Sortowanie

Sortowanie danych jest jednym z podstawowych problemów programowania komputerów, z którym prędzej czy później spotka się każdy programista.

sport - wyniki uzyskane przez poszczególnych zawodników należy ułożyć w określonej kolejności, aby wyłonić zwycięzcę oraz podać lokatę każdego zawodnika.

bank - spłaty kredytów należy ułożyć w odpowiedniej kolejności, aby wiadomo było kto i kiedy ma płacić odsetki do banku.

grafika - wiele algorytmów graficznych wymaga porządkowania elementów, np. ścian obiektów ze względu na odległość od obserwatora. Uporządkowanie takie pozwala później określić, które ze ścian są zakrywane przez inne ściany dając w efekcie obraz trójwymiarowy.

bazy danych - informacja przechowywana w bazie danych może wymagać różnego rodzaju uporządkowania, np. lista książek może być alfabetycznie porządkowana wg autorów lub tytułów, co znacznie ułatwia znalezienie określonej pozycji.

Sortowanie

Sortowanie ma na celu stworzyć zbiór uporządkowany.

Zbiór uporządkowany to taki, w którym elementy występują w określonym porządku, czyli kolejności. Jeśli elementy są liczbami, to porządek może być:

rosnący - każdy następny element zbioru jest większy od swojego poprzednika. W zbiorze nie ma elementów równych. Taki porządek nazywamy porządkiem mocnym.

Np. {0 2 4 7 12 13 16 74 125 200 333 ...}

malejący - każdy następny element jest mniejszy od swojego poprzednika. W zbiorze nie ma elementów równych - porządek mocny.

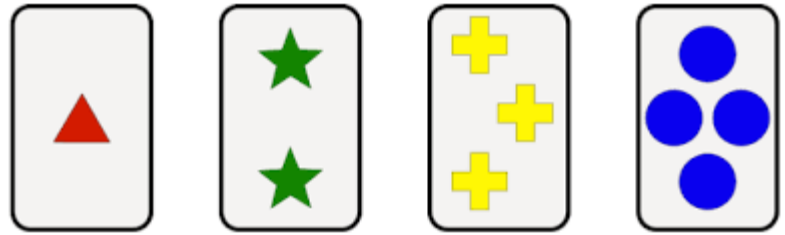
Np. (100 96 64 32 15 10 8 4 2 -4 -9 -22 ... }

niemalejący - każdy następny element jest równy lub większy od swojego poprzednika. Elementy o tych samych wartościach mogą występować wielokrotnie obok siebie. Taki porządek nazywamy porządkiem słabym.

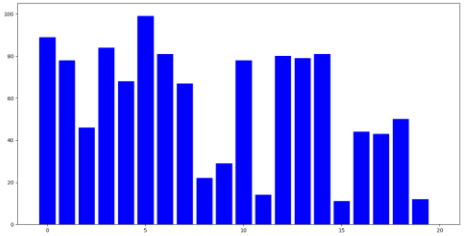
Np. { 1 1 1 1 1 3 4 4 5 8 12 12 12 33 40 ... }

nierosnący - każdy następny element jest równy lub mniejszy od swojego poprzednika. Elementy o tych samych wartościach mogą się powtarzać - porządek słaby.

Np. {100 95 75 75 75 40 23 12 12 12 12 7 6 4 2 2 2 1 1 1 ... }



Sortowanie



Złożoność obliczeniowa

O(n) - Algorytm o liniowej zależności czasu wykonania od ilości danych. Dwukrotny wzrost liczby przetwarzanych danych powoduje dwukrotny wzrost czasu wykonania. Tego typu złożoność powstaje, gdy dla każdego elementu należy wykonać stałą liczbę operacji.

O(n^2) - Algorytm, w którym czas wykonania rośnie z kwadratem liczby przetwarzanych elementów. Dwukrotny wzrost liczby danych powoduje czterokrotny wzrost czasu wykonania. Tego typu złożoność powstaje, gdy dla każdego elementu należy wykonać ilość operacji proporcjonalną do liczby wszystkich elementów.

