb{N}$.

$$F(n) = \begin{cases} 0 & \text{jeżeli } n = 0 \\ 1 & \text{jeżeli } n = 1 \\ F_{n-1} + F_{n-2} & \text{w każdym innym przypadku} \end{cases}$$

```python
def F(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return F(n-1)+F(n-2)
```

Przykład.

$$\begin{aligned} 5! &= 5 \cdot 4! \\ &= 5 \cdot 4 \cdot 3! \\ &= 5 \cdot 4 \cdot 3 \cdot 2! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! \\ &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 120 \end{aligned}$$

Min i max w notebooku

4.3 Algorytm Euklidesa (rekurencyjnie)

```python
def gcd(a,b):
    if a % b == 0:
       return b
    return gcd(b, a % b)
```

5 Sortowanie (cz. 1) oraz analiza algorytmów

5.1 Sortowanie przez wstawianie

Nieformalnie – sortujemy jak talię kart, od lewej.

Przykład. Rozpatrzmy następującą tablicę. Zaczynamy od drugiego elementu:

$$\begin{bmatrix} 5 & 2 & 4 & 6 & 1 & 3 \end{bmatrix}$$

Ponieważ $A[1] > A[0]$, to zamieniamy dwa pierwsze elementy miejscami i mamy:

$$\begin{bmatrix} 2 & 5 & 4 & 6 & 1 & 3 \end{bmatrix}$$

Następnie rozpatrzmy trzeci element, czwórkę. Po przestawieniu mamy:

$$\begin{bmatrix} 2 & 4 & 5 & 6 & 1 & 3 \end{bmatrix}$$

Szóstka jest w dobrym miejscu, nie zmienia się nic:

$$\begin{bmatrix} 2 & 4 & 5 & 6 & 1 & 3 \end{bmatrix}$$

Jedynka wędruje na sam początek:

$$\begin{bmatrix} 1 & 2 & 4 & 5 & 6 & 3 \end{bmatrix}$$

Pozostaje nam wziąć się za trójkę:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

Algorithm 5 Sortowanie przez wstawianie (przykład z książki – liczymy od 1!)

		koszt	krotność
1:	**function** InsertionSort(A)	koszt	krotność
2:	**for** $j \leftarrow 2$ **to** $A.length$	c_1	n
3:	$key \leftarrow A[j]$	c_2	$n-1$
4:	// Wstaw $A[j]$ w posortowany ciąg $A[1 \dots j-1]$	0	$n-1$
5:	$i \leftarrow j-1$	c_4	$n-1$
6:	**while** $i > 0$ **and** $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
7:	$A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
8:	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
9:	$A[i+1] \leftarrow key$	c_8	$n-1$

W tym algorytmie fragment pętli z już posortowanymi liczbami nazwiemy *niezmiennikiem pętli*. Dowodzimy poprawności algorytmów przez dowód trzech rzeczy dotyczących niezmiennika pętli: *inicjowania* (niezmiennik prawdziwy przed iteracją), *utrzymania* (jeżeli jest prawdziwy przed iteracją pętli, to jest prawdziwy w kolejnej iteracji) oraz *zakończenia* (po zakończeniu pętli z niezmiennika wynika coś istotnego dla algorytmu).

```
1  def insertionSort(A):
2    print(A)
3    for j in range(1, len(A)):
4      key = A[j]
5      i = j - 1
6      while i >= 0 and A[i] > key:
7        A[i+1] = A[i]
8        i = i - 1
9      A[i+1] = key
10     print(A)
```

5.2 Analiza złożoności obliczeniowej sortowania przez scalanie

Zbadajmy czas działania $T(n)$, gdzie n jest długością wejściowej tablicy:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

W przypadku optymistycznym (gdy na wejściu dostajemy posortowaną tablicę) mamy $t_j = 1$ i tym samym omijamy linie 7 oraz 8 w algorytmie:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Powyższe z kolei można przedstawić jako $an + b$ dla pewnych stałych a oraz b, a więc jest to funkcja liniowa.

Co w przypadku pesymistycznym, gdy dostajemy *najgorszą* tablicę do posortowania? Każdy element będzie musiał być przesuwany do końca, więc $t_j = j$.

$$\begin{bmatrix} 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

Wiedząc[1], że:

$$\sum_{k=1}^{n} k = S$$
$$S = n + (n-1) + \ldots + 2 + 1$$
$$S = 1 + 2 + \ldots + (n-1) + n$$
$$2S = (n+1) + (n+1) + (n+1) + \ldots + (n+1) = n(n+1)$$
$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

mamy:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1, \qquad \sum_{j=2}^{n} (j-1) = \frac{n(n+1)}{2}.$$

Tak więc w najgorszym wypadku:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n+1)}{2}\right) + c_7\left(\frac{n(n+1)}{2}\right) + c_8(n-1) \\ &= \frac{1}{2}\left(c_5 + c_6 + c_7\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{1}{2}\left(c_5 - c_6 - c_7\right) + c^8\right) n \end{aligned}$$

Powyższe można przedstawić jako $an^2 + bn + c$ dla pewnych stałych a, b i c – jest to więc funkcja kwadratowa.

5.3 Dziel i zwyciężaj

Metoda dziel i zwyciężaj ma trzy fazy:

Dziel. Dzielimy problem na mniejsze podproblemy.

Zwyciężaj Rozwiązujemy podproblemy rekurencyjnie – jeżeli są dostatecznie małe, to robimy to bezpośrednio.

Połącz Scalamy cząstkowe wyniki w jedno rozwiązanie naszego problemu.

