Algorytmy sortujące. Klasyczne sortowanie bąbelkowe

https://eduinf.waw.pl/inf/alg/003_sort/m0003.php

Procesy sortowania zbiorów są jednymi z częstszych problemów rozważanych w naukach algorytmizacji. My także rozpoczniemy od nich i na wstępie omówimy niektóre wersje dosyć nieoptymalnego sortowania bąbelkowego.

Sortowanie bąbelkowe jest jednym z prostszych w implementacji algorytmów sortowania. Swoją nazwę zawdzięcza temu, że w przypadku pionowego przedstawienia zbioru danych, element najmniejszy (o najmniejszej masie) wypływa do góry.

Algorytm **sortowania bąbelkowego** jest jednym z najstarszych algorytmów sortujących. 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.

Uwaga: Algorytm sortowania bąbelkowego jest uważany za bardzo zły algorytm sortujący. Można go stosować tylko dla niewielkiej liczby elementów w sortowanym zbiorze (do około 5000). Przy większych zbiorach czas sortowania może być zbyt długi.

Rozpoczniemy od stworzenia algorytmu w schemacie blokowym.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Algorytm polega na porządkowaniu dwóch sąsiednich elementów zbioru.

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. W efekcie algorytm wykonuje w sumie:

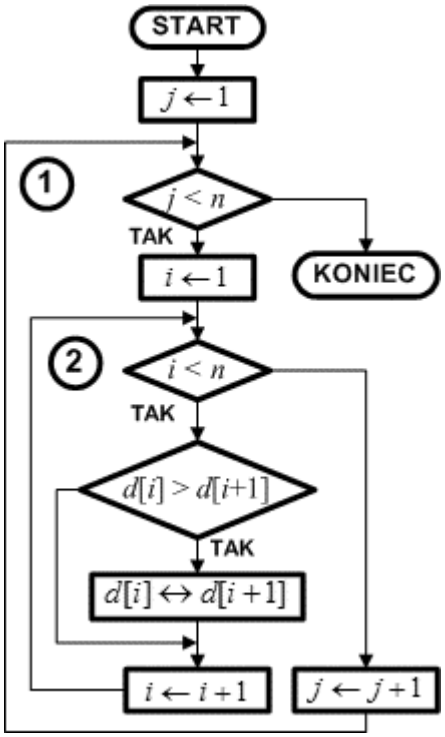
$$T_1(n) = (n - 1)^2 = n^2 - 2n + 1$$

obiegów pętli wewnętrznej, po których zakończeniu zbiór zostanie posortowany.

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.

Algorytm sortowania bąbelkowego przy porządkowaniu zbioru nieposortowanego ma klasę czasowej złożoności obliczeniowej równą $O(n^2)$. Sortowanie odbywa się w miejscu.

Schemat blokowy



Mając **algorytm problemu**, bez wysiłku stworzyć można odpowiedni **kod programu.** Zrobimy to w znanym nam języku C.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór n-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór n-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, i, j Є N;

tymcz - zmienna przechowująca element przestawiany.

Przykład kodu sortowania bąbelkowego

```
// Sortowanie bąbelkowe. Wersja 1
#include <stdio.h>
#include <stdlib.h>
const int N = 20; // Liczebnosc zbioru.

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

  printf(" Sortowanie babelkowe\n       WERSJA NR 1\n\n")          ;

// Najpierw wypelniamy tablice d[] liczbami pseudolosowymi
// a nastepnie wyswietlamy jej zawartosc

  for(i = 1; i <= N; i++) d[i] = rand() % 100;

// wyswietlamy tablice
   printf ("Tablica nieposortowana\n\n   i   d[i]  \n");
   for(i = 1; i <= N; i++)
   printf ("  %d %d \n",i, d[i]);

// Sortujemy
  for(j = 1; j<= N ;  j++)
    for(i = 1; i <= N; i++)
      if(d[i] > d[i + 1]) {
      tymcz=d[i];
      d[i]=d[i+1];
      d[i+1]= tymcz;
      }
// Wyswietlamy wynik sortowania

// wyswietlamy tablice
  printf ("\n\nTablica posortowana\n  i  d[i]\n");
   for(i = 1; i <= N; i++)
   printf ("  %d %d\n",i,d[i]);
  return 0;
}
```

Wynik wykonania kodu

Sortowanie babelkowe

WERSJA NR 1

Tablica nieposortowana	Tablica posortowana
i d[i]	i d[i]
1 41	1 0
2 67	2 5
3 34	3 24
4 0	4 27
5 69	5 27
6 24	6 34
7 78	7 36
8 58	8 41
9 62	9 42
10 64	10 45
11 5	11 58
12 45	12 61
13 81	13 62
14 27	14 64
15 61	15 67
16 91	16 69
17 95	17 78
18 42	18 81
19 27	19 91
20 36	20 95

Process returned 0 (0x0) execution time : 0.031 s

Press any key to continue.

