Akademia Górniczo-Hutnicza
Wydział Elektrotechniki, Automatyki,
Informatyki i Inżynierii Biomedycznej

Adrian Horzyk

WSTĘP DO INFORMATYKI

Złożoność obliczeniowa, efektywność i algorytmy sortowania

EFEKTYWNOŚĆ

Efektywność wykonywania różnych operacji na danych w informatyce jest najczęściej związana z zastosowaniem odpowiednich struktur danych, na których dodatkowo można wykonywać operacje zdecydowanie szybciej, gdy przechowywane dane w nich są uporządkowane.

Wobec tego duże znaczenie ma stosowanie odpowiednio szybkich algorytmów sortowania dopasowanych do rodzaju i rozmiaru sortowanych danych.

Algorytmy sortowania różnią się kilkoma właściwościami:

- mogą sortować dane **w miejscu** lub **nie w miejscu**
- mogą być **stabilne** lub **niestabilne**
- mogą wykonywać sortowanie z różną złożonością obliczeniową

Sortowanie w miejscu nie wymaga wykorzystania dodatkowej pamięci zależnej od ilości sortowanych danych, lecz tylko pewną stałą jej ilość.

Sortowanie stabilne zapewnia, iż względna kolejność sortowanych kluczy o tych samych wartościach będzie zachowana w ciągu posortowanym, zaś w przypadku **sortowania niestabilnego**, takiej gwarancji nie ma.

ZŁOŻONOŚĆ OBLICZENIOWA

Złożoność obliczeniowa i poprawność semantyczna algorytmów badana jest przez **algorytmikę** – dział informatyki zajmujący się **analizą algorytmów**.

W trakcie **analizy algorytmów** próbujemy udzielić odpowiedzi na pytania:

- Czy możliwe jest rozwiązanie zadania w dostępnym czasie i pamięci?
- Który ze znanych algorytmów należy zastosować w danych okolicznościach?
- Czy istnieje lepszy, szybszy lub dokładniejszy algorytm od rozważanego?
- Jak udowodnić, że zastosowany algorytm poprawnie rozwiąże zadanie?
- Czy możliwe jest zrównoleglenie algorytmu i wykorzystanie większej ilości rdzeni obliczeniowych?

Złożoność obliczeniowa badana jest w dwóch podstawowych aspektach:

- **czasu** – złożoność czasowa
- **pamięci** – złożoność pamięciowa

Celem analizy algorytmów jest opracowanie metod formalnych i analiz umożliwiających zbudowanie optymalnych, efektywnych i poprawnych semantycznie algorytmów wykonalnych na dostępnych zasobach.

RODZAJE ZŁOŻONOŚCI OBLICZENIOWEJ

Złożoność czasowa to ilość czasu potrzebnego na wykonanie zadania, wyrażona jako funkcja ilości danych, mierzona ilością elementarnych kroków, tj. instrukcji warunkowych, przypisania i operacji arytmetycznych.

Złożoność pamięciowa to ilość dodatkowej pamięci potrzebnej do wykonania zadania, wyrażona jako funkcja ilości danych.

Rozróżniamy kilka **rodzajów złożoności**:

- **Pesymistyczna** – określa ilość zasobów (czasu lub dodatkowej pamięci) potrzebnych do wykonania algorytmu przy założeniu wystąpienia „złośliwych” lub najgorszych danych. Oznaczamy ją $O(f(n))$, gdzie $f(n)$ jest funkcją ilości danych n.
- **Praktyczna** – określa dokładną liczbę elementarnych kroków potrzebnych do wykonania algorytmu rozwiązującego określone zadanie.
- **Oczekiwana** – określa ilość zasobów potrzebnych do wykonania algorytmu przy założeniu wystąpienia „typowych” lub oczekiwanych danych.
- **Optymistyczna** $\Omega(f(n))$ – określa ilość zasobów potrzebnych do wykonania algorytmu przy założeniu wystąpienia „najlepszych” danych.

Złożoność wykonywania różnych algorytmów możemy szacować:

- **od dołu** stwierdzając, iż złożoność obliczeniowa jest nie mniejsza niż pewna klasa funkcji $\Omega(f(n))$
- **asymptotycznie**, szacując dokładnie złożoność obliczeniową poprzez pewną klasę funkcji $\Theta(f(n))$
- **od góry** ograniczając złożoność obliczeniową poprzez pewną klasę funkcji $O(f(n))$

KLASY ZŁOŻONOŚCI OBLICZENIOWEJ

Klasy złożoności obliczeniowej określają pewne typowe funkcje zależne od ilości danych, które stosowane są wyznaczenia złożoności obliczeniowej danego zadania. Do najbardziej typowych zaliczamy złożoność:

----------PROBLEMY ŁATWE----------

Stałą $\Theta(1)$ – gdy czas wykonania algorytmu jest niezależny od rozmiaru danych

Logarytmiczną $\Theta(\log n)$ – gdy czas wykonania zależny jest od logarytmu rozmiaru danych

Liniową $\Theta(n)$ – gdy czas wykonania zależny jest liniowo od rozmiaru danych

Logarytmiczno-liniową $\Theta(n \log n)$ – gdy czas wykonania algorytmu jest logarytmiczno-linowy

Kwadratową $\Theta(n^2)$ – gdy czas wykonania zależny jest od kwadratu rozmiaru danych

Sześcienną $\Theta(n^3)$ – gdy czas wykonania zależny jest od sześcianu rozmiaru danych

Wielomianową $\Theta(n^k + n^{k-1} + \ldots + n)$ – gdy czas wykonania zależny jest od wielomianu rozmiaru danych

---------- PROBLEMY TRUDNE ----------

Wykładniczą $\Theta(2^n)$ – gdy czas wykonania rośnie wykładniczo z rozmiarem danych

Silnia $\Theta(n!)$ – gdy czas wykonania wyrażona jest za pomocą silni rozmiaru danych, czyli związana jest z przeszukiwaniem wszystkich (kombinacji, wariacji czy permutacji danych)

PRZYKŁADY KLAS ZŁOŻONOŚCI OBLICZENIOWEJ

----------PROBLEMY ŁATWE----------

Stała Θ(1): dowolna elementarna instrukcja przypisania, pojedynczej operacji arytmetycznej lub logicznej, porównania, np. x = 2, x += 1, 2+3, if x<y, z and w

Stała O(1): wyszukiwanie w tablicy haszującej, jeśli możliwe jest określenie funkcji haszującej w taki sposób, aby dostęp do poszczególnych elementów był stały.

Logarytmiczno-logarytmiczna Θ(log log n): algorytm wyszukiwania interpolowanego przy spełnieniu warunków jego zastosowania

Logarytmiczna Θ(log n): algorytm wyszukiwania połówkowego, bisekcji, przechodzenia od korzenia do liścia lub w drugą stronę w drzewie wyważonym zawierającym n węzłów, np. w drzewie BST, kopcu zupełnym, B-drzewach.

Liniowa Θ(n): wyszukiwanie liniowe w n-elementowych ciągach nieposortowanych, sortowanie przez zliczanie i pozycyjne, wyszukiwanie wzorców metodą Knutha-Morrisa-Pratta w ciągach tekstów.

Logarytmiczno-liniowa Θ(n log n): najefektywniejsze metody sortowania elementów polegające na porównaniach (kopcowe, przez łączenie, szybkie) ogólnego przeznaczenia.

Kwadratowa Θ(n^2): proste algorytmy sortowania przez wstawianie, wybieranie, bąbelkowe, szukanie najkrótszej ścieżki w grafie metodą ….

---------- PROBLEMY TRUDNE ----------

Wykładnicza Θ(2^n) – gdy problem dzieli się w każdym kroku na dwa lub więcej składowych operujących na całym zbiorze danych, np. niektóre algorytmy wykorzystujące rekurencję, np. ciąg Fibonacciego.

Silnia Θ(n!) – różne problemy kombinatoryczne polegające na przeszukiwaniu wszystkich kombinacji, wariacji czy permutacji n danych, np. niektóre algorytmy operujące na grafach.

DZIAŁANIA NA KLASACH ZŁOŻONOŚCI OBLICZENIOWEJ

Próbując określić klasę złożoności obliczeniowej dla całego algorytmu lub programu analizujemy jego części wyznaczając ich klasy złożoności obliczeniowej, a następnie korzystając z poniższych operacji wyznaczamy ogólną złożoność:

$\Theta(c) = \Theta(1)$ – dowolna liczba operacji c niezależna od n to klasa stałej złożoności obliczeniowej.