Rekurencję dla tej metody rozwiązujemy w ten sposób:

$$T(n) = \begin{cases} \Theta(1), & \text{jeżeli } n \leqslant c \\ aT(n/b) + D(n) + C(n) & \text{w każdym innym przypadku,} \end{cases}$$

gdzie n to rozmiar problemu, c wyznacza mały rozmiar problemu który można od razu rozwiązać w stałym czasie, $aT(n/b)$ onacza a podproblemów, każdy w rozmiarze b/n, $D(n)$ czas dzielenia, a $C(n)$ to czas scalaania

5.4 Sortowanie przez scalanie

Przykład.

$$\begin{bmatrix}6 & 5 & 3 & 1 & 8 & 7 & 2 & 4\end{bmatrix}$$

Dzielimy i zwyciężamy (sortujemy)

$$\begin{bmatrix}6 & 5 & 3 & 1\end{bmatrix}\begin{bmatrix}8 & 7 & 2 & 4\end{bmatrix}$$
$$\begin{bmatrix}6 & 5\end{bmatrix}\begin{bmatrix}3 & 1\end{bmatrix}\begin{bmatrix}8 & 7 & 2 & 4\end{bmatrix}$$
$$\begin{bmatrix}6\end{bmatrix}\begin{bmatrix}5\end{bmatrix}\begin{bmatrix}3 & 1\end{bmatrix}\begin{bmatrix}8 & 7 & 2 & 4\end{bmatrix}$$
$$\begin{bmatrix}5 & 6\end{bmatrix}\begin{bmatrix}3 & 1\end{bmatrix}\begin{bmatrix}8 & 7 & 2 & 4\end{bmatrix}$$
$$\begin{bmatrix}5 & 6\end{bmatrix}\begin{bmatrix}3\end{bmatrix}\begin{bmatrix}1\end{bmatrix}\begin{bmatrix}8 & 7 & 2 & 4\end{bmatrix}$$
$$\begin{bmatrix}5 & 6\end{bmatrix}\begin{bmatrix}1 & 3\end{bmatrix}\begin{bmatrix}8 & 7 & 2 & 4\end{bmatrix}$$

[1] Patrz szereg $1 + 2 + 3 + 4 + \ldots$

$$\begin{bmatrix}1 & 3 & 5 & 6\end{bmatrix}\begin{bmatrix}8 & 7\end{bmatrix}\begin{bmatrix}2 & 4\end{bmatrix}$$

$$\begin{bmatrix}1 & 3 & 5 & 6\end{bmatrix}\begin{bmatrix}8\end{bmatrix}\begin{bmatrix}7\end{bmatrix}\begin{bmatrix}2 & 4\end{bmatrix}$$

$$\begin{bmatrix}1 & 3 & 5 & 6\end{bmatrix}\begin{bmatrix}7 & 8\end{bmatrix}\begin{bmatrix}2 & 4\end{bmatrix}$$

$$\begin{bmatrix}1 & 3 & 5 & 6\end{bmatrix}\begin{bmatrix}7 & 8\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\begin{bmatrix}4\end{bmatrix}$$

$$\begin{bmatrix}1 & 3 & 5 & 6\end{bmatrix}\begin{bmatrix}7 & 8\end{bmatrix}\begin{bmatrix}2 & 4\end{bmatrix}$$

$$\begin{bmatrix}1 & 3 & 5 & 6\end{bmatrix}\begin{bmatrix}2 & 4 & 7 & 8\end{bmatrix}$$

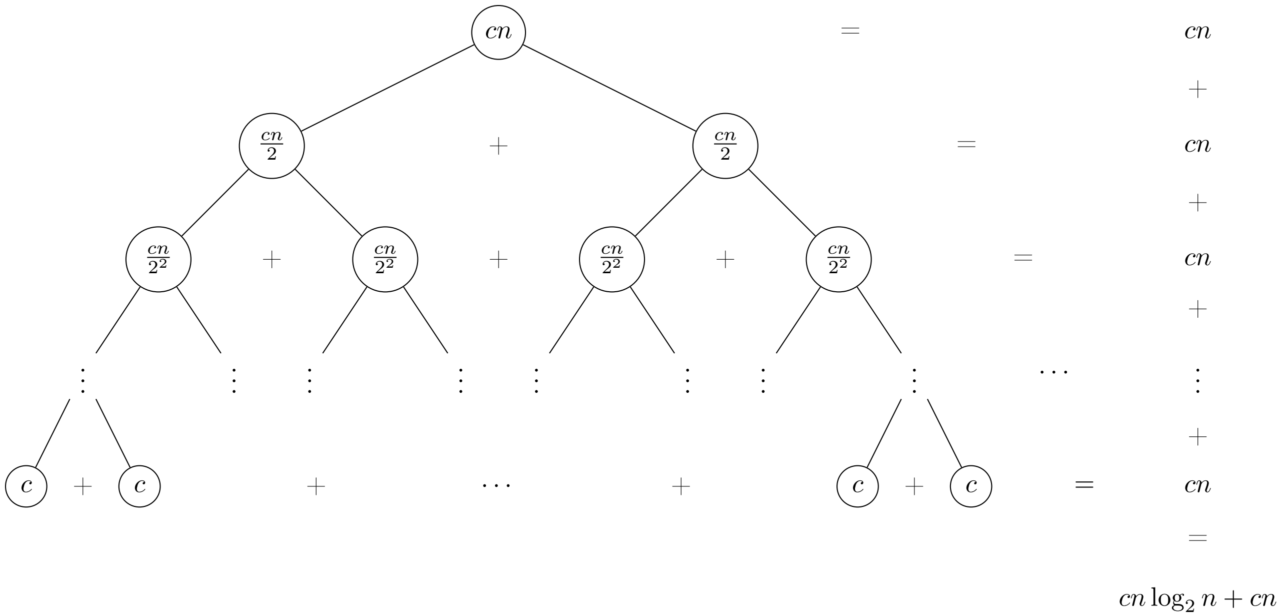
$$\begin{bmatrix}1 & 2 & 3 & 4 & 5 & 6 & 7 & 8\end{bmatrix}$$

