## Zajęcia 13

**Temat:** Sortowanie 2
**Czas trwania:** 2x45 min
**Cel zajęć:**

projektuje i programuje proste problemy z różnych dziedzin, stosuje przy tym: instrukcje wejścia/wyjścia, wyrażenia arytmetyczne i logiczne, instrukcje warunkowe, instrukcje iteracyjne, tablice, rekurencję, pisze własne funkcje rekurencyjne, stosuje algorytmy sortowania, wyznacza złożoność obliczeniową, testuje poprawność programów dla różnych danych, posługuje się zintegrowanym środowiskiem programistycznym przy pisaniu, uruchamianiu i testowaniu programów;

**Efekty:**

- umie uruchomić potrzebne oprogramowanie,
- umie napisać program z wykorzystaniem algorytmu sortującego,
- zna porządek częściowy i liniowy,
- zna algorytm sortowania przez wstawianie, scalanie i szybkie.

**Formy i metody pracy:** praca samodzielna, dyskusja, omówienie

| Zadania do wykonania na zajęciach | Treści programowe |
|---|---|
| 1. Wirgiliusz | M.3, P.2.14, A.3.4 |
| 2. Układanie kart | M.3, P.2.14, A.3.4 |
| 3. Równoważne teksty (rtezad) | M.3, P.2.14, A.3.4 |

**Materiały do zajęć:**

https://www.main2.edu.pl/main2/courses/show/7/13/

**Zadania do wykonania w domu:**

**Czekolada (X OI):**

https://szkopul.edu.pl/problemset/problem/8AKFvYX1GKjeakIidXGH5_h7/site

**Lotniska (X OI):**

https://szkopul.edu.pl/problemset/problem/YtuyhoaeeDNR5zcXbKEewOdA/site

# ZADANIA I ROZWIĄZANIA:

## Zadanie 1. Wirgiliusz

Limit pamięci: 128MB

*Ojciec Wirgiliusz,*
*uczył dzieci swoje,*
*a miał ich wszystkich*
*prawdziwe roje,*
*takie malutkie,*
*i takie wielkie,*
*takie grubiutkie,*
*i takie cienkie.*

*Piosenka tradycyjna*

Wszystkie dzieci Wirgiliusza postanowiły zamieszkać w Bajtomiu w pięknych domkach na nowo wybudowanej ulicy Bajtockiej. Razem z nimi zamieszka również Wirgiliusz. Ponieważ ojciec zamierza często odwiedzać wszystkie swoje dzieci, chciałby znaleźć taki dom jednego z nich, aby mieć możliwie blisko nich wszystkich.

Ojciec Wirgiliusz chce zminimalizować sumaryczną odległość do wszystkich swoich dzieci i poprosił Ciebie, abyś napisał stosowny program.

### Wejście

Pierwszy wiersz zawiera liczbę całkowitą $r$ ($1 \le r \le 10^6$) oznaczającą liczbę dzieci Wirgiliusza. Drugi wiersz zawiera $r$ całkowitych numerów domów $s_1$, $s_2$, ..., $s_i$, ..., $s_r$, w których one mieszkają ($1 \le s_i \le 1\,000\,000\,000$). Zauważ, że różne dzieci mogą mieszkać w tych samych domach.

### Wyjście

W pierwszym i jedynym wierszu wyjścia Twój program musi wypisać minimalną sumę odległości optymalnie położonego domu Wirgiliusza od wszystkich dzieci. Odległość między dwoma domami o numerach $s_i$ i $s_j$ wynosi $d_{ij} = |s_i - s_j|$.

### Przykład

| Wejście | Wyjście |
|---|---|
| 3<br>2 4 6 | 4 |

### Rozwiązanie

Rozwiązanie niepoprawne: wybór mieszkania leżącego najbliżej średniej wszystkich numerów.

