Gdańsk, 14.12.2021 r.

Sprawozdanie nr 2

Algorytmy i Struktury Danych
Laboratorium Komputerowe

Algorytm: Szukanie interpolacyjne (Interpolation Search)
Student: Aleksander Kruczkowski, 185390
Semestr: III
Stopień studiów: I
Kierunek studiów: Fizyka Techniczna
Specjalność: Informatyka Stosowana
Rok akademicki: 2021/2022

1. Wstęp teoretyczny

1.1. Informacje ogólne

Szukanie interpolacyjne to algorytm wyszukujący klucz w tablicy z wykorzystaniem aproksymacji położenia sondowanego elementu przez interpolację liniową. Taki sposób działania dobrze opisuje analogia do zachowania człowieka szukającego nazwiska w książce telefonicznej.

Interpolation Search wymaga danych posortowanych niemalejąco, by funkcjonować. Charakteryzuje się wysoką wydajnością dla dużych zbiorów o równomiernym rozkładzie danych (kiedy różnice wartości między kolejnymi elementami pozostają niewielkie). Średnia złożoność obliczeniowa to **O(lglgN)**. Jest ona jednak osiągalna dla optymalnych danych. Czas pracy wzrasta dla danych o rozkładzie niejednostajnym, aż do złożoności **θ(N)** (dla danych rosnących eksponencjalnie).

1.2. Sposób działania, pseudokod

Podczas pracy algorytmu według poniższego wzoru wyznaczany jest współczynnik interpolacji **a.**

$$a = \frac{x - Tab[lewa]}{Tab[prawa] - Tab[lewa]}$$

Gdzie:
x - wartość szukana
lewa - klucz elementu będącego lewą granicą przeszukiwanego podprzedziału
prawa - klucz elementu będącego prawą granicą przeszukiwanego podprzedziału

Wyznaczenie **a** pozwala obliczyć pozycję **k** sondowanego elementu. Musi być liczbą całkowitą, więc wykorzystano zaokrąglenie w dół.

$$k = \text{floor}\big(lewa + a(prawa - lewa)\big)$$

Pseudokod

```
function Szuk_inter(Tab : Tab, x : integer) : integer;
    var lewa, prawa : integer;     (lewa i prawa granica szukania)
        k : integer;               (kolejny indeks sondowania)
        a : real;                  (współczynnik interpolacji)

    lewa := 0; prawa := N - 1;     (ustaw początkowe granice)

    repeat
        a := (x - Tab[lewa]) / (Tab[prawa] - Tab[lewa]);  (Ad. 1)
        if a in [0, 1] then              (czy mieści się w [0,1])

            k := floor(lewa + a * (prawa - lewa));      (Ad. 2)
            if x > Tab[k]                    (szukana jest większa)
                lewa := k + 1          (przesuń lewą granicę w prawo)
            else
                if x < Tab[k] then          (szukana jest mniejsza)
                    prawa := k - 1     (przesuń prawą granicę w lewo)
    until Tab[k] = x or a not in [0, 1];

    if Tab[k] = k then
        Szuk_inter := k            (znaleziono szukaną; SUKCES)
    else
        Szuk_inter := 0            (nie znaleziono szukanej; PORAŻKA)

(Ad. 1) - (ustal wsp. interpolacji)
(Ad. 2) - (ustaw indeks sondowania)
```

2. Opis implementacji

2.1. Algorytm zaimplementowany z użyciem Python

Zakładano, że dane wejściowe będą posortowane niemalejąco, zastosowano jednak sprawdzenie czy są poprawnego typu (ponieważ jest pomijalnie kosztowne obliczeniowo) - stąd element “try:” (i na końcu programu “except”).

```
3   def szuk_inter(tab, szukana):
4       lewa, prawa, poz, a = 0, len(tab) - 1, 0, 0
5
6       try:
7           if tab[prawa] == tab[lewa]:
8               if tab[prawa] == szukana:
9                   return prawa
10              else:
11                  return -1
12
```