Wizualizacja: `https://commons.wikimedia.org/wiki/File:Merge-sort-example-300px.gif` oraz `https://www.geeksforgeeks.org/wp-content/uploads/Merge-Sort-Tutorial.png`

Algorithm 6 Sortowanie przez scalanie (przykład z książki – liczymy od 1!)

function MERGESORT(A, p, r)
$\quad$ **if** $p < r$ **then**
$\quad\quad$ $q \leftarrow \lfloor (p + r)/2 \rfloor$
$\quad\quad$ MERGESORT(A, p, q)
$\quad\quad$ MERGESORT$(A, q + 1, r)$
$\quad\quad$ MERGE(A, p, q, r)
function MERGE(A, p, q, r)
$\quad$ $n_1 \leftarrow q - p + 1$
$\quad$ $n_2 \leftarrow r - q$
$\quad$ $L[1..n_1 + 1] \leftarrow$ nowa tablica
$\quad$ $R[1..n_2 + 1] \leftarrow$ nowa tablica
$\quad$ **for** i $\leftarrow 1$ **to** n_1
$\quad\quad$ $L[i] \leftarrow A[p + i - 1]$
$\quad$ **for** j $\leftarrow 1$ **to** n_2
$\quad\quad$ $R[j] \leftarrow A[q + j]$
$\quad$ $L[n_1 + 1] \leftarrow \infty$
$\quad$ $R[n_2 + 1] \leftarrow \infty$
$\quad$ $i \leftarrow 1$
$\quad$ $j \leftarrow 1$
$\quad$ **for** k $\leftarrow p$ **to** r
$\quad\quad$ **if** $L[i] \leqslant R[j]$ **then**
$\quad\quad\quad$ $A[k] \leftarrow L[i]$
$\quad\quad\quad$ $i \leftarrow i + 1$
$\quad\quad$ **else**
$\quad\quad\quad$ $A[k] \leftarrow R[j]$
$\quad\quad\quad$ $j \leftarrow j + 1$

```python
def merge(left, right):
  A = []
  i ,j = 0, 0
  while i < len(left) and j < len(right):
    if left[i] <= right[j]:
      A.append(left[i])
      i += 1
    else:
      A.append(right[j])
      j += 1
  A += left[i:]
  A += right[j:]
  return A
```



Rysunek 1: Wizualizacja rekurencji dla sortowania przez scalanie.

```python


def mergesort(A):
        if len(A) > 1:
                q = len(A) // 2
                left = mergesort(A[:q])
                right = mergesort(A[q:])
                return merge(left, right)
        return A

A = [2, 4, 5, 7, 1, 2, 3, 6]
print("unsorted(A): " + str(A))
print("mergesort(A): " + str(mergesort(A)))
```

5.5 Analiza złożoności obliczeniowej sortowania przez scalanie

Dla uproszczenia możemy założyć, że n jest potęgą 2.

Dziel. Znajdujemy środek przedziału i dzielimy w stałym czasie – $D(n) = \Theta(1)$

Zwyciężaj Rozwiązujemy 2 podproblemy rekurencyjnie, każdy o rozmiarze $n/2$, co w sumie daje $2T(n/2)$

Połącz Procedura *merge* działa w czasie $C(n) = \Theta(n)$ (trzy pętle bez zagnieżdżeń, $\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2) = \Theta(n)$)

$$T(n) = \begin{cases} \Theta(1), & \text{jeżeli } n \leqslant c, \\ aT(n/b) + D(n) + C(n) & \text{w każdym innym przypadku.} \end{cases}$$

Podstawiając:

$$T(n) = \begin{cases} \Theta(1), & \text{jeżeli } n = 1, \\ 2T(n/2) + \Theta(1) + \Theta(n) & \text{jeżeli } n > 1. \end{cases}$$

$\Theta(1)$ jest pomijalne ze względu na $\Theta(n)$.

Wysokość tego drzewa to $\log_2 n$, czyli jest $\log_2 n + 1$ poziomów. Każdy poziom kosztuje cn, a więc czas działania to $cn(\log_2 n + 1) = cn \log_2 n + cn = \Theta(n \log n)$.

5.6 Quicksort

Algorithm 7 Sortowanie szybkie (przykład z książki – liczymy od 1!)

function $\textsc{Quicksort}(A, p, r)$
  if $p < r$ **then**
    $q \leftarrow \textsc{Partition}(A, p, r)$
    $\textsc{Quicksort}(A, p, q-1)$
    $\textsc{Quicksort}(A, q+1, r)$
function $\textsc{Partition}(A, p, r)$ $\Theta(n)$
  $x \leftarrow A[r]$
  $i \leftarrow p - 1$
  for $j \leftarrow p$ **to** $r - 1$ c $n = r - p + 1$
    if $A[j] \leqslant x$ **then**
      $i \leftarrow i + 1$
      zamień $A[i]$ z $A[j]$
  zamień $A[i+1]$ z $A[r]$
  return $i + 1$

```python
def quicksort(A, p, r):
  print('Entering QuickSort!')
  if p < r:
    q = partition(A, p, r)
    quicksort(A, p, q-1)
    quicksort(A, q+1, r)
  else:
    print("p >= r ({} >= {}), nothing to sort here...".format(p, r))

def partition(A, p, r):
  print('Pivot: {}, p={}, r={}\t Array: \t{}'.format(A[r], p, r, A))
  x = A[r]
  i = p - 1
  for j in range(p, r):
    if A[j] <= x:
      i += 1
      A[i], A[j] = A[j], A[i]
    print('\t\t\t Subarray:\t{}'.format(A[p:r]))
  A[i+1], A[r] = A[r], A[i+1]
  return i + 1

a = [2, 8, 7, 1, 3 ,5, 6, 4]
quicksort(a, 0, len(a)-1)
print(a)
```