$c \cdot \Theta(f(n)) = \Theta(f(n))$ – gdy c jest stałą niezależną od n.
Taki przypadek występuje np. w sytuacji, gdy wykonujemy pętlę obliczeniową pewną stałą ilość razy niezależną od wymiaru danych n, wtedy taka pętla nie zwiększa złożoności obliczeniowej. Ważne jest ustalenie niezależności c od n!

$\Theta(f(n)) + \Theta(g(n)) = \max(\Theta(f(n)), \Theta(g(n)))$ – a więc sumę złożoności obliczeniowych charakteryzuje ta o większej klasie złożoności obliczeniowej.
To najczęstszy przypadek, gdy obliczamy złożoność na podstawie kolejnych fragmentów kodu następujących po sobie w programie lub algorytmie. O złożoności decyduje fragment kodu o największej złożoności obliczeniowej.

$\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$ – w przypadku iloczynu złożoności obliczeniowych klasa będzie określona poprzez klasę iloczynu funkcji określających poszczególne klasy.
Mamy z tym do czynienia np. w przypadku pętli zagnieżdżonych lub podczas wywoływania funkcji / procedur / metod. Jeśli np. funkcja ma złożoność g(n) i jest wywoływana w programie głównym w pętli o złożoności f(n), wtedy złożoność tego fragmentu kodu, tj. pętli zawierającej wywołanie tej funkcji będzie miało złożoność $\Theta(f(n) \cdot g(n))$.

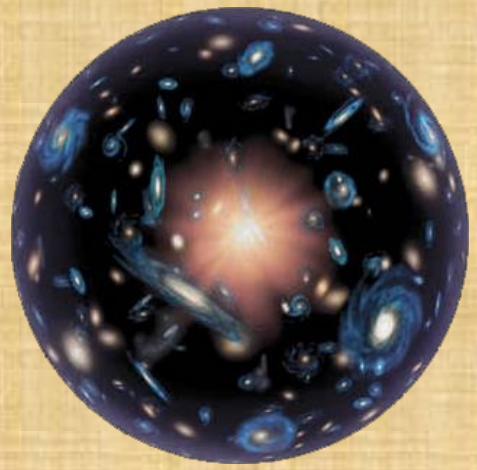
DLACZEGO ZŁOŻONOŚĆ JEST WAŻNA?

Problemy złożoności obliczeniowej bezpośrednio przekładają się na możliwość wykonania algorytmu na współczesnych maszynach obliczeniowych opartych na modelu maszyny Turinga, a ponadto istotnie wpływają na czas obliczeń:

Złożoność obliczeniowa względem rozmiaru danych wejściowych

Rozmiar danych:	10	20	50	100	200	1000
log n	3,32 ns	4,23 ns	5,64 ns	6,64 ns	7,64 ns	9,97 ns
n	10 ns	20 ns	50 ns	100 ns	200 ns	1 µs
n log n	33,21 ns	86,44 ns	282,2 ns	664,4 ns	1,53 µs	9,97 µs
n^2	100 ns	400 ns	2,5 µs	10 µs	40 µs	1 ms
2^n	1 µs	1,05 ms	13 dni	$4 \cdot 10^{13}$ lat	$5{,}1 \cdot 10^{43}$ lat	$3{,}4 \cdot 10^{284}$ lat
n!	3,6 ms	77 lat	$9{,}6 \cdot 10^{44}$ lat	$3 \cdot 10^{141}$ lat	$2{,}5 \cdot 10^{358}$ lat	$1{,}27 \cdot 10^{2551}$ lat

gdzie wiek naszego wszechświata szacuje się na 13,7 mld lat.



Rozróżniamy więc dwie podstawowe i jedną dodatkową klasę algorytmów pod względem złożoności obliczeniowej:

Łatwe (P – polynomial) – czyli rozwiązywalne co najwyżej w czasie wielomianowym

Trudne (NP – non-polynomial) – wymagające czasu dłuższego niż wielomianowy, czyli np. wykładniczy lub silni

NP zupełne (NPC – NP complete) – nie udowodniono, iż nie posiadają rozwiązania wielomianowego.

PRZYKŁAD SORTOWANIA OBIEKTÓW

Najpierw definiujemy klasę zawierającą obiekty składające się z pewnej liczby atrybutów oraz ich nazwy, wczytując je z pliku CSV i zapisując w postaci listy list:

```
3       @author: Adrian Horzyk
4       """
5       import csv
6       import math
7       import time
8       import datetime
9       from SortingAlgorithms import HeapSort, QuickSort, MergeSort
10
11      class Table():
12          def __init__(self, filepath):
13              self.objects = [] # Create empty list of attribute values
14              self.names = [] # Create empty list of attribute names
15              id = -1
16              with open(filepath) as f:
17                  for row in [content for content in csv.reader(f, delimiter='\n')]:
18                      if id == -1:  # Read attribute names from the first row and store them in the list of names
19                          i = 0
20                          for object in row:
21                              self.names = object.split(';')
22                      else:   # Read attribute values from the following rows and store them in the list of objects
23                          for object in row:
24                              attributes = object.split(';')
25                          self.objects.append(attributes)
26                      id += 1
27              self.noObjects = id
```

Następnie przeprowadzimy pomiar szybkości działania różnych algorytmów sortowania, porządkujących te obiekty po kolei według wszystkich ich atrybutów.