Na początku programu dodano proste sprawdzenie, czy pierwszy element zbioru nie równa się ostatniemu wartością. Jeśli by tak było, to przy niemalejących danych wszystkie byłyby identyczne.

```
13          while tab[poz] != szukana and a >= 0 and a <= 1:
14
15              try:
16                  a = (szukana - tab[lewa]) / (tab[prawa] - tab[lewa])
17
18              except ZeroDivisionError:
19                  if tab[lewa] == szukana:
20                      return lewa
21                  else:
22                      return -1
23
24              if a >=0 and a <= 1:
25                  poz = math.floor(lewa + a * (prawa - 1))
26                  if szukana > tab[poz]:
27                      lewa = poz + 1
28                  elif szukana < tab[poz]:
29                      prawa = poz - 1
30
31          if tab[poz] == szukana:
32              return poz
33          else:
34              return -1
35
```

Pętla przedstawiona w pseudokodzie jako konstrukcja **repeat … until** została zastosowana jako pętla **while**. Poza odtworzeniem pseudokodu dodano pomijalną dla złożoności obliczeniowej klauzulę **try … except** opisującą co robić w wypadku gdy **lewa == prawa**.

2.2. Funkcja weryfikująca skuteczność algorytmu

W celu sprawdzenia, czy szukanie interpolacyjne dało poprawny wynik wykorzystano funkcję porównującą po kolei każdy element tablicy wejściowej z szukaną liczbą.

```
42  def weryfikator(tab, szukana):
43      for each in tab:
44          if each == szukana:
45              return szukana, tab.index(szukana)
```

3. Opis testów

3.1. Rodzaje testów

- praca na danych optymalnych dla tego algorytmu, czyli dane o rozkładzie jednostajnym,
- praca na danych nieco gorszych szybkości działania algorytmu, dane o rozkładzie mniej równomiernym,
- praca na najgorszym przypadku danych, na których działa algorytm, elementy tablicy wejściowej rosną eksponencjalnie.

Dla tych przypadków wykonano po 12 prób na tablicach o 12 różnych rozmiarach. Odpowiednio **N** = [10, 100, 300, 750, 1500, 5000, 10000, 20000, 30000, 50000, 75000, 100000]. Wszystkie testy przeprowadzono w możliwie podobnych warunkach obciążenia maszyny, jednak niewątpliwie było to źródło wielu błędów pomiarowych. Za każdym razem poszukiwano elementu położonego w okolicy 7/10 długości zbioru. Obliczano to według formuły: **[math.floor(len(*odpowiednia_lista*) * (7/10))]**

3.2. Złożoność obliczeniowa

Zależy ona nie tylko od liczby elementów w tablicy wejściowej, ale także od rozkładu wartości tych danych i wartości poszukiwanej liczby. Widać to przy rozpatrywaniu poniższych dwóch skrajnych przypadków:

Przypadek 1.

Rozkład równomierny (liniowy): $A = \{1, 3, 5, 7, 9, 11, 13, 15\}$; $x = 11$.

l	p	α	k	Wynik sondowania
1	8	10/14	6	$A[6] = 11 = x$; SUKCES

Przypadek 2.

Rozkład bardzo nierównomierny: $A = \{1, 2, 3, 4, 5, 6, 7, 100\}$; $x = 7$.

l	p	α	k	Wynik sondowania
1	8	6/99	1	$A[1] = 1 < x$
2	8	5/98	2	$A[2] = 2 < x$
...	...	...	...	...
6	8	1/94	6	$A[6] = 6 < x$
7	8	0	7	$A[7] = 7 = x$; SUKCES

W pierwszym wypadku poszukiwany element odnaleziono już w 1. próbie, natomiast dla nieprzystosowanych danych potrzeba było aż 7 iteracji.

Złożoność pesymistyczna

Patrząc na powyższe przykłady i formułę obliczania kolejnych indeksów, można zauważyć, że najgorszym przypadkiem byłyby dane wejściowe o wartościach rosnących wykładniczo. Dla takich danych szukanie interpolacyjne ma złożoność **O(n)**.