Zawsze dzielimy tablicę na 4 elementy, kolejno: elementy $\leqslant x$, elementy $> x$, elementy jeszcze nie sprawdzowone, oraz x. Na końcu (gdy trzeci obszar jest pusty) przestawiamy x pomiędzy dwa pierwsze obszary. Z lotu ptaka:

Krok 1: wybieramy $A[r] = 4$ (ostatni element):

2 8 7 1 3 5 6 4

Krok 2: liczby mniejsze od 4 idą na lewo, a większe na prawo

2 1 3 4 7 5 6 8

Krok 3: Powtarzamy krok 1 dla dwóch podlist

2 1 3 4 7 5 6 8

Krok 4: Powtarzamy krok 1 dla dwóch kolejnych podlist

2 1 3 4 7 5 6 8

Krok 5: Pozostały nam same jednoelementowe listy, których nie trzeba sortować

1 2 3 4 5 6 7 8

Krok 6: Gotowe!

1 2 3 4 5 6 7 8

A teraz dokładniej. Pierwsze wywołanie:

$$[2\ \ 8\ \ 7\ \ 1\ \ 3\ \ 5\ \ 6]\ [4]$$
$$[2]\ [8\ \ 7\ \ 1\ \ 3\ \ 5\ \ 6]\ [4]$$
$$[2]\ [8]\ [7\ \ 1\ \ 3\ \ 5\ \ 6]\ [4]$$
$$[2]\ [8\ \ 7]\ [1\ \ 3\ \ 5\ \ 6]\ [4]$$
$$[2\ \ 1]\ [7\ \ 8]\ [3\ \ 5\ \ 6]\ [4]$$
$$[2\ \ 1\ \ 3]\ [8\ \ 7]\ [5\ \ 6]\ [4]$$
$$[2\ \ 1\ \ 3]\ [8\ \ 7\ \ 5]\ [6]\ [4]$$
$$[2\ \ 1\ \ 3]\ [8\ \ 7\ \ 5\ \ 6]\ [4]$$
$$[2\ \ 1\ \ 3]\ [\mathbf{4}]\ [7\ \ 5\ \ 6\ \ 8]$$

Drugie wywołanie:

$$[2\ \ 1]\ [3]\ [\mathbf{4}]\ [7\ \ 5\ \ 6\ \ 8]$$
$$[2]\ [1]\ [3]\ [\mathbf{4}]\ [7\ \ 5\ \ 6\ \ 8]$$
$$[2\ \ 1]\ [3]\ [\mathbf{4}]\ [7\ \ 5\ \ 6\ \ 8]$$
$$[2\ \ 1]\ [\mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6\ \ 8]$$

Trzecie:

$$[2]\ [1]\ [\mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6\ \ 8]$$
$$[1\ \ 2]\ [\mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6\ \ 8]$$
$$[1]\ [\mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6\ \ 8]$$

Czwarte na lewo od 1 puste. Piąte na prawo od 2 puste. Szóste na prawo od 3 puste. Siódme:

$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6]\ [8]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7]\ [5\ \ 6]\ [8]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7\ \ 5]\ [6]\ [8]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6]\ [8]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7\ \ 5\ \ 6]\ [\mathbf{8}]$$

Ósme:

$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7\ \ 5]\ [6]\ [\mathbf{8}]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [7]\ [5]\ [6]\ [\mathbf{8}]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [5]\ [7]\ [6]\ [\mathbf{8}]$$
$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ 4]\ [5]\ [\mathbf{7}]\ [6]\ [\mathbf{8}]$$

Dziewiąte, dziesiąte i jedenaste puste. Finalnie:

$$[\mathbf{1}\ \ \mathbf{2}\ \ \mathbf{3}\ \ \mathbf{4}\ \ \mathbf{5}\ \ \mathbf{6}\ \ \mathbf{7}\ \ \mathbf{8}]$$

Przypadek pesymistyczny. Algorytm działa najwolniej, gdy procedura tworzy jeden obszar złożony z $n-1$ elementów, a drugi jest pusty. Jeżeli założymy, że algorytm takie podziały tworzy takie podziały, to mamy

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Okazuje się, że w przypadku pesymistycznym algorytm sortowania szybkiego jest asymptotycznie dokładnie oszacowany przez $\Theta(n^2)$ – a sam Quicksort jest tym samym $O(n^2)$. Co więcej – tablica posortowana jest przypadkiem pesymistycznym!

Dowód. Korzystając z metody rozwiązywania rekurencji przez podstawianie załóżmy, że $\Theta(n)$ to $c_2 n$ i sprawdźmy, czy $T(n) \leqslant c_1 n^2$:

$$\begin{aligned} T(n) &= T(n-1) + c_2 n \\ &\leqslant c_1(n-1)^2 + c_2 n \\ &= c_1 n^2 - 2c_1 n + c_1 + c_2 n \qquad (2c_1 > c_2, \quad n \geqslant c_1/(2c_1 - c_2)) \\ &\leqslant c_1 n^2 = \Theta(n^2) \end{aligned}$$

□

Przypadek optymistyczny. W przypadku optymistycznym, tj. kiedy pierwszy podział jest $\lfloor n/2 \rfloor$, a drugi $\lceil n/2 \rceil - 1$ równanie ma postać:

$$T(n) = 2T(n/2) + \Theta(n),$$

a jej rozwiązaniem jest $\Theta(n \log n)$.

