Wstęp do informatyki

Wykład 6

Uniwersytet Wrocławski

Plan na dziś

1. Problem sortowania; sortowanie przez wstawianie (insert sort)
2. Wyszukiwanie binarne
3. Usprawnianie programów/algorytmów:
   - Wyszukiwanie z wartownikiem
   - Schemat Hornera

Sortowanie przy pomocy porównań

Sortowanie przy pomocy porównań

Specyfikacja

Wejście: ciąg elementów $a_1, \ldots, a_n$ ze zbioru uporządkowanego Z

Wyjście: permutacja $b_1,\ldots,b_n$ ciągu $a_1,\ldots a_n$ taka, że $b_1 \leq \ldots \leq b_n$

Dlaczego „zbiór uporządkowany”?

- elementy możemy porównywać,
- rozwiązanie działa niezależnie od wyboru Z (liczby naturalne, rzeczywiste, słowa z porządkiem leksykograficznym, itp.)
- nie można więc wykorzystywać specyfiki zbioru Z (np. nie działa sortowanie przez zliczanie)

Sortowanie przy pomocy porównań

Specyfikacja konkretniej

Wejście:

ciąg elementów a_1, …, a_n ze zbioru uporządkowanego Z, umieszczony w tablicy a[0],…,a[n-1]

Wyjście:

permutacja $b_1,\ldots,b_n$ ciągu $a_1,\ldots a_n$ taka, że $b_1 \leq \ldots \leq b_n$, umieszczona w a[0],…,a[n-1]

Sortowanie przy pomocy porównań

Klasyczne algorytmy sortowania:

- Sortowanie przez wybór (ćw.)
- Sortowanie bąbelkowe (ćw.)
- Shell sort
- Sortowanie przez wstawianie (**wykład**)

(dodatkowe) Kryteria oceny algorytmu sortowania:

- liczba porównań elementów ze zbioru Z
- liczba podstawień (lub zamian) elementów ze zbioru Z

Sortowanie przez wstawianie

Specyfikacja (przypomnienie)

Wejście: a[0],…,a[n-1] – elementy zbioru uporządkowanego Z

Wyjście: a[0],…,a[n-1] zawiera permutację oryginalnego ciągu a[0],…,a[n-1] taką, że a[0]≤… ≤a[n-1]

Algorytm (pseudokod)

Dla k = 1 ,..., n – 1 **powtarzaj**:

1. Znajdź ***p***, pozycję „dla” a[k] w posortowanym ciągu a[0]≤… ≤a[k-1], tzn., takie że
   - a[p-1] ≤ a[k] ≤ a[p] dla 0 < p < k **lub**
   - a[k] ≤ a[p] dla p=0 **lub**
   - a[p – 1] ≤ a[k] dla p=k
2. Przesuń elementy a[p],…,a[k-1] o jedną pozycję w prawo,
3. Przenieś a[k] na pozycję p.

Sortowanie przez wstawianie - implementacja

Algorytm (pseudokod)

Dla *k* = 1 ,..., *n* – 1 **powtarzaj**:

1. Znajdź ***p***, pozycję „dla" a[k] w posortowanym ciągu a[0]≤… ≤a[k-1], tzn., takie że
   - a[p-1] ≤ a[k] ≤ a[p] dla 0 < p < k **lub**
   - a[k] ≤ a[p] dla p=0 **lub**
   - a[p – 1] ≤ a[k] dla p=k
2. Przesuń elementy a[p],…,a[k-1] o jedną pozycję w prawo,
3. Przenieś a[k] na pozycję p.

```
void insertSort(int n,int a[])
{ int x, k, p;
  for (k=1; k<n; k++) {
    x=a[k];
    p=k;
    while (p>0 && x<a[p-1])
    { a[p]=a[p-1];
      p--;
    }
    a[p]=x;
  }
}
```

Uwagi:

- Pozycji p, na którą wstawimy a[k] szukamy „od prawej do lewej"
- Dzięki temu poszukiwanie pozycji możemy połączyć z przesuwaniem elementów w prawo (a[k] zapamiętujemy w pomocniczej zmiennej x)

Sortowanie przez wstawianie - implementacja

Algorytm (pseudokod)

Dla *k* = 1 ,..., *n* – 1 **powtarzaj**:

1. Znajdź ***p***, pozycję „dla” a[k] w posortowanym ciągu a[0]≤… ≤a[k-1], tzn., takie że
   - a[p-1] ≤ a[k] ≤ a[p] dla 0 < p < k **lub**
   - a[k] ≤ a[p] dla p=0 **lub**
   - a[p – 1] ≤ a[k] dla p=k
2. Przesuń elementy a[p],…,a[k-1] o jedną pozycję w prawo,
3. Przenieś a[k] na pozycję p.

```
def insertSort(n,a):
    for k in range(1,n):
        x=a[k]
        p=k
        while p>0 and x<a[p-1]:
            a[p]=a[p-1]
            p=p-1
        a[p]=x
```

Uwagi:

- Pozycji p, na którą wstawimy a[k] szukamy „od prawej do lewej”
- Dzięki temu poszukiwanie pozycji możemy połączyć z przesuwaniem elementów w prawo (a[k] zapamiętujemy w pomocniczej zmiennej x)

Sortowanie przez wstawianie - złożoność

Algorytm (pseudokod)

Dla $k = 1, \ldots, n - 1$ **powtarzaj**:

1. Znajdź ***p***, pozycję „dla” a[k] w posortowanym ciągu $a[0] \leq \ldots \leq a[k-1]$, tzn., takie że
   - $a[p-1] \leq a[k] \leq a[p]$ dla $0 < p < k$ **lub**
   - $a[k] \leq a[p]$ dla $p=0$ **lub**
   - $a[p-1] \leq a[k]$ dla $p=k$
2. Przesuń elementy a[p],…,a[k-1] o jedną pozycję w prawo,
3. Przenieś a[k] na pozycję p.

Złożoność czasowa:

- Kroki 1., 2. i 3. powtarzamy n – 1 razy
- *k*-te wykonanie kroków 1., 2. i 3. ma złożoność O(k) [szukamy pozycji elementu a[k] w ciągu a[0],…,a[k-1]
- Złożoność algorytmu $O(1 + 2 + \ldots + (n-1)) = O(n(n-1)/2) = O(n^2)$

Złożoność pamięciowa:

- Potrzebujemy tablicy rozmiaru n i stałej liczby pomocniczych zmiennych
- Złożoność O(n)
- Dokładniej: wystarczy pamięć n + O(1)

Sortowanie w miejscu: oprócz tablicy z danymi, pamięć tylko O(1).

Sortowanie przez wstawianie – złożoność cd

Liczba porównań (elementów sortowanego zbioru):
- k-te wykonanie kroków 1., 2. i 3. wymaga co najwyżej k porównań
- Liczba porównań to O(1 + 2 + … + (n – 1)) = O(n(n-1)/2) = O(n^2)

Liczba przestawień (podstawień) elementów sortowanego zbioru:
- w k-tym wykonaniu kroków 1., 2. i 3. „przesuwamy” co najwyżej k elementów
- liczba przestawień O(1 + 2 + … + (n – 1)) = O(n(n-1)/2) = O(n^2)

Pytanie

Potrafisz skonstruować algorytm z mniejszą liczbą
- *porównań?*
- *przestawień/podstawień?*

Sortowanie przez wstawianie – złożoność cd

Zależność czasu działania od danych wejściowych

Dane wejściowe	Porównania	Podstawienia
$a[0] \le a[1] \le \ldots \le a[n-1]$	$n-1$	$n-1$
$a[0] \ge a[1] \ge \ldots \ge a[n-1]$	$n \cdot (n-1) / 2$	$n \cdot (n-1) / 2$

Wyszukiwanie binarne

Wyszukiwanie w ciągu uporządkowanym

Specyfikacja problemu

Wejście:

- n – liczba naturalna
- ciąg elementów $a_1 \leq \ldots \leq a_n$ ze zbioru uporządkowanego Z, umieszczony w tablicy a[0],…,a[n-1]
- x – element zbioru Z

Wyjście:

- p – pozycja elementu x w ciągu a[0],…,a[n-1] LUB
- -1 – gdy x nie występuje w ciągu a[0],…,a[n-1]

Wyszukiwanie w ciągu uporządkowanym

Specyfikacja podproblemu

Wejście:

- b, e – liczby naturalne
- ciąg $a[b] \leq \ldots \leq a[e]$ elementów zbioru uporządkowanego Z
- x – element zbioru Z

Wyjście:

- p – pozycja elementu x w ciągu a[b],…,a[e] LUB
- -1 – gdy x nie występuje w ciągu a[b],…,a[e]