O(n log n) - Dobre algorytmy sortujące mają taką właśnie złożoność obliczeniową. Czas wykonania przyrasta dużo wolniej od wzrostu kwadratowego. Tego typu złożoność powstaje, gdy zadanie dla n elementów można rozłożyć na dwa zadania zawierające po połowie elementów.

O(n!) - Bardzo pesymistyczny algorytmy, czas wykonania rośnie szybko ze wzrostem liczby elementów wejściowych, czyli znalezienie rozwiązania może zająć najszybszym komputerom całe wieki lub tysiąclecia. Takich algorytmów należy unikać.

Sortowanie zwariowane

$$P = \frac{1}{4!} = \frac{1}{24} \text{ dla 4 elementów}$$

$$P = \frac{1}{10!} = \frac{1}{3628800} \text{ dla 10 elementów}$$

$$P = \frac{1}{100!} = \frac{1}{9,3 \cdot 10^{157}} \text{ dla 100 elementów}$$

Opiera się na dosyć zwariowanych zasadach. Jego działanie możemy scharakteryzować na przykładzie układania talii kart. Bierzemy talię kart. Sprawdzamy, czy jest ułożona. Jeśli nie, tasujemy ją i znów sprawdzamy ułożenie. Operacje sprawdzania i tasowania wykonujemy dotąd, aż talia nam się ułoży w pożądanej kolejności kart.

Nic nie sortujemy, wręcz dokonujemy operacji odwrotnej - tasowania, a talia może zostać posortowana. Dlaczego? Wynika to z praw rachunku prawdopodobieństwa. Otóż tasowanie powoduje, iż karty przyjmują losowe permutacje swoich położeń. Ponieważ każda permutacja zbioru kart jest równie prawdopodobna zatem możemy też otrzymać układ uporządkowany. Oczywiście wynik taki pojawia się dosyć bardzo rzadko. Nie poleca się sortowania tą metodą zbiorów liczniejszych niż 9 elementów. Złożoność O(n!)

Sortowanie bąbelkowe

Algorytm sortowania bąbelkowego jest jednym z najstarszych algorytmów sortujących. Można go potraktować jako ulepszenie algorytmu sortowania głupiego. Zasada działania opiera się na cyklicznym porównywaniu par sąsiadujących elementów i zamianie ich kolejności w przypadku niespełnienia kryterium porządkowego zbioru. Operację tę wykonujemy dotąd, aż cały zbiór zostanie posortowany.

Algorytm sortowania bąbelkowego przy porządkowaniu zbioru nieposortowanego ma klasę czasowej złożoności obliczeniowej równą $O(n^2)$

https://www.youtube.com/watch?v=lyZQPjUT5B4
Taniec węgierski

https://www.youtube.com/watch?v=h-titXlHGBI
Taniec uczniów z Chorzowa

Sortowanie głupie

Sortowanie głupie jest również bardzo złym algorytmem sortującym, lecz, w przeciwieństwie do sortowania zwariowanego, daje zawsze poprawne wyniki. Zasada działania jest bardzo prosta:

Przeglądamy kolejne pary sąsiednich elementów sortowanego zbioru. Jeśli bieżąco przeglądana para elementów jest w złej kolejności, elementy pary zamieniamy miejscami i całą operację rozpoczynamy od początku zbioru.
Jeśli przeglądniemy wszystkie pary i nie wystąpi zamiana, to zbiór będzie posortowany i algorytm może zakończyć działanie.
Głupota algorytmu wyraża się tym, iż po napotkaniu nieposortowanych elementów algorytm zamienia je miejscami, a następnie rozpoczyna całą pracę od początku zbioru.
Złożoność obliczeniowa algorytmu przy sortowaniu zbioru nieuporządkowanego ma klasę $O(n^3)$.