Dowód. Z twierdzenia o rekurencji uniwersalnej $T(n) = \Theta(n \log n)$ □

Przypadek oczekiwany. Aby określić asymptotyczną złożoność w przypadku oczekiwanym należy lekko zmodyfikować algorytm w taki sposób, żeby $x \leftarrow A[r]$ zamiast ostatniej liczby dostawał losową. W tym celu zamieniamy $A[r]$ z pewną wylosowaną $A[i]$ na początku procedury PARTITION. Można udowodnić, że randomizowany algorytm quicksort ma oczekiwaną złożoność $O(n \log n)$, gdy sortowane wartości są różne.

6 Rzędy wielkości funkcji

6.1 Notacja asympotyczna

Powiemy, że funkcja $g(n)$ jest **asymptotycznie dokładnym oszacowaniem** dla $f(n)$, co zapisujemy jako $f(n) = \Theta(g(n))$, gdzie:

$$\Theta(g(n)) = \{f(n) : \exists_{c_1,c_2,n_0>0} \forall_{n \geqslant n_0} \, (0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n))\}$$

Zamiast pisać $f(n) \in \Theta(g(n))$ (co jest formalnie poprawne), często pisze się $f(n) = \Theta(g(n))$. Jest to wygodne, ale należy pamiętać, że tracimy wtedy przemienność. Często pisze się po prostu $O(n)$ w tym znaczeniu, choć jest to nieprecyzyjne.

Przykład. Udowodnijmy, że $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Potrzebujemy takich dodatnich stałych c_1, c_2, n_0, że:

$$c_1 n^2 \leqslant \frac{1}{2} n^2 - 3n \leqslant c_2 n^2$$

dla każdego $n \geqslant n_0$. Podzielmy przez n^2:

$$c_1 \leqslant \frac{1}{2} - \frac{3}{n} \leqslant c_2$$

Prawa nierówność jest prawdziwa dla $n \geqslant 1$, gdy dobierzemy $c_2 \geqslant 1/2$. Lewa strona się zgadza dla $n \geqslant 7$ oraz $c_1 \leqslant 1/14$.

Tabela 1: Popularne rzędy wielkości wraz z nazewnictwem

Rząd	Opis
$\Theta(0) = 0$	brak potrzebnych operacji
$\Theta(1)$	stała
$\Theta(\log n)$	logarytmiczny
$\Theta((\log n)^c)$	polilogarytmiczny
$\Theta(n)$	liniowy
$\Theta(n \log n)$	log-liniowy (quasi-liniowy)
$\Theta(n^2)$	kwadratowy
$\Theta(n^c)$	wielomianowy
$\Theta(c^n)$	wykładniczy
$\Theta(n!)$	silnia
$\Theta(n^n)$	...

Rysunek 2: Wizualizacja różnych klas złożoności funkcji. Źródło: Stack Overflow

Powiemy, że funkcja $g(n)$ jest **asymptotycznym ograniczeniem górnym** dla $f(n)$, co zapisujemy jako $f(n) = O(g(n))$, gdzie:

$$O(g(n)) = \{f(n) : \exists_{c,n_0>0} \forall_{n \geqslant n_0} \, (0 \leqslant f(n) \leqslant cg(n))\}$$

Zauważmy, że $f(n) = \Theta(g(n)) \implies f(n) = O(g(n))$. Np. każda funkcja liniowa zawiera się w $O(n^2)$. Górne oszacowanie to niejako szacowanie przypadku pesymistycznego – algorytm zadziała nie gorzej niż $cg(n)$.

Powiemy, że funkcja $g(n)$ jest **asymptotycznym ograniczeniem dolnym** dla $f(n)$, co zapisujemy jako $f(n) = \Omega(g(n))$, gdzie:

$$\Omega(g(n)) = \{f(n) : \exists_{c,n_0>0} \forall_{n \geqslant n_0} \, (0 \leqslant cg(n) \leqslant f(n))\}$$

Dolne oszacowanie to niejako oszacowanie przypadku optymistycznego – algorytm nie zadziała lepiej niż $cg(n)$. Zauważmy, że $f(n) = \Theta(g(n)) \implies f(n) = \Omega(g(n))$. Co więcej, $f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \land f(n) = O(g(n))$.

Powiemy, że funkcja $g(n)$ jest **asymptotycznie niedokładnym ograniczeniem górnym** dla $f(n)$, co zapisujemy jako $f(n) = o(g(n))$, gdzie::

$$o(g(n)) = \{f(n) : \forall_c \; \exists_{n_0>0} \; \forall_{n \geqslant n_0} \, (0 \leqslant f(n) < cg(n))\}$$

Główna różnica polega na tym, że oszacowanie zachodzi dla każdej stałej $c > 0$ (zamiast dla pewnej stałej c). Inna definicja to:

$$o(g(n)) = f(n) \implies \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Przykład. $2n = o(n^2)$, ale $2n^2 \neq o(n^2)$

Powiemy, że funkcja $g(n)$ jest **asymptotycznie niedokładnym ograniczeniem dolnym** dla $f(n)$, co zapisujemy jako $f(n) = \omega(g(n))$, gdzie:

$$\omega(g(n)) = \{f(n) : \forall_c \; \exists_{n_0>0} \; \forall_{n \geqslant n_0} \, (0 \leqslant cg(n) < f(n))\}$$

Główna różnica polega na tym, że oszacowanie zachodzi dla każdej stałej $c > 0$ (zamiast dla pewnej stałej c). Inna definicja to:

$$\omega(g(n)) = f(n) \implies \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Przykład. $\frac{n^2}{2} = \omega(n)$, ale $\frac{n^2}{2} \neq \omega(n^2)$.