ZBIÓR OBIEKTÓW DO POSORTOWANIA

ATRYBUTY OBIEKTÓW

OBIEKT

	leaf-length	leaf-width	petal-length	petal-width	class
1	leaf-length;leaf-width;petal-length;petal-width;class				
2	5.10	3.50	1.40	0.20	Iris-setosa
3	4.90	3.00	1.40	0.20	Iris-setosa
4	4.70	3.20	1.30	0.20	Iris-setosa
5	4.60	3.10	1.50	0.20	Iris-setosa
6	5.00	3.60	1.40	0.20	Iris-setosa
7	5.40	3.90	1.70	0.40	Iris-setosa
8	4.60	3.40	1.40	0.30	Iris-setosa
9	5.00	3.40	1.50	0.20	Iris-setosa
10	4.40	2.90	1.40	0.20	Iris-setosa
11	4.90	3.10	1.50	0.10	Iris-setosa
12	5.40	3.70	1.50	0.20	Iris-setosa
13	4.80	3.40	1.60	0.20	Iris-setosa
14	4.80	3.00	1.40	0.10	Iris-setosa
15	4.30	3.00	1.10	0.10	Iris-setosa
16	5.80	4.00	1.20	0.20	Iris-setosa
17	5.70	4.40	1.50	0.40	Iris-setosa
18	5.40	3.90	1.30	0.40	Iris-setosa
19	5.10	3.50	1.40	0.30	Iris-setosa
20	5.70	3.80	1.70	0.30	Iris-setosa
21	5.10	3.80	1.50	0.30	Iris-setosa
22	5.40	3.40	1.70	0.20	Iris-setosa
23	5.10	3.70	1.50	0.40	Iris-setosa
24	4.60	3.60	1.00	0.20	Iris-setosa
25	5.10	3.30	1.70	0.50	Iris-setosa
26	4.80	3.40	1.90	0.20	Iris-setosa
27	5.00	3.00	1.60	0.20	Iris-setosa
28	5.00	3.40	1.60	0.40	Iris-setosa
29	5.20	3.50	1.50	0.20	Iris-setosa
30	5.20	3.40	1.40	0.20	Iris-setosa
31	4.70	3.20	1.60	0.20	Iris-setosa
32	4.80	3.10	1.60	0.20	Iris-setosa
33	5.40	3.40	1.50	0.40	Iris-setosa
34	5.20	4.10	1.50	0.10	Iris-setosa
35	5.50	4.20	1.40	0.20	Iris-setosa
36	4.90	3.10	1.50	0.10	Iris-setosa
37	5.00	3.20	1.20	0.20	Iris-setosa
38	5.50	3.50	1.30	0.20	Iris-setosa
39	4.90	3.10	1.50	0.10	Iris-setosa
40	4.40	3.00	1.30	0.20	Iris-setosa
41	5.10	3.40	1.50	0.20	Iris-setosa
42	5.00	3.50	1.30	0.30	Iris-setosa
43	4.50	2.30	1.30	0.30	Iris-setosa
44	4.40	3.20	1.30	0.20	Iris-setosa
45	5.00	3.50	1.60	0.60	Iris-setosa
46	5.10	3.80	1.90	0.40	Iris-setosa
47	4.80	3.00	1.40	0.30	Iris-setosa
48	5.10	3.80	1.60	0.20	Iris-setosa
49	4.60	3.20	1.40	0.20	Iris-setosa
50	5.30	3.70	1.50	0.20	Iris-setosa
51	5.00	3.30	1.40	0.20	Iris-setosa
52	7.00	3.20	4.70	1.40	Iris-versicolor
53	6.40	3.20	4.50	1.50	Iris-versicolor
54	6.90	3.10	4.90	1.50	Iris-versicolor
55	5.50	2.30	4.00	1.30	Iris-versicolor
56	6.50	2.80	4.60	1.50	Iris-versicolor
57	5.70	2.80	4.50	1.30	Iris-versicolor
58	6.30	3.30	4.70	1.60	Iris-versicolor
59	4.90	2.40	3.30	1.00	Iris-versicolor
60	6.60	2.90	4.60	1.30	Iris-versicolor
61	5.20	2.70	3.90	1.40	Iris-versicolor
62	5.00	2.00	3.50	1.00	Iris-versicolor
63	5.90	3.00	4.20	1.50	Iris-versicolor
64	6.00	2.20	4.00	1.00	Iris-versicolor
65	6.10	2.90	4.70	1.40	Iris-versicolor
66	5.60	2.90	3.60	1.30	Iris-versicolor
67	6.70	3.10	4.40	1.40	Iris-versicolor
68	5.60	3.00	4.50	1.50	Iris-versicolor
69	5.80	2.70	4.10	1.00	Iris-versicolor
70	6.20	2.20	4.50	1.50	Iris-versicolor
71	5.60	2.50	3.90	1.10	Iris-versicolor
72	5.90	3.20	4.80	1.80	Iris-versicolor
73	6.10	2.80	4.00	1.30	Iris-versicolor
74	6.30	2.50	4.90	1.50	Iris-versicolor
75	6.10	2.80	4.70	1.20	Iris-versicolor
76	6.40	2.90	4.30	1.30	Iris-versicolor
77	6.60	3.00	4.40	1.40	Iris-versicolor
78	6.80	2.80	4.80	1.40	Iris-versicolor
79	6.70	3.00	5.00	1.70	Iris-versicolor
80	6.00	2.90	4.50	1.50	Iris-versicolor
81	5.70	2.60	3.50	1.00	Iris-versicolor
82	5.50	2.40	3.80	1.10	Iris-versicolor
83	5.50	2.40	3.70	1.00	Iris-versicolor
84	5.80	2.70	3.90	1.20	Iris-versicolor
85	6.00	2.70	5.10	1.60	Iris-versicolor
86	5.40	3.00	4.50	1.50	Iris-versicolor
87	6.00	3.40	4.50	1.60	Iris-versicolor
88	6.70	3.10	4.70	1.50	Iris-versicolor
89	6.30	2.30	4.40	1.30	Iris-versicolor
90	5.60	3.00	4.10	1.30	Iris-versicolor
91	5.50	2.50	4.00	1.30	Iris-versicolor
92	5.50	2.60	4.40	1.20	Iris-versicolor
93	6.10	3.00	4.60	1.40	Iris-versicolor
94	5.80	2.60	4.00	1.20	Iris-versicolor
95	5.00	2.30	3.30	1.00	Iris-versicolor
96	5.60	2.70	4.20	1.30	Iris-versicolor
97	5.70	3.00	4.20	1.20	Iris-versicolor
98	5.70	2.90	4.20	1.30	Iris-versicolor
99	6.20	2.90	4.30	1.30	Iris-versicolor
100	5.10	2.50	3.00	1.10	Iris-versicolor
101	5.70	2.80	4.10	1.30	Iris-versicolor
102	6.30	3.30	6.00	2.50	Iris-virginica
103	5.80	2.70	5.10	1.90	Iris-virginica
104	7.10	3.00	5.90	2.10	Iris-virginica
105	6.30	2.90	5.60	1.80	Iris-virginica
106	6.50	3.00	5.80	2.20	Iris-virginica
107	7.60	3.00	6.60	2.10	Iris-virginica
108	4.90	2.50	4.50	1.70	Iris-virginica
109	7.30	2.90	6.30	1.80	Iris-virginica
110	6.70	2.50	5.80	1.80	Iris-virginica
111	7.20	3.60	6.10	2.50	Iris-virginica
112	6.50	3.20	5.10	2.00	Iris-virginica
113	6.40	2.70	5.30	1.90	Iris-virginica
114	6.80	3.00	5.50	2.10	Iris-virginica
115	5.70	2.50	5.00	2.00	Iris-virginica
116	5.80	2.80	5.10	2.40	Iris-virginica
117	6.40	3.20	5.30	2.30	Iris-virginica
118	6.50	3.00	5.50	1.80	Iris-virginica
119	7.70	3.80	6.70	2.20	Iris-virginica
120	7.70	2.60	6.90	2.30	Iris-virginica
121	6.00	2.20	5.00	1.50	Iris-virginica
122	6.90	3.20	5.70	2.30	Iris-virginica
123	5.60	2.80	4.90	2.00	Iris-virginica
124	7.70	2.80	6.70	2.00	Iris-virginica
125	6.30	2.70	4.90	1.80	Iris-virginica
126	6.70	3.30	5.70	2.10	Iris-virginica
127	7.20	3.20	6.00	1.80	Iris-virginica
128	6.20	2.80	4.80	1.80	Iris-virginica
129	6.10	3.00	4.90	1.80	Iris-virginica
130	6.40	2.80	5.60	2.10	Iris-virginica
131	7.20	3.00	5.80	1.60	Iris-virginica
132	7.40	2.80	6.10	1.90	Iris-virginica
133	7.90	3.80	6.40	2.00	Iris-virginica
134	6.40	2.80	5.60	2.20	Iris-virginica
135	6.30	2.80	5.10	1.50	Iris-virginica
136	6.10	2.60	5.60	1.40	Iris-virginica
137	7.70	3.00	6.10	2.30	Iris-virginica
138	6.30	3.40	5.60	2.40	Iris-virginica
139	6.40	3.10	5.50	1.80	Iris-virginica
140	6.00	3.00	4.80	1.80	Iris-virginica
141	6.90	3.10	5.40	2.10	Iris-virginica
142	6.70	3.10	5.60	2.40	Iris-virginica
143	6.90	3.10	5.10	2.30	Iris-virginica
144	5.80	2.70	5.10	1.90	Iris-virginica
145	6.80	3.20	5.90	2.30	Iris-virginica
146	6.70	3.30	5.70	2.50	Iris-virginica
147	6.70	3.00	5.20	2.30	Iris-virginica
148	6.30	2.50	5.00	1.90	Iris-virginica
149	6.50	3.00	5.20	2.00	Iris-virginica
150	6.20	3.40	5.40	2.30	Iris-virginica
151	5.90	3.00	5.10	1.80	Iris-virginica

SPOSÓB DZIAŁANIA SORTOWANIA SZYBKIEGO

Sortowanie szybkie (quick sort) w podanym przedziale wyszukuje niemniejszy klucz z lewej i niewiększy klucz z prawej i dokonuje ich zamiany miejscami dopóki indeksy i, j się nie miną:

```
3   @author: Adrian Horzyk
4   """
5   def Swap(data, i, j):
6       data[i], data[j] = data[j], data[i]
7
8   def QuickSort(data, key, first, last):
9       mid = data[(last+first) // 2][key]
10      i = first
11      j = last
12      while i <= j:
13          while data[i][key]<mid:
14              i += 1
15          while data[j][key]>mid:
16              j -= 1
17          if i <= j:
18              Swap(data, i, j)
19              i += 1
20              j -= 1
21      if first<j:
22          QuickSort(data, key, first, j)
23      if i<last:
24          QuickSort(data, key, i, last)
```

[4]-[5]-[1]-[9]-[6]-[3]-[6]-[8]-[3]-[1]-[2]-[5]-[7]