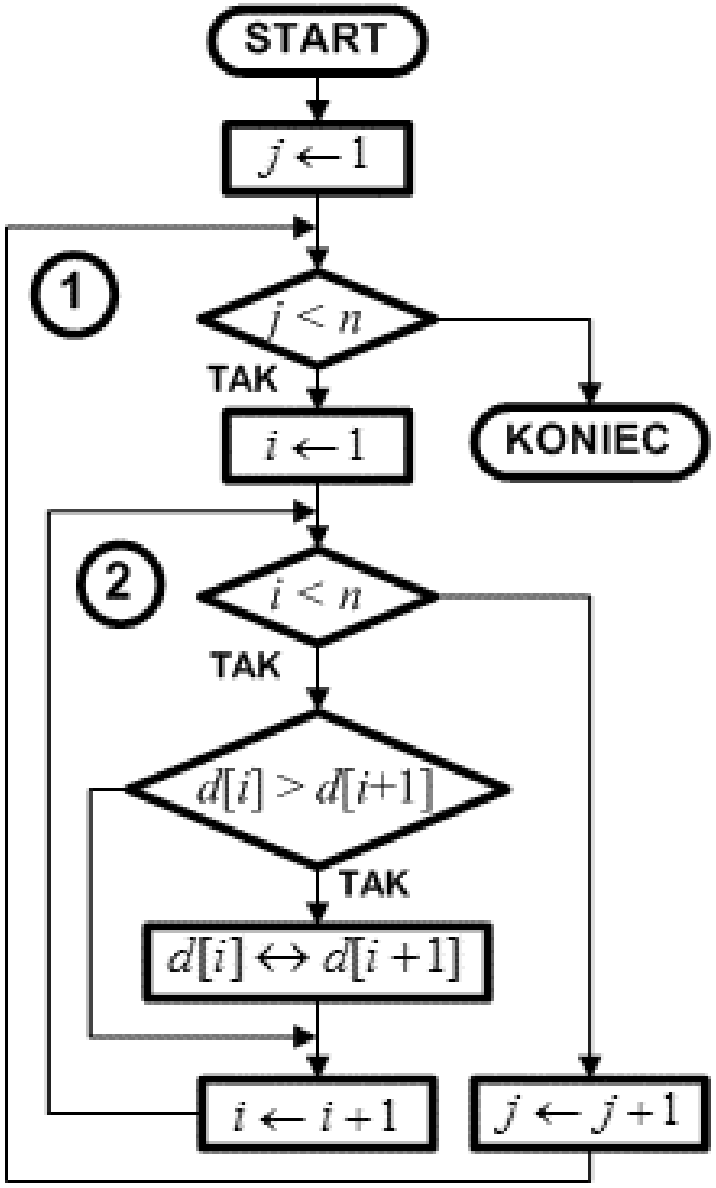
Sortowanie bąbelkowe

Obieg	Zbiór	Opis operacji
1	**5 4** 3 2 1	Rozpoczynamy od pierwszej pary, która wymaga wymiany elementów
	4 **5 3** 2 1	Druga para też wymaga zamiany elementów
	4 3 **5 2** 1	Wymagana wymiana elementów
	4 3 2 **5 1**	Ostatnia para również wymaga wymiany elementów
	4 3 2 **1 5**	Stan po pierwszym obiegu. Najstarszy element (5) znalazł się na końcu zbioru, a najmłodszy (1) przesunął się o jedną pozycję w lewo.
2	**4 3** 2 1 5	Para wymaga wymiany
	3 **4 2** 1 5	Para wymaga wymiany
	3 2 **4 1** 5	Para wymaga wymiany
	3 2 1 **4 5**	Elementy są w dobrej kolejności, zamiana nie jest konieczna.
	3 2 **1** 4 5	Stan po drugim obiegu. Najmniejszy element (1) znów przesunął się o jedną pozycję w lewo. Z obserwacji tych można wywnioskować, iż po każdym obiegu najmniejszy element wędruje o jedną pozycję ku początkowi zbioru. Najstarszy element zajmuje natomiast swe miejsce końcowe.
3	**3 2** 1 4 5	Para wymaga wymiany
	2 **3 1** 4 5	Para wymaga wymiany
	2 1 **3 4** 5	Dobra kolejność
	2 1 3 **4 5**	Dobra kolejność
	2 1 3 4 5	Stan po trzecim obiegu. Wnioski te same.
4	**2 1** 3 4 5	Para wymaga wymiany
	1 **2 3** 4 5	Dobra kolejność
	1 2 **3 4** 5	Dobra kolejność