v

Tabela 2: Czas potrzebny na wykonanie funkcji o różnej złożoności. Źródło: DZone

n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$n^2 + 100n$	$O(n3)$
1	1	1	1	1	1	101	1
2	1	1	2	2	4	204	8
4	1	2	4	8	16	416	64
8	1	3	8	24	64	864	512
16	1	4	16	64	256	1,856	4,096
1,024	1	10	1,024	10,240	1,048,576	1,150,976	1,073,741,824

Tabela 3: Złożoność obliczeniowa podstawowych struktur danych

struktura	Access(S, n)		Search(S, k)		Insert(S, x)		Delete(S, x)	
	średnia	pes.	średnia	pes.	średnia	pes.	średnia	pes.
Tablica	$\Theta(1)$	$O(1)$	$\Theta(n)$	$O(n)$	–	–	–	–
Lista (jedno-/dwukierunkowa)	$\Theta(n)$	$O(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$	$O(1)$	$\Theta(1)$	$O(1)$
Stos	$\Theta(n)$	$O(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$	$O(1)$	$\Theta(1)$	$O(1)$
Kolejka	$\Theta(n)$	$O(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$	$O(1)$	$\Theta(1)$	$O(1)$
Drzewo BST	$\Theta(\log n)$	$O(n)$	$\Theta(\log n)$	$O(n)$	$\Theta(\log n)$	$O(n)$	$\Theta(\log n)$	$O(n)$

6.2 Standardowe rzędy wielkości funkcji

7 Struktury danych

Podstawowe operacje na kolekcjach: Search(S, x) – wyszukiwanie, Insert(S, x) – dodanie nowego elementu, Delete(S, x) – usunięcie, Minimum(S), Maximum(S), Successor(S, x) – następny element w liniowo uporządkowanym zbiorze, Predecessor(S, x) – poprzedni element w liniowo uporządkowanym zbiorze.

7.1 Stos

Stos jest *LIFO*. Operacje push, pop – $\Theta(1)$. Dostęp i wyszukiwanie są $\Theta(n)$ w przypadku średnim. Jeżeli stos jest większy niż n to powiemy, że jest przepełniony. Stos przechowuje wskaźnik *top* na *czubek* stosu.

```
stack = [3, 4, 5]
stack.append(6). # jak push
stack.append(7)
stack
# [3, 4, 5, 6, 7]
stack.pop()
# 7
stack
# [3, 4, 5, 6]
stack.pop()
# 6
stack.pop()
# 5
stack
# [3, 4]
```

7.2 Kolejka

Kolejka jest *FIFO*. Dodawanie do kolejki to *enqueue* i jest $\Theta(1)$, a usuwanie to *dequeue* i również jest $\Theta(1)$. Dostęp i wyszukiwanie są $\Theta(n)$ w przypadku średnim. Kolejka ma dwa wskaźniki – tail i head.

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
```

```
queue.append("Graham")          # Graham arrives
queue.popleft()                 # The first to arrive now leaves
# 'Eric'
queue.popleft()                 # The second to arrive now leaves
# 'John'
queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

7.3 Lista z dowiązaniami

Lista składa z elementów listy, z czego jeden jest głową (jeżeli nie ma poprzednika), a jeden ogonem (jeżeli nie ma następnika). Lista przechowuje wartość i wskaźnik na następnik (oraz na poprzednik, jeżeli jest podwójnie dowiązana). Dostęp i wyszukiwanie jest $\Theta(n)$, a wstawianie i usuwanie jest $\Theta(1)$.

7.4 Drzewa

Drzewo to niezorientowany graf spójny i acykliczny. Zbiór drzew to las, węzeł nie posiadający *rodzica* to korzeń, a węzeł nie posiadający *dzieci* to liść.

7.5 Drzewa BST

Dla drzewa BST dostęp, wyszukiwanie, wstawanie i usuwanie jest $\Theta(\log n)$, a w najgorszym przypadku mamy $\Theta(n)$. Zasada tworzenia drzew: jeżeli x jest węzłem drzewa, a y jest jego lewym dzieckiem, to $y.key \leqslant x.key$. Jeżeli y jest jego prawym dzieckiem, to $y.key \geqslant x.key$. Klasa drzewa i elementu drzewa:

```
class Tree():                         # drzewo
  def __init__(self, root=None):      # konstruktor drzewa, w pythonie dla klas pierwszy argument jest pomi
    self.root = root

class Node:                           # węzeł drzewa
  def __init__(self, k, left, right, parent=None): # konstruktur (klucz, lewe dziecko, prawe dziecko, rodzi
    self.parent = parent              # rodzic
    self.key = k                      # wartość przechowywana w tym węźle
    self.left = left                  # wskaźnik na lewe poddrzewo
    self.right = right                # wskaźnik na prawe poddrewo
  def __str__(self):                  # metoda zamieniająca drzewo na łańcuch znaków (przydatna przy printo
    return "Jestem węzłem drzewa i przechowuję wartość {}".format(self.key)
```

Tworzenie drzewa:

```
# drzewo z CLRS, str. 288, rys. 12.1a.

n_two = Node(2, None, None)
n_five = Node(5, None, None)
n_five_2 = Node(5, n_two, n_five)
n_eight = Node(8, None, None)
n_seven = Node(7, None, n_eight)
n_six = Node(6, n_five_2, n_seven)

n_six.parent = None
n_five_2.parent = n_six
n_five.parent = n_five_2
n_two.parent = n_five_2
n_seven.parent = n_six
n_eight.parent = n_seven

tree = Tree(root=n_six)
```