Złożoność średnia

Uśredniając po wszystkich możliwych rozkładach elementów tablicy i wartościach poszukiwanych otrzymano oszacowanie oczekiwanej liczby sondowań: **O(lglgN)**.

Złożoność optymistyczna

Tylko dla dane o jednostajnym rozkładzie. Jest bardzo zbliżona do średniej: **Θ(lglgN)**.

3.3. Przykładowe dane wejściowe

Dane do testów generowano za pomocą następujących funkcji:

- dane o rozkładzie równomiernym

```
def gen_rowno(n):
    lista_rownomierna = []
    x, i = 0, 0

    while i < n:
        i += 1
        x += 10
        lista_rownomierna.append(x)
    return lista_rownomierna
```

- dane o rozkładzie nierównomiernym

```python
def gen_nierowno(n):
    lista_nierownomierna = []
    x, i = 0, 0
    czesc1 = math.ceil(n/4)
    czesc2 = math.ceil(n/2)

    while i < czesc1:
        i += 1
        x += 10
        lista_nierownomierna.append(x)
    while i < czesc2:
        i += 1
        x += 200
        lista_nierownomierna.append(x)
    while i < n:
        i += 1
        x += 10
        lista_nierownomierna.append(x)
    return lista_nierownomierna
```

- dane rosnące wykładniczo

```python
def gen_exp(n):
    lista_exp = []
    x, i = 2, 0

    while i < n:
        i += 1
        x = 2 ** i

        lista_exp.append(x)
    return lista_exp
```

3.4. Przykładowe dane zwrócone przez testowany algorytm i weryfikator:

```
main
C:\castel_gandolfo\conda\python.exe "E:/Po
70000
weryfikator 70000
Czas szukania interpolacyjnego: 444267500
Czas szukania liniowego: 9924200

Process finished with exit code 0
```

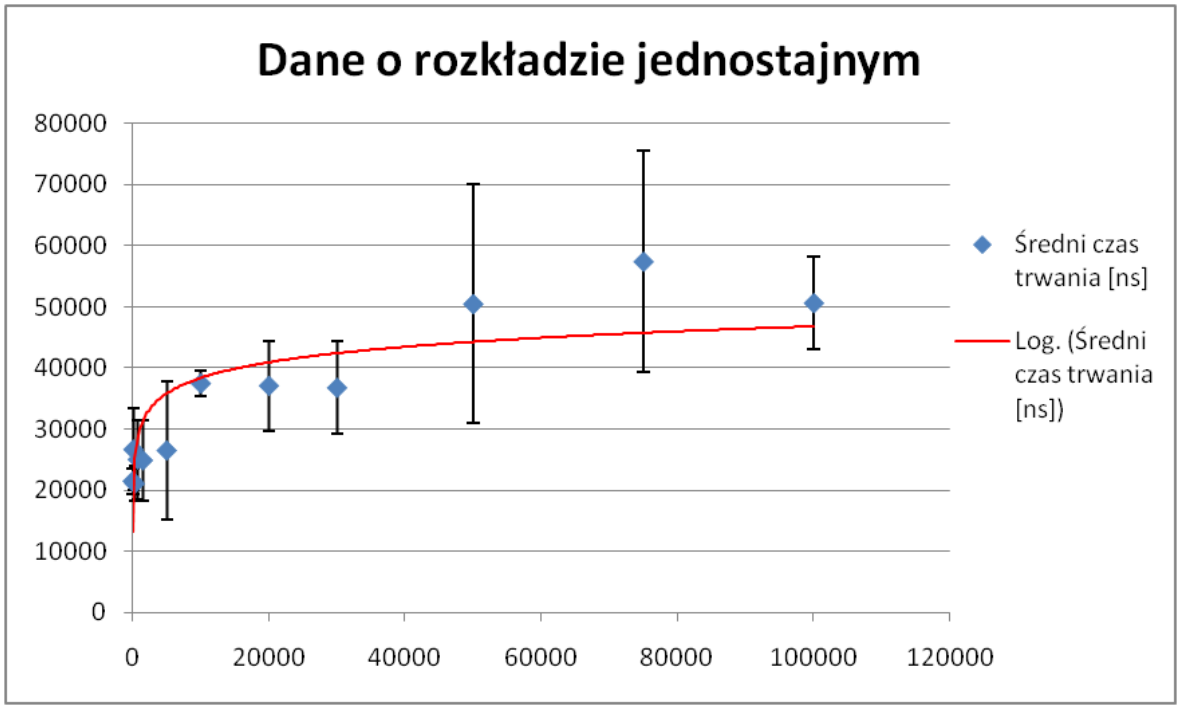
Zwrócone indeksy są równe sobie, dzięki czemu wiadomo, że algorytm działa.