Algorytm binarnego wyszukiwania

1. Jeśli b > e : zwróć -1 // ciąg pusty
2. s ← (b + e) / 2 // s to „środek” przedziału [b,e]
3. Jeśli x = a[s] : zwróć s
4. Jeśli x < a[s] : przeszukaj a[b]..a[s-1] // x nie występuje w a[s]…a[e]
5. Jeśli x > a[s] : przeszukaj a[s+1]..a[e] // x nie występuje w a[b]…a[s]

Wyszukiwanie w ciągu uporządkowanym

Algorytm binarnego wyszukiwania

1. Jeśli b > e : zwróć -1 // ciąg pusty
2. s ← (b + e) / 2 // s to „środek” przedziału [b,e]
3. Jeśli x = a[s] : zwróć s
4. Jeśli x < a[s] : przeszukaj a[b]..a[s-1] // x nie występuje w a[s]…a[e]
5. Jeśli x > a[s] : przeszukaj a[s+1]..a[e] // x nie występuje w a[b]…a[s]

```
int znajdzR(int b, int e,
            int a[], int x)
{int s;
 if (b>e) return -1;
 s = (b+e)/2;
 if (a[s]==x) return s;
 if (x<a[s])
  return znajdzR(b,s-1,a,x);
 return znajdzR(s+1,e,a,x);
}
```

```
def znajdzR(b,e,a,x):
 if (b>e): return -1
 s = (b+e)/2
 if a[s]==x:
  return s
 if x<a[s]:
  return znajdzR(b,s-1,a,x)
 return znajdzR(s+1,e,a,x)
```

Wyszukiwanie w ciągu uporządkowanym

<u>Oryginalna specyfikacja problemu</u>

Wejście: n – liczba naturalna

- <u>uporządkowany</u> ciąg elementów a[0] ≤ …≤ a[n-1]
- x – element

Wyjście:

p – pozycja elementu x w ciągu a[0],…,a[n-1] LUB

-1 – gdy x nie występuje w ciągu a[0],…,a[n-1]

<u>**Specyfikacja realizowana przez** znajdzR(**int** b, **int** e, **int** a[], **int** x)</u>

Wejście:

- b, e – liczby naturalne
- <u>uporządkowany</u> ciąg a[b] ≤ … ≤ a[e] elementów
- x – element

Wyjście:

p – pozycja elementu x w ciągu a[b],…,a[e] LUB

-1 – gdy x nie występuje w ciągu a[b],…,a[e]

```
int znajdz(int n, int a[], int x)
{ return znajdzR(0,n-1,a,x); }
```

```
def znajdz(n, a, x):
    return znajdzR(0,n-1,a,x)
```

Wyszukiwanie bez rekurencji

Algorytm binarnego wyszukiwania

1. Jeśli b > e : zwróć -1 // ciąg pusty
2. s ← (b + e) / 2 // s to „środek” przedziału [b,e]
3. Jeśli x = a[s] : zwróć s
4. Jeśli x < a[s] : przeszukaj a[b]..a[s-1] // x nie występuje w a[s]…a[e]
5. Jeśli x > a[s] : przeszukaj a[s+1]..a[e] // x nie występuje w a[b]…a[s]

```
int znajdzR(int b, int e, int a[], int x)
{ int s;
  if (b>e) return -1;
  s = (b+e)/2;
  if (a[s]==x) return s;
  if (x<a[s]) return znajdzR(b,s-1,a,x);
  return znajdzR(s+1,e,a,x);
}

int znajdz(int n, int a[], int x)
{ return znajdzR(0,n-1,a,x);}
```

```
int znajdzI(int n, int a[], int x)
{ int b,e,s;
  b = 0; e = n - 1;
  while (b<=e){
    s = (b+e)/2;
    if (a[s]==x) return s;
    if (x<a[s]) e=s-1;
    else b=s+1;
  }
  return -1;}
```

Zamiast wywołań rekurencyjnych:

1. pętla „while (b<=e)” (przeciwny do bezwarunkowego zakończenia)
2. wywołanie z nowymi wartościami b i e: zmiana tych wartości w pętli

Wyszukiwanie bez rekurencji

Algorytm binarnego wyszukiwania

1. Jeśli b > e : zwróć -1 // ciąg pusty
2. s ← (b + e) / 2 // s to „środek" przedziału [b,e]
3. Jeśli x = a[s] : zwróć s
4. Jeśli x < a[s] : przeszukaj a[b]..a[s-1] // x nie występuje w a[s]…a[e]
5. Jeśli x > a[s] : przeszukaj a[s+1]..a[e] // x nie występuje w a[b]…a[s]

```
def znajdzR(b,e,a,x):
    if (b>e): return -1
    s = (b+e)/2
    if a[s]==x:
        return s
    if x<a[s]: return znajdzR(b,s-1,a,x)
    return znajdzR(s+1,e,a,x)

def znajdz(n, a, x):
    return znajdzR(0,n-1,a,x)
```

```
def znajdzI(n, a, x):
  b = 0
  e = n - 1
  while b<=e:
     s = (b+e)/2
     if a[s]==x: return s
     if x<a[s]: e=s-1
     else: b=s+1
  return -1
```