Przechodzenie drzewa inorder, preorder, postorder:

```python
def preorderTreeWalk(x):
  if x is not None:
    print(x.key)
    preorderTreeWalk(x.left)
    preorderTreeWalk(x.right)

def inorderTreeWalk(x):
  if x is not None:
    inorderTreeWalk(x.left)
    print(x.key)
    inorderTreeWalk(x.right)

def postorderTreeWalk(x):
  if x is not None:
    postorderTreeWalk(x.left)
    postorderTreeWalk(x.right)
    print(x.key)
```

Rysunek 3: Przechodzenie po drzewie. Źródło: `http://geeksforgeeks.com`

Szukanie elementu po wartości:

```python
def treeSearch(x, k):
  if x is None or k == x.key:
    return x
  if k < x.key:
    return treeSearch(x.left, k)
  else:
    return treeSearch(x.right, k)

def treeSearchIterative(x, k):
  while x is not None and k != x.key:
    if k < x.key:
      x = x.left
    else:
      x = x.right
  return x

nodeWithSeven = treeSearch(tree.root, 7)
print(nodeWithSeven)
```

Minimum i maksimum:

```python
def treeMinimum(x):
  while x.left != None:
    x = x.left
  return x

def treeMaximum(x):
  while x.right != None:
    x = x.right
  return x
```

Następny element do odwiedzenia w porządku inorder:

```python
def treeSuccessor(x):
  if x.right is not None:
```

```
    return treeMinimum(x.right)
  y = x.parent
  while y is not None and x == y.right:
    x = y
    y = y.parent
  return y
print(treeSuccessor(tree.root)) # inorder
```

Wstawianie do drzewa

```
def treeInsert(T, z):
  y = None
  x = T.root
  while x is not None:
    y = x
    if z.key < x.key:
      x = x.left
    else:
      x = x.right
  z.parent = y
  if y is None:
    T.root = z
  elif z.key < y.key:
    y.left = z
  else:
    y.right = z

print('Before insertion')
inorderTreeWalk(tree.root)
treeInsert(tree, Node(4, None, None))
print('After insertion')
inorderTreeWalk(tree.root)
```

Usuwanie z drzewa

```
def transplant(T, u, v):
  if u.parent is None:
    T.root = v
  elif u == u.parent.left:
    u.parent.left = v
  else:
    u.parent.right = v
  if v is not None:
    v.parent = u.parent

def treeDelete(T, z):
  if z.left is None:
    transplant(T, z, z.right)
  elif z.right is None:
    transplant(T, y, y.left)
  else:
    y = treeMinimum(z.right)
    if y.parent != z:
      transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
    transplant(T, z, y)
    y.left = z.left
    y.left.parent = y
```

```python

print('Before deletion:')
inorderTreeWalk(tree.root)
nodeToBeRemoved = treeSearch(tree.root, 4)
treeDelete(tree.root, nodeToBeRemoved)
print('After deletion:')
inorderTreeWalk(tree.root)
```

7.6 Kodowanie Huffmana

Algorytm zachłanny (dokonujący wyboru, który w danej chwili jest najkorzystniejszy) na obliczanie kodu prefiksowego, tj. takiego, gdzie żadne słowo kodowe nie jest prefiksem innego słowa kodowego.

Tabela 4: Kodowanie znaków. Źródło: Cormen et al.

	a	b	c	d	e	f
Częstość (tys.)	45	13	12	16	9	5
Słowo kodowe o stałej długości	000	001	010	011	100	101
Słowo kodowe o zmiennej długości	0	101	100	111	1101	1100

Aby przechować plik stałą szerokością:

$$3 \cdot (45 + 13 + 12 + 16 + 9 + 5) \cdot 1000 = 3 \cdot 100 \cdot 1000 = 300000$$

W przypadku szerokości zmiennej:

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224000$$

Tym samym oszczędzamy ok. 25%!

Algorithm 8 Kodowanie Huffmana

```
function HUFFMAN(C)                          // O(n log n)
    n ← |C|
    Q ← PRIORITYQUEUE(C)                     // Kolejka priorytetowa
    for i ← 1 to n − 1
        z ← nowy węzeł
        z.left ← x ← EXTRACTMIN(Q)
        z.right ← y ← EXTRACTMIN(Q)
        z.freq ← x.freq + y.freq
        INSERT(Q, z)
    return EXTRACTMIN(Q)
```

Przykład: `https://people.ok.ubc.ca/ylucet/DS/Huffman.html`

8 Wstęp do teorii grafów

8.1 Reprezentacja grafów

Graf G definiujemy jako $G = (V, E)$, gdzie V to zbiór krawędzi, a E – zbiór wierzchołków.

Listy sąsiedztwa. Jest to tablica zawierająca $|V|$ list, a suma długości wszystkich list to $|E|$. Oczywiście można przechowywać informacje o np. wagach.