Sortowanie bąbelkowe

Kroki	Sortowana tablica			
0	6	4	8	1
1	6	4	8	1
2	4	6	8	1
3	4	6	8	1
4	4	6	8	1
5	4	6	8	1
6	4	6	1	8
7	4	6	1	8
8	4	6	1	8
9	4	6	1	8
10	4	1	6	8
11	4	1	6	8
12	1	4	6	8
13	1	4	6	8
14	1	4	6	8

Sortowanie bąbelkowe



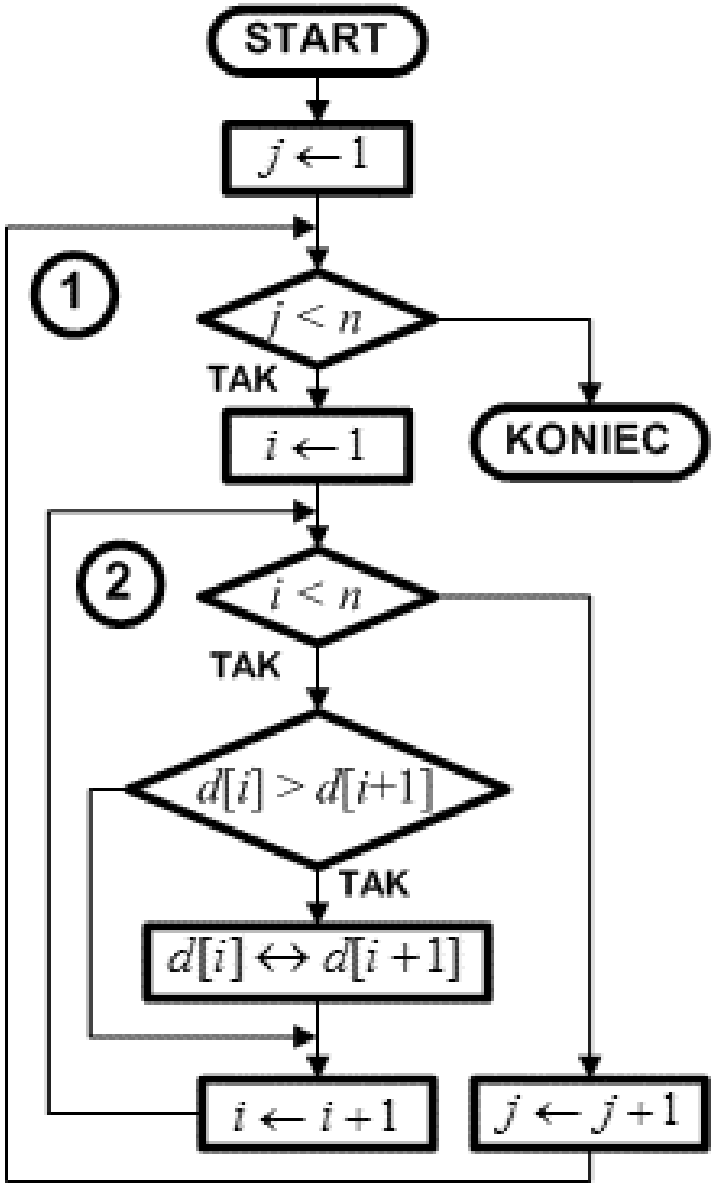
Cz.1

```
#include <cmath>
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <time.h>

using namespace std;

const int N = 20; // Liczebność zbioru.

// Program główny
//---------------

int main()
{
  int d[N],i,j;
```