Zamiast wywołań rekurencyjnych:

1. pętla „while (b<=e)" (przeciwny do bezwarunkowego zakończenia)
2. wywołanie z nowymi wartościami b i e: zmiana tych wartości w pętli

Wyszukiwanie binarne – poprawność

Formalny dowód poprawności algorytmu (pierwsza przymiarka) - etapy:

- zwracany wynik jest (zawsze) zgodny ze specyfikacją;
- algorytm zawsze kończy działanie (własność stopu).

Wyszukiwanie binarne – poprawność wyniku

Oznaczenie: a[i,j] = a[i], a[i+1],…,a[j]

Własność 1:

- $x \notin$ a[0,b – 1] $\cup$ a[e+1,n – 1] **(∗)**

DOWÓD

1. Na początku b=0 i e = n – 1, więc a[0,b – 1] $\cup$ a[e+1,n – 1] = $\emptyset$
2. if (x<a[s]) e=s-1: uporządkowanie ciągu gwarantuje, że x $\notin$ a[s,e], więc podstawienie e=s-1 zachowuje **(∗)**;
3. if (x>a[s]) b=s+1: uporządkowanie ciągu gwarantuje, że x $\notin$ a[b,s], więc podstawienie b=s+1 zachowuje **(∗)**;

Wniosek

- Jeśli algorytm zwraca wartość s≥0, działa zgodnie ze specyfikacją (wskazuje element a[s] równy x)
- Jeśli algorytm zwraca wartość -1, to również działa zgodnie ze specyfikacją:
  - wówczas n>b>e>-1 czyli $x \notin$ a[0,b – 1] $\cup$ a[e+1,n – 1] = a[0, n – 1] (Własność 1)

Wyszukiwanie binarne – własność stopu

Własność 2:
Każda iteracja zmniejsza wielkość przeszukiwanego przedziału (co najmniej) dwukrotnie.

Inaczej mówiąc:
Niech b_2, e_2 oraz b_1, e_1 to wartości zmiennych b, e na początku i końcu jednego wykonania instrukcji pętli oraz $e_2 \geq b_2$. Wówczas:

$e_2 - b_2 + 1 \leq (e_1 - b_1 + 1) / 2$

DOWÓD (żmudna ale dokładna wersja; ten jeden raz ☺)

Przypadek 1: $e_1 + b_1$ parzyste

Wówczas $s = (e_1 + b_1) / 2$ oraz

$e_2 - b_2 + 1 = (s - 1) - b_1 + 1 = (e_1 + b_1) / 2 - b_1 < (e_1 - b_1 + 1) / 2$

LUB

$e_2 - b_2 + 1 = e_1 - (s + 1) + 1 = e_1 - (e_1 + b_1) / 2 < (e_1 - b_1 + 1) / 2$

Przypadek 2: $e_1 + b_1$ nieparzyste

Wówczas $s = (e_1 + b_1 - 1) / 2$ oraz

$e_2 - b_2 + 1 = (s - 1) - b_1 + 1 = (e_1 + b_1 - 1) / 2 - b_1 < (e_1 - b_1 + 1) / 2$

LUB

$e_2 - b_2 + 1 = e_1 - (s + 1) + 1 = e_1 - (e_1 + b_1 - 1) / 2 = (e_1 - b_1 + 1) / 2$

Wyszukiwanie binarne – własność stopu

Własność 2:
Każda iteracja zmniejsza wielkość przedziału (co najmniej) dwukrotnie.
Inaczej mówiąc:
Niech b_2, e_2 oraz b_1, e_1 to wartości zmiennych b, e na początku i końcu jednego wykonania instrukcji pętli oraz $e_2 \geq b_2$. Wówczas:
$e_2 - b_2 + 1 \leq (e_1 - b_1 + 1) / 2$

Wniosek
Algorytm binarnego wyszukiwania zawsze kończy działanie.