(0-12) mid = 6

[[4]-[5]-[1]-[5]-[6]-[3]-[6]-[8]-[3]-[1]-[2]-[9]-[7]]

[[4]-[5]-[1]-[5]-[2]-[3]-[6]-[8]-[3]-[1]-[6]-[9]-[7]]

[[4]-[5]-[1]-[5]-[2]-[3]-[1]-[8]-[3]-[6]-[6]-[9]-[7]]

[[4]-[5]-[1]-[5]-[2]-[3]-[1]-[3]-[8]-[6]-[6]-[9]-[7]]

(0-7) mid = 5

[[4]-[5]-[1]-[5]-[2]-[3]-[1]-[3]]-[8]-[6]-[6]-[9]-[7]

[[4]-[3]-[1]-[5]-[2]-[3]-[1]-[5]]-[8]-[6]-[6]-[9]-[7]

[[4]-[3]-[1]-[1]-[2]-[3]-[5]-[5]]-[8]-[6]-[6]-[9]-[7]

[[4]-[3]-[1]-[1]-[2]-[3]-[5]-[5]]-[8]-[6]-[6]-[9]-[7]

SPOSÓB DZIAŁANIA SORTOWANIA SZYBKIEGO

[4]-[5]-[1]-[9]-[6]-[3]-[6]-[8]-[3]-[1]-[2]-[5]-[7]

1 (0-12) mid = 6

[[4]-[5]-[1]-[5]-[6]-[3]-[6]-[8]-[3]-[1]-[2]-[9]-[7]]

[[4]-[5]-[1]-[5]-[2]-[3]-[6]-[8]-[3]-[1]-[6]-[9]-[7]]

[[4]-[5]-[1]-[5]-[2]-[3]-[1]-[8]-[3]-[6]-[6]-[9]-[7]]

[[4]-[5]-[1]-[5]-[2]-[3]-[1]-[3]-[8]-[6]-[6]-[9]-[7]]

2 (0-7) mid = 5

[[4]-[5]-[1]-[5]-[2]-[3]-[1]-[3]]-[8]-[6]-[6]-[9]-[7]

[[4]-[3]-[1]-[5]-[2]-[3]-[1]-[5]]-[8]-[6]-[6]-[9]-[7]

[[4]-[3]-[1]-[1]-[2]-[3]-[5]-[5]]-[8]-[6]-[6]-[9]-[7]

[[4]-[3]-[1]-[1]-[2]-[3]-[5]-[5]]-[8]-[6]-[6]-[9]-[7]

7 (8-12) mid = 6

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[[8]-[6]-[6]-[9]-[7]]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[[6]-[6]-[8]-[9]-[7]]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[[6]-[6]-[8]-[9]-[7]]

3 (0-5) mid = 1

[[4]-[3]-[1]-[1]-[2]-[3]]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[[1]-[3]-[1]-[4]-[2]-[3]]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[[1]-[1]-[3]-[4]-[2]-[3]]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

(6-7) mid = 5

[1]-[1]-[2]-[3]-[3]-[4]-[[5]-[5]]-[8]-[6]-[6]-[9]-[7]

[1]-[1]-[2]-[3]-[3]-[4]-[[5]-[5]]-[8]-[6]-[6]-[9]-[7]

8 (10-12) mid = 9

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[[8]-[9]-[7]]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[[8]-[7]-[9]]

4 (0-1) mid = 1

[[1]-[1]]-[3]-[4]-[2]-[3]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[[1]-[1]]-[3]-[4]-[2]-[3]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

5 (2-5) mid = 4

[1]-[1]-[[3]-[4]-[2]-[3]]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[1]-[1]-[[3]-[3]-[2]-[4]]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[1]-[1]-[[3]-[3]-[2]-[4]]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

9 (10-11) mid = 8

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[[8]-[7]]-[9]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[[7]-[8]]-[9]

6 (2-4) mid = 3

[1]-[1]-[[3]-[3]-[2]]-[4]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[1]-[1]-[[2]-[3]-[3]]-[4]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[1]-[1]-[[2]-[3]-[3]]-[4]-[5]-[5]-[8]-[6]-[6]-[9]-[7]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

ANALIZA ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie szybkie (*quick sort*):

Ile operacji elementarnych?

Ile porównań?

Ile przypisań?

Niestabilne, w miejscu.

```
3       @author: Adrian Horzyk
4       """
5       def Swap(data, i, j):
6           data[i], data[j] = data[j], data[i]
7
8       def QuickSort(data, key, first, last):
9           mid = data[(last+first) // 2][key]
10          i = first
11          j = last
12          while i <= j:
13              while data[i][key]<mid:
14                  i += 1
15              while data[j][key]>mid:
16                  j -= 1
17              if i <= j:
18                  Swap(data, i, j)
19                  i += 1
20                  j -= 1
21          if first<j:
22              QuickSort(data, key, first, j)
23          if i<last:
24              QuickSort(data, key, i, last)
```

```
29      start_time = time.clock()
30      filepath = "resources/iris.csv"
31      table = Table(filepath)
32      end_time = time.clock()
33      elapsed_time_ReadTable = end_time - start_time
34      print("Table read in " + str(elapsed_time_ReadTable))
35
36      start_time = time.clock()
37      for key in range(0,len(table.names)):
38          print("QuickSort for key " + str(key))
39          QuickSort(table.objects, key, 0, len(table.objects)-1)
40          for id in range(0,len(table.objects),1):
41              print(table.objects[id])
42      end_time = time.clock()
43      elapsed_time_QuickSort = end_time - start_time
44      print("Table sorted using QuickSort in " + str(elapsed_time_QuickSort))
```

```
Table read in 0.00517706004764
QuickSort for key 0
['4,30'  '3,00', '1,10', '0,10', 'Iris-setosa']
['4,40'  '3,20', '1,30', '0,20', 'Iris-setosa']
['4,40'  '2,90', '1,40', '0,20', 'Iris-setosa']
['4,40'  '3,00', '1,30', '0,20', 'Iris-setosa']
['4,50'  '2,30', '1,30', '0,30', 'Iris-setosa']
['4,60'  '3,20', '1,40', '0,20', 'Iris-setosa']
['4,60'  '3,40', '1,40', '0,30', 'Iris-setosa']
['4,60'  '3,60', '1,00', '0,20', 'Iris-setosa']
['4,60'  '3,10', '1,50', '0,20', 'Iris-setosa']
['4,70'  '3,20', '1,60', '0,20', 'Iris-setosa']
['4,70'  '3,20', '1,30', '0,20', 'Iris-setosa']
['4,80'  '3,10', '1,60', '0,20', 'Iris-setosa']
['4,80'  '3,40', '1,90', '0,20', 'Iris-setosa']
['4,80'  '3,40', '1,60', '0,20', 'Iris-setosa']
['4,80'  '3,00', '1,40', '0,30', 'Iris-setosa']
['4,80'  '3,00', '1,40', '0,10', 'Iris-setosa']
['4,90'  '2,40', '3,30', '1,00', 'Iris-versicolor']
['4,90'  '3,10', '1,50', '0,10', 'Iris-setosa']
['4,90'  '3,10', '1,50', '0,10', 'Iris-setosa']
['4,90'  '2,50', '4,50', '1,70', 'Iris-virginica']
['4,90'  '3,00', '1,40', '0,20', 'Iris-setosa']
['4,90'  '3,10', '1,50', '0,10', 'Iris-setosa']
['5,00'  '3,40', '1,50', '0,20', 'Iris-setosa']
['5,00'  '3,50', '1,60', '0,60', 'Iris-setosa']
['5,00'  '3,60', '1,40', '0,20', 'Iris-setosa']
['5,00'  '2,00', '3,50', '1,00', 'Iris-versicolor']
['5,00'  '3,20', '1,20', '0,20', 'Iris-setosa']
['5,00'  '3,50', '1,30', '0,30', 'Iris-setosa']
['5,00'  '3,00', '1,60', '0,20', 'Iris-setosa']
['5,00'  '3,30', '1,40', '0,20', 'Iris-setosa']
['5,00'  '2,30', '3,30', '1,00', 'Iris-versicolor']
['5,00'  '3,40', '1,60', '0,40', 'Iris-setosa']
['5,10'  '2,50', '3,00', '1,10', 'Iris-versicolor']
['5,10'  '3,80', '1,60', '0,20', 'Iris-setosa']
['5,10'  '3,40', '1,50', '0,20', 'Iris-setosa']
['5,10'  '3,80', '1,90', '0,40', 'Iris-setosa']
['5,10'  '3,50', '1,40', '0,20', 'Iris-setosa']
['5,10'  '3,80', '1,50', '0,30', 'Iris-setosa']
['5,10'  '3,50', '1,40', '0,30', 'Iris-setosa']
['5,10'  '3,30', '1,70', '0,50', 'Iris-setosa']
['5,10'  '3,70', '1,50', '0,40', 'Iris-setosa']
['5,20'  '3,40', '1,40', '0,20', 'Iris-setosa']
['5,20'  '3,50', '1,50', '0,20', 'Iris-setosa']
['5,20'  '2,70', '3,90', '1,40', 'Iris-versicolor']
```

```
Table sorted using QuickSort in 0.0794245951649
```