Cz.2

```
// Najpierw wypełniamy tablicę d[] liczbami pseudolosowymi
// a następnie wyświetlamy jej zawartość

  srand((unsigned)time(NULL));

  for(i = 0; i < N; i++) d[i] = rand() % 100;
  for(i = 0; i < N; i++) cout << setw(4) << d[i];
  cout << endl;

// Sortujemy

  for(j = 0; j < N - 1; j++)
    for(i = 0; i < N - 1; i++)
      if(d[i] > d[i + 1]) swap(d[i], d[i + 1]);

// Wyświetlamy wynik sortowania

  cout << "Po sortowaniu:\n\n";
  for(i = 0; i < N; i++) cout << setw(4) << d[i];
  cout << endl;
  return 0;
}
```

Sortowanie bąbelkowe



Sortowanie wykonywane jest w dwóch zagnieżdżonych pętlach. Pętla zewnętrzna nr 1 kontrolowana jest przez zmienną j. Wykonuje się ona n - 1 razy. Wewnątrz pętli nr 1 umieszczona jest pętla nr 2 sterowana przez zmienną i. Wykonuje się również n - 1 razy.

Sortowanie odbywa się wewnątrz pętli nr 2. Kolejno porównywany jest i-ty element z elementem następnym. Jeśli elementy te są w złej kolejności, to zostają zamienione miejscami. W tym miejscu jest najważniejsza różnica pomiędzy algorytmem sortowania bąbelkowego a algorytmem sortowania głupiego. Ten drugi w momencie napotkania elementów o złej kolejności zamienia je miejscami i rozpoczyna cały proces sortowania od początku. Algorytm sortowania bąbelkowego wymienia miejscami źle ułożone elementy sortowanego zbioru i przechodzi do następnej pary zwiększając indeks i o 1.

Anagram

Anagram jest to wyraz, wyrażenie lub zdanie utworzone przez przestawienie liter lub sylab innego wyrazu, wyrażenia lub zdania. Warunkiem utworzenia anagramu jest wykorzystanie wszystkich liter lub sylab innego wyrazu (zdania) wyjściowego.

tyran-narty
klisza-szalik
wyżły-łyżwy
paczka-czapka
niska-sanki
matura – trauma
adam – dama
arbuz - burza

Najsłynniejszym przykładem anagramu, który jest całym zdaniem, jest pytanie, jakie skierował Piłat do Jezusa i odpowiedź, jakiej ten mu udzielił:
– Quid est veritas? („Co to jest prawda?”)
– Vir est qui adest („Człowiek, który jest przed tobą”).

Anagramy

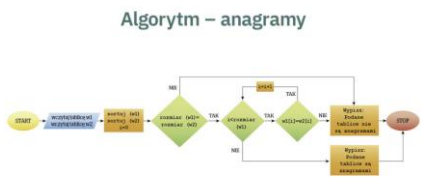
Ułóż anagramy dla podanych wyrazów.