Dowód
- e – b +1 jest zawsze liczbą całkowitą
- więc Własność 2 oznacza, że wartość e – b + 1 będzie równa 1 po skończonej liczbie iteracji (na razie nie liczymy po ilu…)
- zatem algorytm zakończy działanie (sprawdź zachowanie gdy e=b…)

Wyszukiwanie binarne - złożoność

Złożoność czasowa

- każde wywołanie rekurencyjne / iteracja zmniejsza (co najmniej) dwukrotnie długość ciągu, czyli różnicę $e - b + 1$
- (najpóźniej) obliczenia kończą się po wywołaniu, w którym długość ciągu to 1
- Początkowa długość ciągu: $e - b + 1 = n$
- Czas: $O(\log n)$

Złożoność pamięciowa

- implementacja iteracyjna (bez rekurencji) – pamięć $O(1)$ [+tabl.z danymi]
- implementacja rekurencyjna:
  - pamięć zajmowana przez stos wywołań, proporcjonalna do **głębokości** drzewa wywołań rekurencyjnych
  - głębokość drzewa wywołań szacujemy tak samo jak czas: $O(\log n)$
    - można zmniejszyć – rekursja ogonowa….

Wyszukiwanie z wartownikiem czyli „podrasowujemy”

Wyszukiwanie z wartownikiem

Specyfikacja problemu

Wejście:

- n – liczba naturalna większa od zera
- ciąg elementów $a_1,\ldots,a_n$ ze zbioru Z, umieszczony w tablicy a[0],…,a[n-1]
- x – element zbioru Z

Wyjście:

- p – pozycja elementu x w ciągu a[0],…,a[n-1] LUB
- -1 – gdy x nie występuje w ciągu a[0],…,a[n-1]

```
int znajdz(int n,int a[],int x)
{ int i;
  for(i = 0; i<n; i++)
       if (a[i]==x) return i;
  return -1;
}
```

```
def znajdz(n, a, x):
    for i in range(n):
        if a[i]==x: return i
    return -1
```

Wyszukiwanie z wartownikiem

```
int znajdz(int n,int a[],int x)
{ int i;
  for(i = 0; i<n; i++)
       if (a[i]==x) return i;
  return -1;
}
```

```
def znajdz(n, a, x):
    for i in range(n):
        if a[i]==x: return i
    return -1
```

W czym problem?

- wyszukiwanie często wykonywaną operacją
- (w najgorszym przypadku) n-krotnie musimy powtarzać sprawdzanie czy ciąg nie zakończył się: i<n

Idea wartownika:

- umieść na końcu przeszukiwanego ciągu szukany element x;
- wówczas wyszukiwanie na pewno zakończy się znalezieniem wystąpienia x… nie musimy sprawdzać czy koniec ciągu;
- na końcu (za pętlą) sprawdź czy jest to „rzeczywiste” wystąpienie.

UWAGA

Celem nie jest poprawa złożoności asymptotycznej.

Wyszukiwanie z wartownikiem

Bez wartownika:

```
int znajdz(int n,int a[],int x)
{  int i;
   for(i = 0; i<n; i++)
          if (a[i]==x) return i;
   return -1;
}
```

```
def znajdz(n, a, x):
    for i in range(n):
        if a[i]==x: return i
    return -1
```

Wartownik na „dodatkowej” pozycji:

```
int znajdz(int n, int a[], int x)
{  int i;
    a[n] = x;
   for(i = 0; a[i]!=x; i++) ;

    if (i<n) return i;
   return -1;
}
```

```
def znajdz(n, a, x):
    a[n] = x
    i=0
    while a[i]!=x:
        i=i+1
    if (i<n):
        return i
    return -1
```

Wyszukiwanie z wartownikiem

Efekt:

- Liczba porównań elementów zbioru Z nadal (co najwyżej) n
- Eliminujemy porównania i<n kosztem stałej liczby operacji.
- Złożoność asymptotyczna O(n) nie ulega zmianie, ale w praktyce przyspieszamy kluczowe obliczenia

Schemat Hornera

Wartościowanie wielomianu

Specyfikacja:

Wejście:
- n – liczba naturalna (stopień wielomianu,)
- a[0], a[1],…,a[n] – współczynniki wielomianu (rzeczywiste);
- x – liczba rzeczywista;

Wyjście: wartość wielomianu w punkcie x, czyli

$$a[0] + a[1] * x + a[2] * x^2 + \ldots + a[n] * x^n$$

Algorytm (w miarę sprytny):
1. potega ← 1
2. wynik ← 0
3. Powtarzaj dla i=0,1,…,n:
   - wynik ← wynik + a[i] * potega
   - potega ← potega * x
4. Zwróć wynik