4. Opracowanie wyników testów

4.1. Dane o jednostajnym rozkładzie

L.p.	N	Średni czas trwania [ns]	Odchylenie standardowe
1	10	21483,33	2063,46
2	100	26691,67	6738,82
3	300	21066,67	2825,64
4	750	25041,67	6488,38
5	1500	24908,33	6564,98
6	5000	26516,68	7264,53
7	10000	37508,33	2106,47
8	20000	37125	7339,01
9	30000	36800	7635,32
10	50000	50516,67	19545,56
11	75000	57466,67	18174,77
12	100000	50675	7641,77

Wykres sporządzony na podstawie powyższych danych



Do wartości na wykresie dopasowano funkcję logarytmiczną o równaniu:

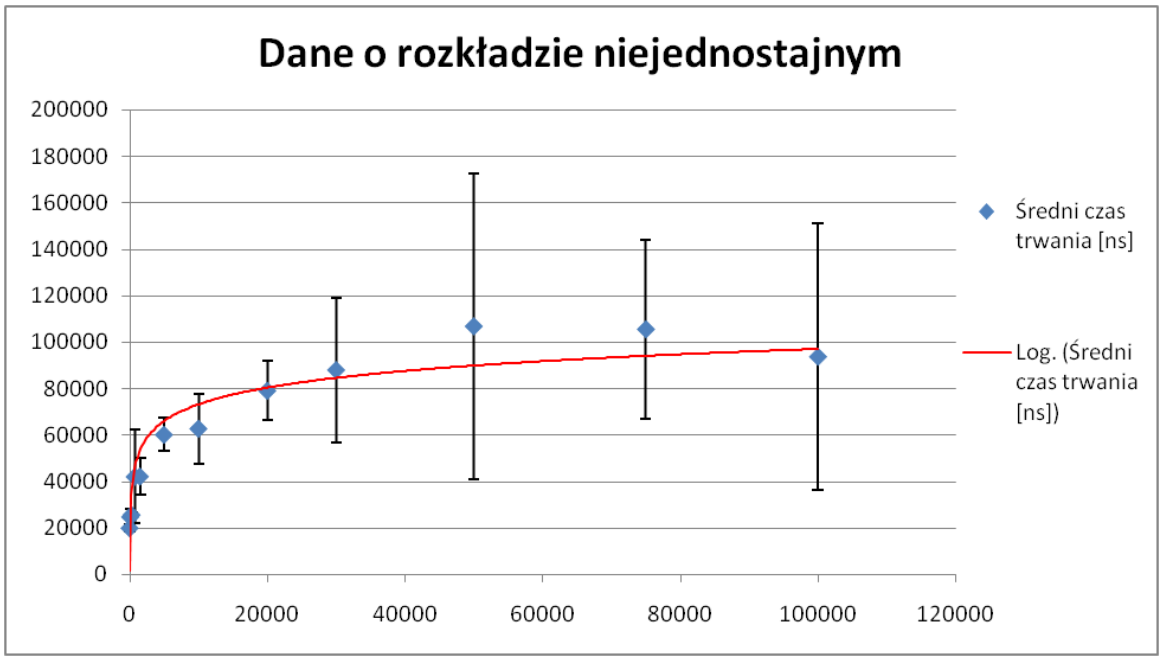
y = 1 * log(x) + 27749,35

Współczynnik wyznaczania $\mathbf{R^2 = 0,94}$. Jest bardzo zbliżony do 1,
co oznacza bardzo dobre dopasowanie krzywej.
Widoczny jest wysoki stopień dopasowania funkcji do otrzymanych wyników.