PRZYKŁAD ANALIZY ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie szybkie n obiektów względem wszystkich k atrybutów:

```
29  start_time = time.clock()                                          Θ(1)
30  filepath = "resources/iris.csv"                                    Θ(1)
31  table = Table(filepath)                                            Θ(n · k)
32  end_time = time.clock()                                            Θ(1)
33  elapsed_time_ReadTable = end_time - start_time                     Θ(1)
34  print("Table read in " + str(elapsed_time_ReadTable))              Θ(1)
35                                                                     Θ(1)
36  start_time = time.clock()                                          Θ(1)
37  for key in range(0,len(table.names)):                              MAX
38      print("QuickSort for key " + str(key))                         Θ(k) · Θ(wnętrze pętli)
39      QuickSort(table.objects, key, 0, len(table.objects)-1)           Θ(1)
40      for id in range(0,len(table.objects),1):                         Θ(QuickSort)
41          print(table.objects[id])                                     Θ(wnętrze pętli)
                                                                          Θ(1) · Θ(k)
42  end_time = time.clock()                                            Θ(1)
43  elapsed_time_QuickSort = end_time - start_time                     Θ(1)
44  print("Table sorted using QuickSort in " + str(elapsed_time_QuickSort))  Θ(1)
```

Próbując wyznaczyć złożoność obliczeniową całego programu, wyznaczamy maksimum z następujących po sobie operacji na tym samym poziomie, wyznaczając złożoność operacji złożonych poprzez przemnożenie przez złożoność wnętrza pętli.

PRZYKŁAD ANALIZY ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie szybkie dla wybranego klucza (atrybutu):

```
3   @author: Adrian Horzyk
4   """
5   def Swap(data, i, j):
6       data[i], data[j] = data[j], data[i]
7
8   def QuickSort(data, key, first, last):
9       mid = data[(last+first) // 2][key]
10      i = first
11      j = last
12      while i <= j:
13          while data[i][key]<mid:
14              i += 1
15          while data[j][key]>mid:
16              j -= 1
17          if i <= j:
18              Swap(data, i, j)
19              i += 1
20              j -= 1
21      if first<j:
22          QuickSort(data, key, first, j)
23      if i<last:
24          QuickSort(data, key, i, last)
```

Ilość sortowanych obiektów m w każdym rekurencyjnym wywołaniu maleje o 2, więc otrzymujemy: $\Theta(m) + 2 \cdot \Theta(m/2) = \Theta(m)$, a oczekiwana ilość takich rekurencyjnych wywołań jest: $\Theta(\log n)$ dla n obiektów, co daje złożoność sortowania $\Theta(n \log n)$.

Wywołany dla m sortowanych obiektów:

- $\Theta(1)$
- $\Theta(1)$
- $\Theta(1)$
- $\Theta(m) \cdot \Theta(\text{wnętrze pętli})$
  - $\Theta(1) \cdot \Theta(\text{wnętrze pętli})$
    - $\Theta(1)$
  - $\Theta(1) \cdot \Theta(\text{wnętrze pętli})$
    - $\Theta(1)$
  - $\Theta(1)$
    - $\Theta(\text{Swap})$
    - $\Theta(1)$
    - $\Theta(1)$
- $\Theta(1)$
  - $\Theta(\text{QuickSort dla średnio } m/2 \text{ obiektów})$
- $\Theta(1)$
  - $\Theta(\text{QuickSort dla średnio } m/2 \text{ obiektów})$

Tutaj pętle wewnętrzne są sprzężone z pętlą nadrzędną, tzn. nie iterują po innym atrybucie zależnym od m, lecz po tym samym co pętla główna, stąd nie przemnażamy złożoności obliczeniowych $\Theta(m) \cdot \Theta(m)$

DRZEWO

Drzewo to spójna struktura nieliniowa składająca się z węzłów (wierzchołków) i krawędzi, w której istnieje dokładnie jedna ścieżka pomiędzy parą dowolnych dwóch węzłów.

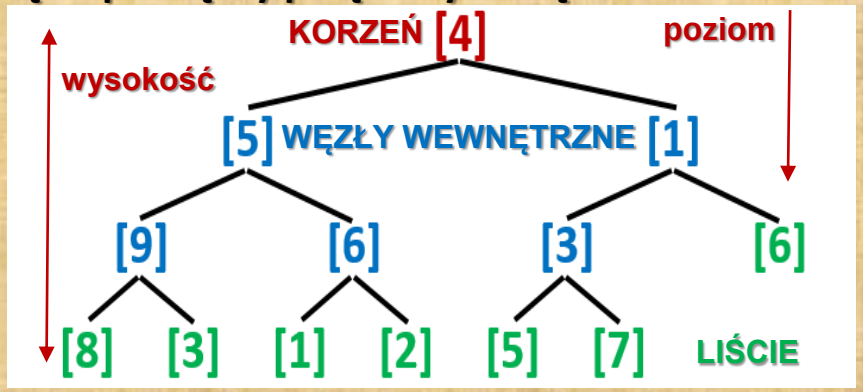
Rodzic (poprzednik, przodek) to węzeł w drzewie, posiadający przynajmniej jedno dziecko (następnika, potomka) będący o jedną krawędź bliżej korzenia w stosunku do węzła będącego jego **dzieckiem** (następnikiem, potomkiem).

Korzeń to węzeł główny w drzewie nie mający rodzica (poprzednika, przodka).

Liście to takie węzły w drzewie, które nie mają dzieci (następników, potomków).

Węzły wewnętrzne to węzły posiadające rodzica i przynajmniej jedno dziecko.

Ścieżka w drzewie to droga wyznaczona przez ciąg krawędzi pomiędzy połączonymi węzłami prowadząca pomiędzy dwoma dowolnymi węzłami. Istnieje zawsze tylko dokładnie jedna ścieżka w drzewie pomiędzy dowolnymi dwoma węzłami.



Długością ścieżki nazywamy ilość krawędzi tworzących ścieżkę w drzewie.

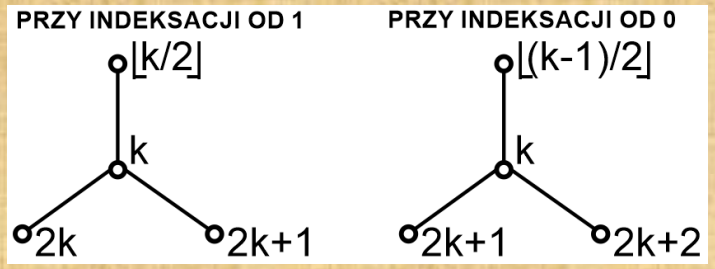
Poziom węzła w drzewie wyznaczony jest równy długości ścieżki łączącej go z korzeniem.

Wysokość drzewa wyznaczona jest przez maksymalny poziom wszystkich jego węzłów.

Drzewo binarne to takie drzewo, którego każdy węzeł (rodzic) ma co najwyżej dwójkę dzieci.

Drzewo regularne to drzewo o określonej maksymalnej ilości dzieci dla każdego węzła.

Drzewo jest zupełne, gdy ma wszystkie poziomy z wyjątkiem ostatniego całkowicie zapełnione, a ostatni jest spójnie zapełniony od strony lewej.