Złożoność czasowa:
- Czas O(n)
- Liczba mnożeń: 2n

Wartościowanie wielomianu

Specyfikacja:

Wejście:

- n – stopień wielomianu,
- a[0], a[1],…,a[n] – współczynniki wielomianu (rzeczywiste);
- x – liczba rzeczywista;

Wyjście: wartość wielomianu w punkcie x, czyli

$$a[0] + a[1] * x + a[2] * x^2 + \ldots + a[n] * x^n$$

Algorytm (w miarę sprytny):

```
int  wiel(int  n,  float  a[],
   float x)
{ float wynik = 0, potega=1;
  int i;
  for(i=0; i <= n; i++){
    wynik += a[i] * potega;
     potega *= x;
  }
  return wynik;
}
```

```
def wiel(n, a, x):
    wynik = 0.0
    potega=1.0
    for i in range(n+1):
        wynik+=a[i]*potega
        potega *= x
    return wynik
```

Wartościowanie wielomianu - schemat Hornera

Zauważmy, że

$$w(x)=a_0 + a_1 * x + a_2 * x^2 + \ldots + a_n * x^n$$

Możemy zapisać

$$w(x) = a_0 + x\ (a_1 + x\ (a_2 + x\ (a_3\ \ldots + x(a_{n-1} + a_n * x)...)$$

wyciągając x „przed nawias” zawsze gdy to możliwe.

Niech:

$h_n\,(\mathrm{x}) = a_n$

$h_k\,(\mathrm{x}) = a_k + h_{k+1}\,(\mathrm{x}) * x$ dla $k < n$

Wówczas:

$h_0 = w(x)$.

Schemat Hornera

Przykład

$w(x) = a_0 + a_1 * x + a_2 * x^2 + a_3 * x^3 \quad = 2 + 3x + 5x^2 + 4x^3$

$h_3(x) = a_3 \quad = \quad 4$

$h_2(x) = a_2 + h_3(x) * x \quad = \quad 5 + 4x$

$h_1(x) = a_1 + h_2(x) * x \quad = \quad 3 + (5 + 4x) * x = 3 + 5x + 4x^2$

$h_0(x) = a_0 + h_1(x) * x \quad = \quad 2 + (3 + 5x + 4x^2) * x = 2 + 3x +$

$+5x^2 + 4x^3$

Schemat Hornera

```
int horner(int n, float a[], float x)
{  float wynik = a[n];
   int i;
   for(i=n-1; i >= 0; i--) wynik = wynik * x + a[i];
   return wynik;
}
```

```
def horner(n,a, x):
    wynik = a[n]
    i=n-1
    while i>=0:
        wynik = wynik * x + a[i]
        i=i-1
    return wynik
```

Złożoność czasowa:

- Czas O(n)
- Liczba mnożeń: n

Optymalizacja/”podrasowanie” – kiedy warto

Instrukcja dominująca:
Instrukcja, której liczba wykonań jest takiego samego rzędu jak złożoność algorytmu

Optymalizujemy liczbę
- operacji dominujących, w tym …
- operacji szczególnie czasochłonnych, np.
  - operacje arytmetyczne zmiennoprzecinkowe (alg. Hornera)
  - porównania/podstawienia w algorytmach sortowania (porównujemy długie „klucze”, przestawiamy całe „opisy-rekordy”)

Funkcje f(n) i g(n) są tego samego rzędu, gdy
- f(n) = O(g(n)) oraz
- g(n) = O(f(n)).

Optymalizacja/”podrasowanie” – kiedy warto

Dokładność numeryczna innym powodem/aspektem optymalizacji:

- wpływ liczby operacji zmiennoprzecinkowych na dokładność wyniku
- wpływ kolejności operacji zmiennoprzecinkowych na dokładność wyniku

… szczegóły na przedmiocie analiza numeryczna

Instrukcje dominujące – przykład 1

Przykład

```
for(i=0; i<n; i++)
   a[i] = 0;        // nie jest dominujaca - n powtórzeń


for (i=0; i<n; i++)
   for (j=0; j<n; i++)
       b[i,j] = 0; // dominujaca - n2 powtórzeń
```

Instrukcje dominujące – przykład 2

```
void insertSort(int n,int a[])
{ int x, k, p;
  for (k=1; k<n; k++) {
    x=a[k]; //niedominująca
    p=k;
    while (p>0 && x<a[p-1])
    { a[p]=a[p-1]; // dominująca
      p--;
    }
    a[p]=x;
  }
}
```