4.2. Dane o niejednostajnym rozkładzie

L.p.	N	Średni czas trwania [ns]	Odchylenie standardowe
1	10	20100	1450,39
2	100	24883,33	3235,55
3	300	25741,67	1174,31
4	750	42141,67	20106,1
5	1500	42258,33	7919,65
6	5000	60150	7161,83
7	10000	62741,67	14988,08
8	20000	79025	12739,21
9	30000	87925	31083
10	50000	106700	65695,37
11	75000	105400	38615,78
12	100000	93625	57341,82

Wykres sporządzony na podstawie powyższych danych

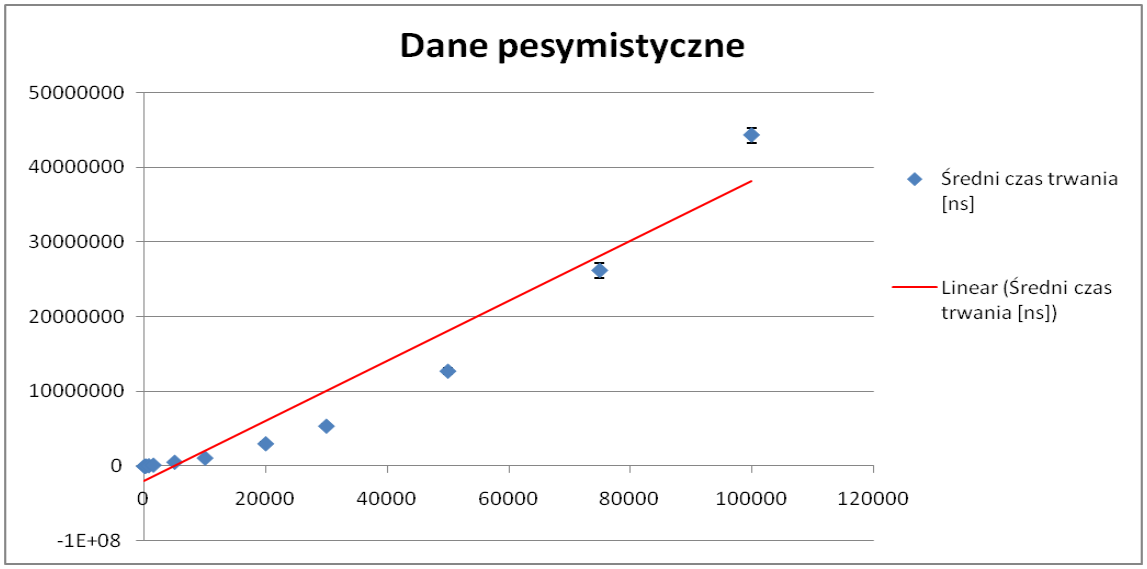


Dopasowano funkcję logarytmiczną o równaniu:
y = 1 * log(N) + 39381,80
Współczynnik wyznaczania $\mathbf{R^2}$ = **0,54**. Dopasowanie krzywej jest znacznie mniej precyzyjne, niż dla danych zrównoważonych. Widoczne jest również znaczne zwiększenie średnich czasów pracy algorytmu.
Oczekiwano zależności ciężkiej do określenia jako logarytmiczną. Spodziewana złożoność obliczeniowa miała być czymś pośrednim między logarytmiczną a liniową zależnością.