Rozwiązanie wystarczające: Prosta obserwacja prowadzi do wniosku, że najlepszym wyborem miejsca zamieszkania dla Wirgiliusza będzie mediana wszystkich numerów. Wystarczającym rozwiązaniem będzie posortowanie tablicy `s[]` z `r` numerami mieszkań z użyciem STL, a następnie obliczenie sumy wszystkich różnic między miejscem zamieszkania Wirgiliusza i pozostałymi domami (poniżej fragment kodu).

```
sort(s,s+r);
vito = r / 2;
for (int i=0; i<r; i++)
  d = d + abs( s[i] - s[vito] );
cout << d;
```

Lepszym rozwiązaniem byłoby samodzielne wyznaczenie mediany z użyciem algorytmu Hoare'a (działającym w czasie O(n)) zamiast sortowania.

## Zadanie 2. Układanie kart

Limit pamięci: 64MB

Mały Bitek dostał od rodziców talię kart. Ze zdziwieniem jednak zauważył, że karty zamiast tradycyjnych oznaczeń typu *król pik* są oznaczone kolejnymi liczbami całkowitymi. Szybko jednak bardzo mu się to spodobało.

Bitek grając w karty zawsze ustawia je od najmniejszej do największej. Stosuje przy tym popularną metodę:

- zakłada, że pierwsza karta jest już na swoim miejscu,
- każdą kolejną kartę przesuwa w lewo tak długo, aż znajdzie się ona na właściwej pozycji.

Bitek zauważył, że w trakcie porządkowania *i*-ta karta mija zawsze $k_i$ innych kart. Bardzo go to zaintrygowało i teraz chciałby wiedzieć, ile łącznie takich „minięć" zostanie wykonanych dla całej potasowanej talii.

### Wejście

W pierwszym wierszu znajduje się jedna liczba całkowita n – liczba kart ($1 \le n \le 10^6$). W kolejnej linii znajduje się n różnych liczb całkowitych $x_i$ – wartości kolejnych kart ($1 \le x_i \le n$).

### Wyjście

Jedna liczba całkowita oznaczająca łączną liczbę miniętych w trakcie układania kart.

### Przykład

| Wejście | Wejście |
|---|---|
| 4<br>4 2 3 1 | 5 |

### Rozwiązanie

Rozwiązanie wolne: Możemy zliczać wszystkie minięte elementy symulując sortowanie przez wstawianie. Takie rozwiązanie zyska 20 punktów.

```
insertion_sort (T[], n)
i ← 1
wynik ← 0
    dopóki i < n // wykonaj n-1 krotnie
    j ← i - 1   // liczby do i-1 włącznie są uporządkowane
```

```
    k ← T[i]    // zapamiętaj i-tą liczbę do wstawienia
    // dopóki nie dojdziesz do początku ciągu
    // i element badany jest większy niż element wstawiany
    dopóki j ≥ 0 i T[j] > k
        T [j + 1] ← T [j] // przesuń badany element o jeden w prawo
        j ← j - 1
        wynik ← wynik + 1 // zapamiętaj kolejne przesunięcie
        // element wstawiany zapisz na ostatniej zapamiętanej
pozycji
    T [j + 1] ← k
    i = i + 1
zwróć wynik
```