IMPLEMENTACJA DRZEWA W PYTHONIE

```python
# -*- coding: utf-8 -*-
class Node:
    def __init__(self, name="Node", parent=None):
        self.__parent__=parent
        self.__childs__=[]
        self.name=name # nazwa
    def isRoot(self): # jest korzeniem, czy nie posiada rodzica
        if self.__parent__==None: return True
        return False
    def isNode(self): # czy jest węzłem, czyli czy posiada dzieci
        if len(self.__childs__)>=1: return True
        return False
    def getParent(self):
        return self.__parent__
    def getChilds(self):
        return self.__childs__
    def getNext(self): # następny węzeł (sąsiad) powiązany z tym samym rodzicem
        if self.__parent__==None: return None
        n=self.__parent__.getChilds()
        this=False
        for child in n:
            if this and child!=self:
                return child
            if child==self:
                this=True
        return None
    def getPrev(self): # poprzedni węzeł (sąsiad) powiązany z tym samym rodzicem
        if self.__parent__==None: return None
        n=self.__parent__.getChilds()
        p=None
        for child in n:
            if child==self:
                return p
            p=child
        return None
    def removeChild(self, remove): # usuwa węzeł z listy dzieci
        if remove in self.__childs__:
            self.__childs__.remove(remove)
    def setParent(self, newParent): # zmienia rodzica dla węzła
        if newParent.__class__!= Node: return None
        if self.__parent__!=None: # jeśli miał rodzica
            self.__parent__.removeChild(self) # usuwa się z niego
        self.__parent__=newParent # i ustawia nowego rodzica
        newParent.addChild(self) # oraz dodaje się do listy jego dzieci
    def addChild(self, newChild): # dodaje dziecko do węzła
        if newChild.__class__!= Node: return None
        if newChild in self.__childs__: return None
        if newChild.getParent()!=None:
            newChild.getParent().removeChild(newChild) # wcześniej musi jednak
            # usunąć je od poprzedniego rodzica
        self.__childs__.append(newChild)
        newChild.__parent__=self
    def getRoot(self): # szuka korzenia
        root=self
        while root.isRoot()==False: # przechodzi w gore, aż dojdzie do korzenia
            root=root.getParent()
        return root
    def printTree(self, intend=0): # rysowanie drzewa od obecnego elementu
        p=""                       # z podanym wcięciem (zaczynając od 0)
        i=intend
        while i>0:
            if i>1: p+="    +"
            else: p+="    ";
            i-=1 # dodanie wcięcia wcięcia o odpowiedniej wielkości
        if(self.isNode()):
            print(p+"+"+self.name) # jeśli ma dzieci, dodaje +
        else:
            print(p+"-"+self.name) # jeśli nie ma dzieci dodaje -
        for c in self.getChilds(): # i to samo dla każdego dziecka
            c.printTree(intend+1)  # z odpowiednio dużym wcięciem


# Przykład utworzenia drzewa 3-stopnia, potem przeniesienie jednej gałęzi do
# drugiego drzewa i znalezienie rodzica
a=Node("root")
b=Node("child 1")
c=Node("child 2.1")
d=Node("child 2.2")
b.setParent(a)
c.setParent(b)
d.setParent(b)
b.addChild(c)
s=Node("root 2")
s.addChild(b) # Przepisanie drzewa pochodnego od b do korzenia s
print("Root of d: "+d.getRoot().name)
print("neighboor of c: "+c.getNext().name)
print("\nTree:")
c.getRoot().printTree() # Pobiera korzeń i rysuje całe drzewo
```

KOPIEC

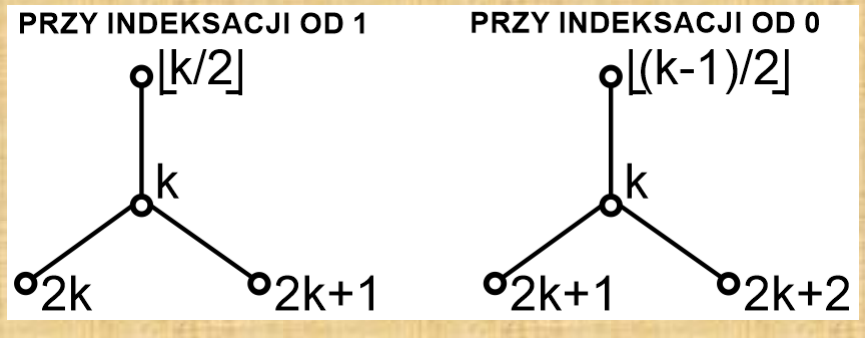
Kopiec to drzewo binarne, w którego węzłach znajdują się elementy reprezentowanego multizbioru, które spełniają **warunek kopca**.

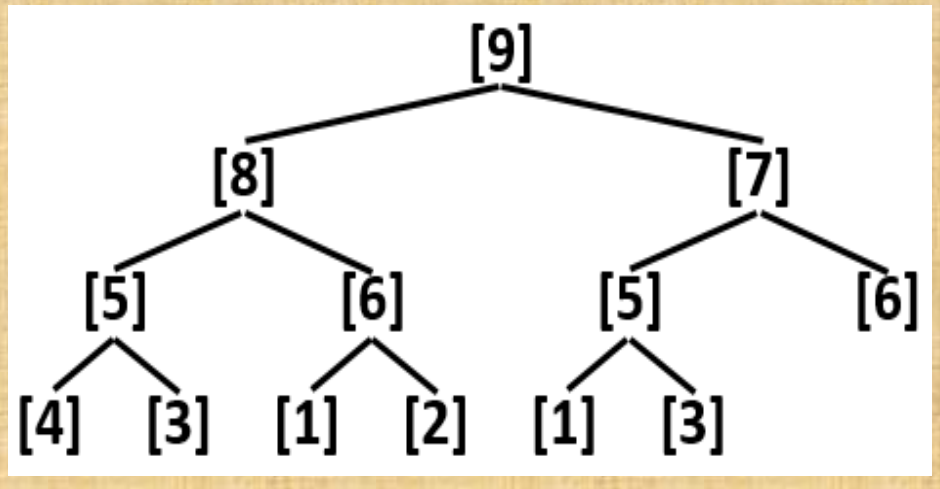
Warunek kopca mówi, iż jeśli węzeł y jest następnikiem węzła x, to element w węźle y jest nie większy niż element w węźle x.

Drzewo ma **uporządkowanie kopcowe**, gdy wszystkie jego elementy zachowują porządek kopcowy, tzn. elementy na ścieżkach w drzewie od korzenia do liścia uporządkowane są w sposób nierosnący.

Kopiec zupełny to zupełne drzewo binarne o uporządkowaniu kopcowym, czyli takie, którego wszystkie poziomy są wypełnione całkowicie za wyjątkiem co najwyżej ostatniego, który jest spójnie wypełniony od strony lewej.

Każde drzewo binarne można w łatwy sposób reprezentować w tablicy, a indeksy określone są przy pomocy następującej zależności:



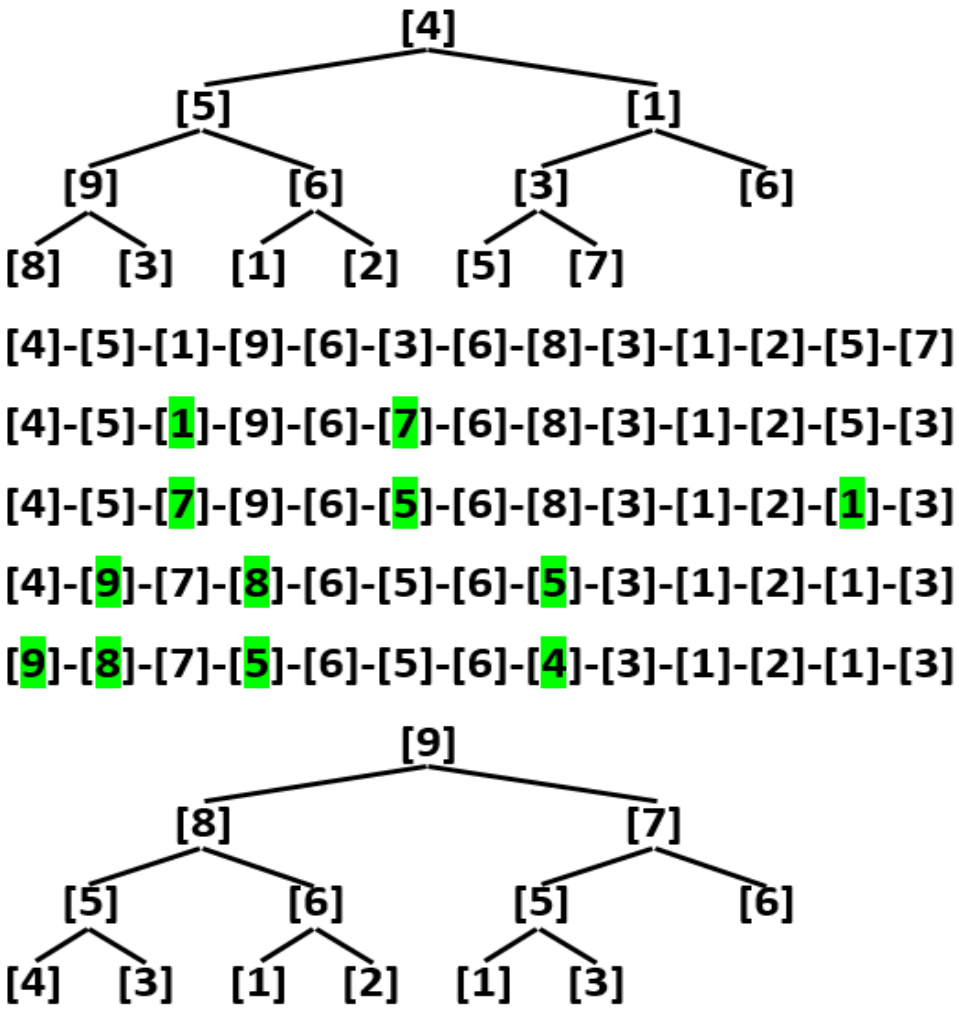


SPOSÓB DZIAŁANIA SORTOWANIA STOGOWEGO

Sortowanie stogowe (heap sort) polega na zbudowaniu kopca zupełnego, który jest drzewem binarnym spełniającym warunek kopca, czyli rodzic jest nie mniejszy niż dziecko, zaś zupełność oznacza, iż wszystkie poziomy drzewa poza ostatnim są w pełni wypełnione, a ostatni spójnie od strony lewej:

```
23      def Swap(data, i, j):
24          data[i], data[j] = data[j], data[i]
25
26      def Heapify(data, key, end, i):
27          l = 2 * i + 1
28          r = 2 * (i + 1)
29          max = i
30          if l < end and data[i][key] < data[l][key]:
31              max = l
32          if r < end and data[max][key] < data[r][key]:
33              max = r
34          if max != i:
35              Swap(data, i, max)
36              if 2 * max + 1 < end:
37                  Heapify(data, key, end, max)
38
39      def HeapSort(data, key):
40          end = len(data)
41          start = end // 2 - 1
42          for i in range(start, -1, -1):
43              Heapify(data, key, end, i)
44          for i in range(end-1, 0, -1):
45              Swap(data, i, 0)
46              Heapify(data, key, i, 0)
```

BUDOWA KOPCA ZUPEŁNEGO:



SPOSÓB DZIAŁANIA SORTOWANIA STOGOWEGO

Sortowanie stogowe (heap sort):

```
23  def Swap(data, i, j):
24      data[i], data[j] = data[j], data[i]
25
26  def Heapify(data, key, end, i):
27      l = 2 * i + 1
28      r = 2 * (i + 1)
29      max = i
30      if l < end and data[i][key] < data[l][key]:
31          max = l
32      if r < end and data[max][key] < data[r][key]:
33          max = r
34      if max != i:
35          Swap(data, i, max)
36          if 2 * max + 1 < end:
37              Heapify(data, key, end, max)
38
39  def HeapSort(data, key):
40      end = len(data)
41      start = end // 2 - 1
42      for i in range(start, -1, -1):
43          Heapify(data, key, end, i)
44      for i in range(end-1, 0, -1):
45          Swap(data, i, 0)
46          Heapify(data, key, i, 0)
```

ROZBIERANIE KOPCA ZUPEŁNEGO:

[8]-[6]-[7]-[5]-[3]-[5]-[6]-[4]-[3]-[1]-[2]-[1]-[9]

[7]-[6]-[6]-[5]-[3]-[5]-[1]-[4]-[3]-[1]-[2]-[8]-[9]

[6]-[5]-[6]-[4]-[3]-[5]-[1]-[2]-[3]-[1]-[7]-[8]-[9]

[6]-[5]-[5]-[4]-[3]-[1]-[1]-[2]-[3]-[6]-[7]-[8]-[9]

[5]-[4]-[5]-[3]-[3]-[1]-[1]-[2]-[6]-[6]-[7]-[8]-[9]

[5]-[4]-[2]-[3]-[3]-[1]-[1]-[5]-[6]-[6]-[7]-[8]-[9]

[4]-[3]-[2]-[1]-[3]-[1]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

[3]-[3]-[2]-[1]-[1]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

[3]-[1]-[2]-[1]-[3]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

[2]-[1]-[1]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

ANALIZA ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie stogowe (heap sort):

Ile operacji elementarnych?

Jak wyznaczyć złożoność czasową?

Które części programu wykonywane są zawsze, które jednorazowo, a które czasami?

Stabilne, w miejscu.

```
23      def Swap(data, i, j):
24          data[i], data[j] = data[j], data[i]
25
26      def Heapify(data, key, end, i):
27          l = 2 * i + 1
28          r = 2 * (i + 1)
29          max = i
30          if l < end and data[i][key] < data[l][key]:
31              max = l
32          if r < end and data[max][key] < data[r][key]:
33              max = r
34          if max != i:
35              Swap(data, i, max)
36              if 2 * max + 1 < end:
37                  Heapify(data, key, end, max)
38
39      def HeapSort(data, key):
40          end = len(data)
41          start = end // 2 - 1
42          for i in range(start, -1, -1):
43              Heapify(data, key, end, i)
44          for i in range(end-1, 0, -1):
45              Swap(data, i, 0)
46              Heapify(data, key, i, 0)
```

```
46      start_time = time.clock()
47      filepath = "resources/iris.csv"
48      table = Table(filepath)
49      end_time = time.clock()
50      elapsed_time_ReadTable = end_time - start_time
51      print("Table read in " + str(elapsed_time_ReadTable))
52
53      start_time = time.clock()
54      for key in range(0,len(table.names)):
55          print("HeapSort for key " + str(key))
56          HeapSort(table.objects, key)
57          for id in range(0, len(table.objects), 1):
58              print(table.objects[id])
59      end_time = time.clock()
60      elapsed_time_HeapSort = end_time - start_time
61      print("Table sorted using HeapSort in " + str(elapsed_time_HeapSort))
```

PRZYKŁAD ANALIZY ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie stogowe (heap sort):

```
23      def Swap(data, i, j):
24          data[i], data[j] = data[j], data[i]
25
26      def Heapify(data, key, end, i):
27          l = 2 * i + 1
28          r = 2 * (i + 1)
29          max = i
30          if l < end and data[i][key] < data[l][key]:
31              max = l
32          if r < end and data[max][key] < data[r][key]:
33              max = r
34          if max != i:
35              Swap(data, i, max)
36              if 2 * max + 1 < end:
37                  Heapify(data, key, end, max)
38
39      def HeapSort(data, key):
40          end = len(data)
41          start = end // 2 - 1
42          for i in range(start, -1, -1):
43              Heapify(data, key, end, i)
44          for i in range(end-1, 0, -1):
45              Swap(data, i, 0)
46              Heapify(data, key, i, 0)
```

Θ(1)

Złożoność stała razy ilość rekurencyjnych wywołań: Θ(log n).

Θ(1)
Θ(1)
Θ(1)
Θ(1)
Θ(1)
Θ(1)
Θ(1)
Θ(Swap)
Θ(1)
Θ(Heapify)

Wywołany dla n sortowanych obiektów rekurencyjnie Θ(log n) razy.