4.3. Dane pesymistyczne (elementy rosną wykładniczo)

L.p.	N	Średni czas trwania [ns]	Odchylenie standardowe
1	10	12408,33	545,16
2	100	84883,33	16338,57
3	300	310433,33	23184,02
4	750	736375	65322,53
5	1500	1609633,33	305984,81
6	5000	5376058,33	945192,41
7	10000	10998691,67	1316398,41
8	20000	30130500	2604292,62
9	30000	53658625	2685321,27
10	50000	127172791,7	3899538,18
11	75000	262011716,7	9823129,2
12	100000	442983725	10335863,3

Wykres sporządzony na podstawie powyższych danych



Do wykresu dopasowano funkcję liniową o równaniu:
y = 4015,57081 * N - 20009259,31
Współczynnik wyznaczania $\mathbf{R^2}$ = **0,94**. Dopasowanie funkcji liniowej skutecznie określa zależność czasu pracy algorytmu od ilości przetwarzanych elementów.

Wartości błędów pomiarowych są jednak bardzo niskie i dopasowana krzywa nie mieści się w nich. Analizując wartości sprawdzono korelację z tendencją liniową oraz wykładniczą. Choć wykres sprawia wrażenie wystąpienia złożoności wykładniczej przez większość punktów pomiarowych, dla ostatnich elementów przestaje być to prawdziwe. Również współczynnik $\mathbf{R^2}$ był niższy, niż dla dopasowania funkcji liniowej.
Oczekiwano, że dla danych rosnących eksponencjalnie szukanie interpolacyjne będzie miało zależność liniową od wielkości przeszukiwanych zbiorów i taką zależność otrzymano. Widoczny jest znaczny wzrost średnich czasów poszukiwań względem poprzednich badań.

5. Podsumowanie i wnioski

Dopasowanie złożoności obliczeniowej algorytmu do wyników pomiarowych jest akceptowalne. Krzywe naniesione na wykresy mieściły się w przedziałach błędów pomiarowych.

Szukanie interpolacyjne jest wysoce skuteczne pod względem czasu obliczeń pod warunkiem zastosowania go do ściśle określonych danych, czyli zbiorów posortowanych niemalejąco, o jednostajnym rozkładzie wartości. Ponadto, muszą one być bardzo duże (**N** rzędu kilkudziesięciu tysięcy i więcej), gdyż Interpolation Search wymaga znacznie bardziej złożonej arytmetyki niż np. szukanie binarne, więc czas zużyty na jedno sondowanie jest znacznie dłuższy. Podczas testów funkcja weryfikująca działająca w zależności liniowej od **N** wykazywała znacznie większą skuteczność do momentu, kiedy **N** było równe 750.

Ciężko przewidzieć czas pracy szukania interpolacyjnego danych o nieznanym rozkładzie wartości. Szybkość znajdowania wybranych wartości znacznie spada przy nierównomiernym wzroście elementów przeszukiwanych.

Podobnie problematyczne jest wyznaczenie średniej złożoności obliczeniowej w sposób analityczny. Dane zewnętrzne i wykonane testy wskazują złożoność logarytmiczną, a dokładnie **O(lglgN)**.

Przy podejrzeniu, że rozkład elementów w tablicy znacznie odbiega od liniowego lepiej wykorzystać szukanie binarne, które jest bardzo zbliżone w działaniu do szukania interpolacyjnego. Interpolacyjne jest wręcz wariantem szukania binarnego dla specyficznych danych, jednak jest skuteczniejsze tylko w precyzyjnie określonych warunkach.

6. Literatura

1. Krzysztof Goczyła *Struktury danych*, Wydawnictwo Politechniki Gdańskiej, 2002
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Wprowadzenie do algorytmów*. Wydawnictwa Naukowo-Techniczne, 2007. ISBN 978-83-204-3328-9. OCLC 749241843
3. S. L. Graham, R. L. Rivest *Programming Techniques*: http://www.cs.technion.ac.il/~itai/publications/Algorithms/p550-perl.pdf

Załączniki

1. folder skompresowany z kodem programu, spr2_Kruczkowski.zip