ALERGIA

BRUDAS

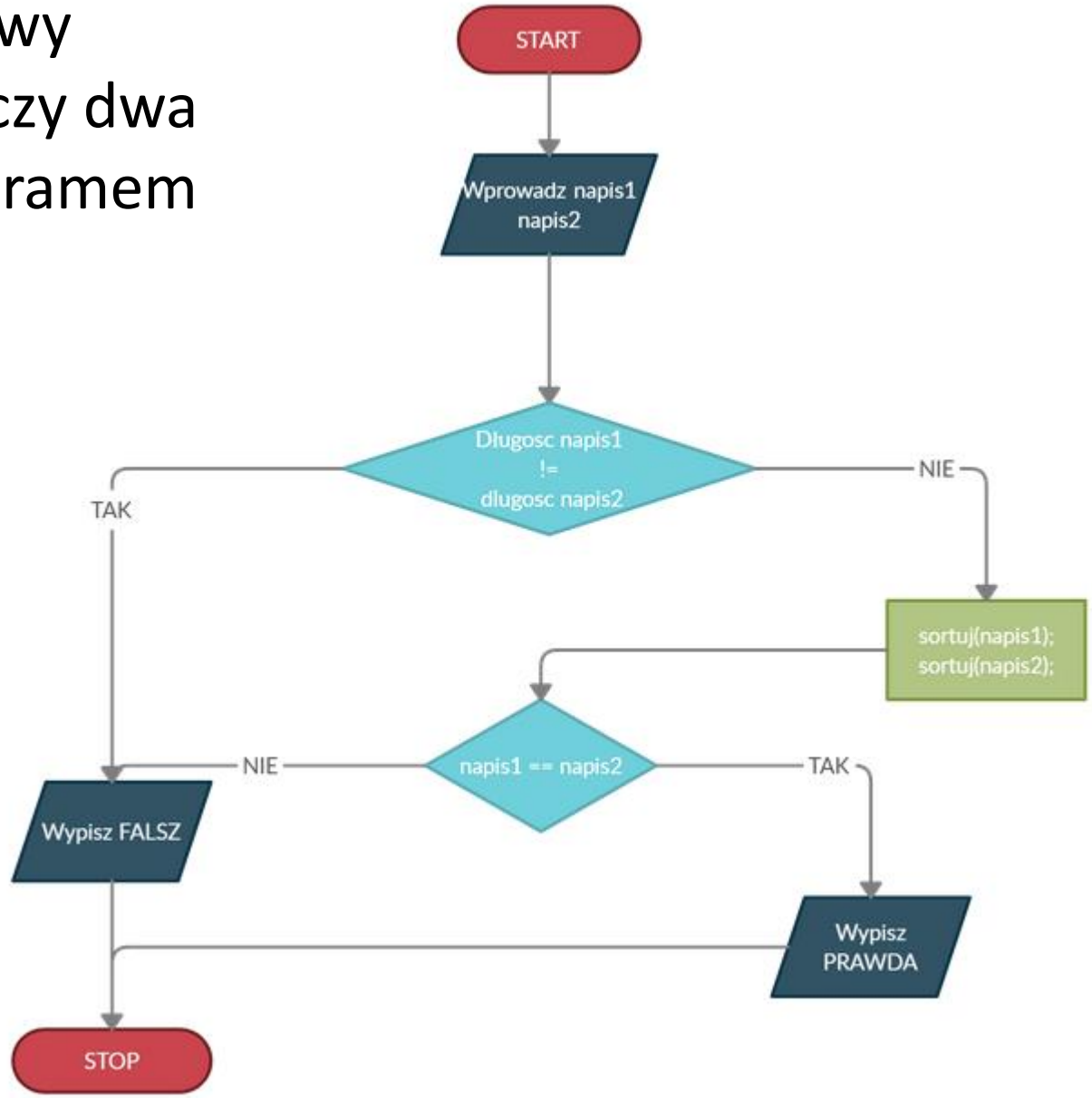
BANDZIOR

Napisz swoje imię i postaraj się ułożyć z jego liter jak najwięcej wyrazów.
Wczytaj stronę https://wordlist.eu/anagramy/
Sprawdź, czy udało Ci się znaleźć wszystkie anagramy Twojego imienia. Baza wyrazów może Ci się przydać np. do gry w Scrabble.



Schemat blokowy sprawdzający, czy dwa wyrazy są anagramem

Jeśli chcemy się dowiedzieć, czy dany wyraz jest anagramem innego wyrazu musimy sprawdzić, czy długość obu wyrazów jest identyczna. Następnie je posortować alfabetycznie i sprawdzić, czy tworzą identyczne łańcuchy znaków.



```
#include <iostream>
using namespace std;

bool anagram(string a, string b)
{
if(a.length() != b.length())
return false;
int La[26], Lb[26];
fill(La, La + 26, 0);
fill(Lb, Lb + 26, 0);
for(int i = 0; i < a.length(); i++)
{
La[a[i] - 'a']++;
Lb[b[i] - 'a']++;

}
for(int k = 0; k < 26; k++)
if(La[k] != Lb[k])
return false;
return true;
}

int main()
{
string a, b;
cout << "Podaj dwa slowa po spacji" << endl;
cin >> a >> b;
if(anagram(a, b))
cout << "TAK" << endl;
else
cout << "NIE" << endl;
}
```

Program sprawdzając, czy podane słowa są anagramami