Macierz sąsiedztwa. Wymaga $\Theta(|V|^2)$

8.2 Przechodzenie przez graf

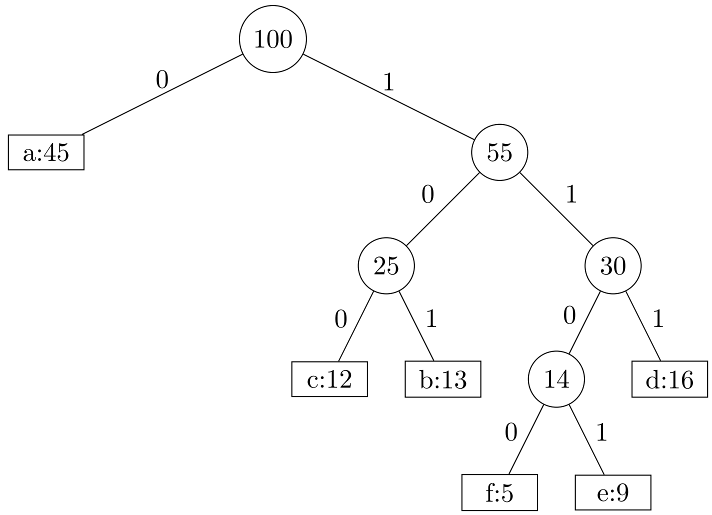
Przeszukiwanie wszerz

Algorithm 9 Przeszukiwanie wszerz

```
function BFS(G, s)                                   //O(|V| + |E|), s – start
    for  each u ∈ G.V − {s}
        u.color ← biały
        u.d ← ∞
        u.π ← NIL
    s.color ← szary
    s.d ← 0
    s.π ← NIL
    Q ← ∅
    Enqueue(Q, s)
    while Q ≠ ∅
        u ← Dequeue(Q)
        for each v ∈ G.Adj[u]
            if v.color = biały then
                v.color ← szary
                v.d ← u.d + 1
                v.π ← u
                Enqueue(Q, v)
        u.color ← czarny
```

Algorithm 10 Przeszukiwanie wgłąb

```
function DFS(G)                                      //O(|V| + |E|)
    for  each u ∈ G.V
        u.color ← biały
        u.π ← NIL
    time ← 0
    for  each u ∈ G.V
        if u.color = biały then
            DFS-Visit(G, u)
function DFS-Visit(G, u)
    time ← time + 1
    u.d ← time
    u.color ← szary
    for  each u ∈ G.Adj[u]
        if v.color = biały then
            v.π ← u
            DFS-Visit(G, v)
    u.color ← czarny
    time ← time + 1
    u.f ← time
```



Rysunek 4: Drzewo odpowiadające kodom

Rysunek 5: Reprezentacja grafu nieskierowanego jako lista sąsiedztwa oraz macierz sąsiedztwa. Źródło: Cormen et al.

Przeszukiwanie wgłąb

8.3 Minimalne drzewo rozpinające

Problem minimalnego drzewa rozpinającego to problem znalezienia takiego zbioru $T \subseteq E$, dla którego $w(T) = \sum_{(u,v)\in T} w(u,v)$ jest najmniejsze (gdzie $w(u,v)$ to waga krawędzi między u i v). Przykład: położenie kabla pod systemem drogowym.

Algorytm Kruskala. Algorytm zachłanny – w każdym kroku dodajemy krawędź o najmniejszej wadze.

Algorytm Prima.

8.4 Algorytm Dijkstry

Algorytm Dijkstry służy do rozwiązywania problemu najkrótszych ścieżek z jednym źródłem w ważonym grafie skierowanym z nieujemnymi wagami.

9 Wyszukiwanie wzorca w tekście

(nie do zrobienia w 15h)

9.1 Wyszukiwanie naiwne

(nie do zrobienia w 15h)

9.2 Algorytm Aho-Corasik

(nie do zrobienia w 15h)

Rysunek 6: Reprezentacja grafu skierowanego jako lista sąsiedztwa oraz macierz sąsiedztwa. Źródło: Cormen et al.

Rysunek 7: BFS. Źródło: Cormen et al.

Rysunek 8: DFS (na wierzchołkach czas odwiedzenia/czas przetworzenia, krawędzie niedrzewowe: B – back, C – cross, F – forward. Źródło: Cormen et al.

Algorithm 11 Algorytm Kruskala

function MST-KRUSKAL(G, w) $//O(|E| \log |V|)$
  $A \leftarrow \varnothing$
  for each $v \in G.V$
    MAKE-SET(v) // jednoelementowe drzewa
  posortuj niemalejąco $G.E$ względem wag w
  for each $(u, v) \in G.E$ w kolejności niemalejących wag
    if FIND-SET$(u) \neq$ FIND-SET(v) **then**
      $A \leftarrow A \cup \{(u, v)\}$
      UNION(u, v)

Rysunek 9: Algorytm Kruskala. Źródło: Cormen et al.

Rysunek 10: Algorytm Kruskala (c.d.). Źródło: Cormen et al.

Algorithm 12 Algorytm Prima

function MST-PRIM(G, w, r) // $O(|E| \log |V|)$
  for each $u \in G.V$
    $u.key \leftarrow \infty$
    $u.\pi \leftarrow$ `NIL`
  $r.key \leftarrow 0$
  $Q \leftarrow G.V$
  while $Q \neq \varnothing$
    $u \leftarrow$ EXTRACT-MIN(Q)
    for each $v \in G.Adj[u]$
      if $v \in Q \wedge w(u, v) < v.key$ **then**
        $v.\pi = u$
        $v.key = w(u, v)$

Rysunek 11: Algorytm Prima. Źródło: Cormen et al.

Rysunek 12: Algorytm Dijkstry. Źródło: Cormen et al.

Algorithm 13 Algorytm Dijkstry

```
function INITIALIZE-SINGLE-SOURCE(G, s)                 // Θ(|V|)
    for each v ∈ G.V
        v.d ← ∞
        v.π ← NIL
    s.d ← 0
function RELAX(u, v, w)                                 // Θ(1)
    if v.d > u.d + w(u, v) then
        v.d ← u.d + w(u, v)
        v.π ← u
function DIJKSTRA(G, u)                                 // naiwnie O(|V|^2),
    INITIALIZE-SINGLE-SOURCE(G, s)                      // da się zejść do O(|V| log |V| + |E|)
    S ← ∅                                               // ale wymaga to kopca Fibonacciego
    Q ← G.V
    while Q ≠ ∅
        u ← EXTRACT-MIN(Q)
        S ← S ∪ {u}
        for each v ∈ G.Adj[u]
            RELAX(u, v, w)
```

10 NP-zupełność. Czy P=NP?

?