Rozwiązanie szybkie. Przeanalizujmy proces sortowania przez zliczanie. Zwróćmy przede wszystkim uwagę na proces scalania dwóch posortowanych ciągów. Jeżeli założymy, że scalane ciągi znajdują się w sortowanej tablicy bezpośrednio po sobie, to zabranie elementu z pierwszego z nich do sortowanej tablicy nie spowoduje „minięcia" żadnego z elementów drugiej tablicy. Z kolei element zabrany z drugiego ciągu mija jednocześnie wszystkie nie zabrane jeszcze elementy z ciągu pierwszego. Dzięki temu uzyskamy złożoność sortowania przez scalanie: O(*n* log *n*).

```
scal(t[],P,L)
  //przepisz dane do tablicy pomocniczej
  dla i=P, P+1, …, L wykonaj pom[i] ← t[i]
  //znajdź środek
  s ← (P+L)/2
  //zacznij od początku lewego i prawego ciągu
  i ← L, j ← s+1, k ← L
  //dopóki nie skończy się lewy lub prawy ciąg
  dopóki i ≤ s i j ≤ P
    //wybierz mniejszy element z prawego lub lewego ciągu
    //i przepisz go do tablicy wynikowej
    jeżeli pom[i] ≤ pom[j]
      t[k] ← pom[i], i←i+1,
    przeciwnie
      t[k] ←pom[j],j←j+1
      // policz elementy, które mija w tym momencie pom[j]
      policz ← policz + (s-i+1)
    k ← k+1
  //przepisz pozostałe elementy
  dopóki i ≤ s
    t[k] ← pom[i], i←i+1, k ← k+1
  dopóki j ≤ P
    t[k] ←pom[j], j←j+1, k ← k+1
```

Główna część algorytmu merge-sort:

```
mergesort(L, P)
jeśli jest więcej niż jedna liczba do posortowania
znajdź środek
posortuj lewą część
posortuj prawą część
scal posortowane ciągi
```

## Zadanie 3. Równoważne teksty

Dostępna pamięć: 256MB

Dwa teksty *a* oraz *b* nazwiemy równoważnymi, jeżeli spełniają jeden z dwóch warunków:

1. są identyczne,
2. jeżeli tekst *a* podzielimy na dwie równej długości części $a_1$ i $a_2$, a tekst *b* podzielimy podobnie na teksty $b_1$ i $b_2$, to jeden z warunków będzie poprawny:
   - $a_1$ i $b_1$ oraz $a_2$ i $b_2$ będą równoważne

     *lub*
   - $a_1$ i $b_2$ oraz $a_2$ i $b_1$ będą równoważne.

Sprawdź, czy podane dwa teksty są równoważne według powyższej definicji!

### Wejście

W dwóch wierszach wejścia znajduje się po jednym tekście. Teksty mają długość od 1 do $2 \cdot 10^5$ znaków i składają się wyłącznie z małych liter alfabetu łacińskiego. Długości tekstów są równe.

### Wyjście

Wypisz jedno słowo: TAK lub NIE, odpowiedź na pytanie, czy podane teksty są równoważne.

### Przykład

| Dla danych wejściowych | poprawnym wynikiem jest |
|---|---|
| `abcd`<br>`cdba` | `TAK` |

| Dla danych wejściowych | poprawnym wynikiem jest |
|---|---|
| `abcd`<br>`acbd` | `NIE` |

Objaśnienie przykładu 1

Tekst abcd rozkładamy na ab i cd. Tekst cdba rozkładamy na cd i ba. W tym wypadku cd jest równoważne cd. Postępując tą samą metodą stwierdzimy, że ab i ba są równoważne.

### Rozwiązanie

Zauważmy, że znacznie łatwiej będzie nam stwierdzić, czy dwa teksty o parzystej długości są równoważne, jeśli ich porównywane części weźmiemy do porównania w kolejności leksykograficznej. W tym wypadku po prostu powinny być sobie równe. To samo możemy zrobić rekurencyjnie dla ich podciągów. Spróbujmy więc uporządkować oba porównywane ciągi w opisany sposób i sprawdzić, czy po porównaniu otrzymujemy identyczne ciągi.

W poniższym algorytmie dla uproszczenia wykorzystana została funkcja `substr` oraz `min` z biblioteki STL.

```
sortuj (s[])
      n ← |s[]|
      jeżeli (n mod 2 = 1)
            zwróć s[]
      tekst x ← sortuj(s.substr (0, n/2))
      tekst y ← sortuj(s.substr (n/2))
      zwróć min (x+y, y+x)
```

Główna część programu:

```
tekst a, b
wczytaj (a, b)
jeżeli sortuj(a) = sortuj(b)
      wypisz „TAK”
przeciwnie
      wypisz „NIE”
```