Θ(1)
Θ(1)
Θ(n)
Θ(Heapify)
Θ(n)
Θ(Swap)
Θ(Heapify)

SPOSÓB DZIAŁANIA SORTOWANIA PRZEZ SCALANIE

Sortowanie przez scalanie (merge sort) na koniec listy wynikowej merged dołącza obiekty o rosnących wartościach kluczy uzyskanych z podzielonych posortowanych list we wcześniejszych rekurencyjnych etapach sortowania:

```
48  def MergeSort(data, key):
49      if len(data) < 2:
50          return data
51      else:
52          merged = []
53          mid = len(data) // 2
54          left = MergeSort(data[:mid], key)
55          right = MergeSort(data[mid:], key)
56          while (len(left) > 0) or (len(right) > 0
57              if len(left) == 0:
58                  while len(right) > 0:
59                      merged.append(right.pop(0))
60              elif len(right) == 0:
61                  while len(left) > 0:
62                      merged.append(left.pop(0))
63              else:
64                  if left[0][key] > right[0][key]:
65                      merged.append(right.pop(0))
66                  else:
67                      merged.append(left.pop(0))
68      return merged
```

[4]-[5]-[1]-[9]-[6]-[3]-[6]-[8]-[3]-[1]-[2]-[5]-[7]

[4]

[1]-[5]

[1]-[4]-[5]

[9]

[3]-[6]

[3]-[6]-[9]

[1]-[3]-[4]-[5]-[6]-[9]

[6]

[3]-[8]

[3]-[6]-[8]

[1]-[2]

[5]-[7]

[1]-[2]-[5]-[7]

[1]-[2]-[3]-[5]-[6]-[7]-[8]

[1]-[1]-[2]-[3]-[3]-[4]-[5]-[5]-[6]-[6]-[7]-[8]-[9]

ANALIZA ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie przez scalanie (merge sort):

Na ile kolejność wykonywania porównań może zmniejszyć ilość wykonywanych operacji elementarnych?

Stabilne, nie w miejscu.

```
63  start_time = time.clock()
64  filepath = "resources/iris.csv"
65  table = Table(filepath)
66  end_time = time.clock()
67  elapsed_time_ReadTable = end_time - start_time
68  print("Table read in " + str(elapsed_time_ReadTable))
69
70  start_time = time.clock()
71  for key in range(0,len(table.names)):
72      print("MergeSort for key " + str(key))
73      table.objects = MergeSort(table.objects, key)
74      for id in range(0, len(table.objects), 1):
75          print(table.objects[id])
76  end_time = time.clock()
77  elapsed_time_MergeSort = end_time - start_time
78  print("Table sorted using MergeSort in " + str(elapsed_time_MergeSort))
```

```
48  def MergeSort(data, key):
49      if len(data) < 2:
50          return data
51      else:
52          merged = []
53          mid = len(data) // 2
54          left = MergeSort(data[:mid], key)
55          right = MergeSort(data[mid:], key)
56          while (len(left) > 0) or (len(right) > 0
57              if len(left) == 0:
58                  while len(right) > 0:
59                      merged.append(right.pop(0))
60              elif len(right) == 0:
61                  while len(left) > 0:
62                      merged.append(left.pop(0))
63              else:
64                  if left[0][key] > right[0][key]:
65                      merged.append(right.pop(0))
66                  else:
67                      merged.append(left.pop(0))
68      return merged
```

PRZYKŁAD ANALIZY ZŁOŻONOŚCI OBLICZENIOWEJ

Sortowanie przez scalanie (merge sort)

```
48  def MergeSort(data, key):
49      if len(data) < 2:
50          return data
51      else:
52          merged = []
53          mid = len(data) // 2
54          left = MergeSort(data[:mid], key)
55          right = MergeSort(data[mid:], key)
56          while (len(left) > 0) or (len(right) > 0
57              if len(left) == 0:
58                  while len(right) > 0:
59                      merged.append(right.pop(0))
60              elif len(right) == 0:
61                  while len(left) > 0:
62                      merged.append(left.pop(0))
63              else:
64                  if left[0][key] > right[0][key]:
65                      merged.append(right.pop(0))
66                  else:
67                      merged.append(left.pop(0))
68      return merged
```

Sortowanie przez scalanie na koniec listy wynikowej merged dołącza obiekty o rosnących wartościach kluczy uzyskanych a podzielonych posortowanych list we wcześniejszych rekurencyjnych etapach sortowania.

Θ(1)
Θ(1)

Θ(1)
Θ(2)
Θ(MergeSort m/2 obiektach)
Θ(MergeSort m/2 obiektach)
Θ(5) · Θ(wnętrze pętli)
Θ(2)
Θ(2) · Θ(wnętrze pętli)
Θ(2)
Θ(2)
Θ(2) · Θ(wnętrze pętli)
Θ(2)

Θ(3)
Θ(2)
Θ(2)

Funkcja MergeSort wykonywana jest w czasie stałym, lecz jest rekurencyjnie wywoływana Θ(log n) razy.

Θ(1)

ZŁOŻONOŚĆ PAMIĘCIOWA

Złożoność pamięciowa określa nam, ile dodatkowych jednostek pamięci potrzeba do wykonania algorytmu (rozwiązania zadania).

Złożoność pamięciową również określamy jako funkcję względem rozmiaru danych.

Jeśli w algorytmie sortowania przez scalanie korzystamy z dodatkowej listy **merged[]** do scalania posortowanych list, której wielkość jest zależna od ilości sortowanych danych, wtedy złożoność pamięciowa rośnie i mówimy, że dane sortowanie nie odbywa się w miejscu, lecz **nie w miejscu (*not in place*)**.

```
48  def MergeSort(data, key):
49      if len(data) < 2:
50          return data
51      else:
52          merged = []
53          mid = len(data) // 2
54          left = MergeSort(data[:mid], key)
55          right = MergeSort(data[mid:], key)
56          while (len(left) > 0) or (len(right) > 0
57              if len(left) == 0:
58                  while len(right) > 0:
59                      merged.append(right.pop(0))
60              elif len(right) == 0:
61                  while len(left) > 0:
62                      merged.append(left.pop(0))
63              else:
64                  if left[0][key] > right[0][key]:
65                      merged.append(right.pop(0))
66                  else:
67                      merged.append(left.pop(0))
68      return merged
```

WBUDOWANE SORTOWANIE W PYTHONIE

Sortowanie jest jedną ze standardowych operacji wykonywanych na danych, więc w Pythonie istnieją funkcje sortujące dane **w miejscu**:

list.sort()

sorted(list)

sorted(dictionary)

sorted(tuple)

PRZYKŁADY:

```
list = [['4'], ['5'], ['1'], ['9'], ['6'], ['3'], ['6'], ['8'], ['3'], ['1'], ['2'], ['5'], ['7']]

list.sort()

print(list)
```

zwróci: `[['1'], ['1'], ['2'], ['3'], ['3'], ['4'], ['5'], ['5'], ['6'], ['6'], ['7'], ['8'], ['9']]`

```
print sorted("Ten wykład odkrywa przed nami tajemnice Pythona, algorytmiki oraz programowania".split(), key=str.lower)
```

zwróci: `['algorytmiki', 'nami', 'odkrywa', 'oraz', 'programowania', 'przed', 'Pythona,', 'tajemnice', 'Ten', 'wyk\xc5\x82ad']`

gdzie **key=str.lower** sprawia, iż małe i duże litery są traktowane jednakowo w trakcie sortowania.

BIBLIOGRAFIA I LITERATURA UZUPEŁNIAJĄCA

1. L. Banachowski, K. Diks, W. Rytter: „Algorytmy i struktury danych”, WNT, Warszawa, 2001.
2. Z. Fortuna, B. Macukow, J. Wąsowski: „Metody numeryczne”, WNT, Warszawa, 1993.
3. J. i M. Jankowscy: „Przegląd metod i algorytmów numerycznych”, WNT, Warszawa, 1988.
4. A. Kiełbasiński, H. Schwetlick: „Numeryczna algebra liniowa”, WNT, Warszawa 1992.
5. M. Sysło: „Elementy Informatyki”.
6. A. Szepietowski: „Podstawy Informatyki”.
7. R. Tadeusiewicz, P. Moszner, A. Szydełko: „Teoretyczne podstawy informatyki”.
8. W. M. Turski: „Propedeutyka informatyki”.
9. N. Wirth: „Wstęp do programowania systematycznego”.
10. N. Wirth: „ALGORYTMY + STRUKTURY DANYCH = PROGRAMY”.
11. Złożoność obliczeniowa: http://xion.org.pl/files/texts/mgt/pdf/M_B.pdf
12. Sortowanie w Pythonie: https://docs.python.org/2/howto/sorting.html?highlight=complexity