Sortowanie bąbelkowe - wersja nr 2

Podany w poprzednim wykładzie algorytm sortowania bąbelkowego można zoptymalizować pod względem czasu wykonania. Jeśli przyjrzymy się dokładnie obiegom wykonywanym w tym algorytmie, to zauważymy bardzo istotną rzecz:

Przykład:

Wykonamy jeden obieg sortujący dla zbioru pięcioelementowego

{ **9** 3 1 7 0 }. Elementem największym jest pierwszy element - liczba 9.

Obieg nr 1

9 3 1 7 0 Zamiana
3 **9 1** 7 0 Zamiana
3 1 **9 7** 0 Zamiana
3 1 7 **9 0** Zamiana
3 1 7 0 **9** Koniec obiegu.

Widać, że po wykonaniu pełnego obiegu w algorytmie sortowania bąbelkowego najstarszy element wyznaczony przez przyjęty porządek zostaje umieszczony na swoim właściwym miejscu - na końcu zbioru.

Wniosek ten jest oczywisty. W każdej kolejnej parze porównywanych elementów element starszy przechodzi na drugą pozycję. W kolejnej parze jest on na pierwszej pozycji, a skoro jest najstarszym, to po porównaniu znów przejdzie na pozycję drugą itd. - jest jakby ciągnięty na koniec zbioru (jak bąbelek powietrza wypływający na powierzchnię wody).

Co z tego wynika dla nas? Otóż po każdym obiegu na końcu zbioru tworzy się podzbiór uporządkowanych najstarszych elementów. Zatem w kolejnych obiegach możemy pomijać sprawdzanie ostatnich elementów - liczebność zbioru do posortowania z każdym obiegiem maleje o 1.

Przykład:

Dokończmy sortowania podanego powyżej zbioru uwzględniając podane przez nas fakty. Po pierwszym obiegu na końcu zbioru mamy umieszczony element najstarszy. W drugim obiegu będziemy zatem sortować zbiór 4 elementowy, w trzecim obiegu 3 elementowy i w obiegu ostatnim, czwartym - zbiór 2 elementowy.

Obieg nr 2

3 1 7 0 **9** Zamiana
1 **3 7** 0 **9** Dobra kolejność
1 3 **7 0 9** Zamiana
1 3 0 **7 9** Koniec obiegu.

Obieg nr 3

1 3 0 **7 9** Brak zamiany
1 **3 0 7 9** Zamiana
1 0 **3 7 9** Koniec obiegu.

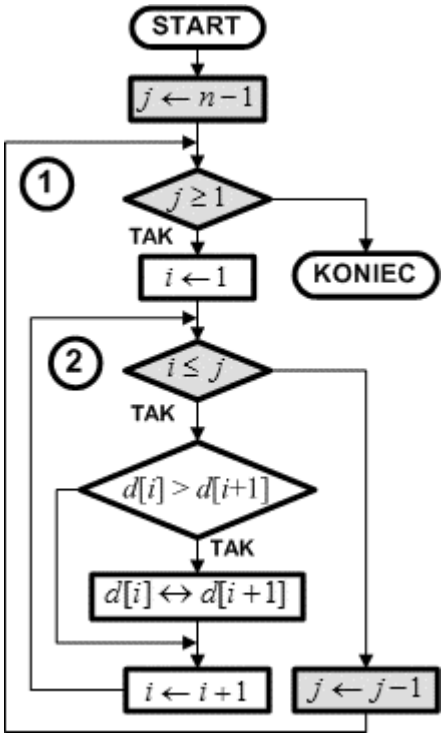
Obieg nr 4

1 0 3 7 9 Zamiana
0 1 3 7 9 Koniec ostatniego obiegu.

Zbiór jest posortowany.

W porównaniu do z poprzednim algorytmem nawet wzrokowo możemy zauważyć istotne zmniejszenie ilości niezbędnych operacji.

Schemat blokowy



Zmiany w stosunku do poprzedniej wersji są następujące:

- pętla zewnętrzna nr 1 zlicza obiegi wstecz, tzn. pierwszy obieg ma numer n-1. Dzięki takiemu podejściu w zmiennej j mamy zawsze numer ostatniego elementu, do którego ma dojść pętla wewnętrzna nr 2. Ta zmiana wymaga również odwrotnej iteracji zmiennej j.

- pętla wewnętrzna sprawdza w warunku kontynuacji, czy wykonała j obiegów, a nie jak poprzednio n-1 obiegów. Dzięki temu po każdym obiegu pętli nr 1 (zewnętrznej) pętla nr 2 będzie wykonywać o jeden obieg mniej.

Pozostała część algorytmu nie jest zmieniona - w pętli wewnętrznej nr 2 sprawdzamy, czy element $d[i]$ jest w złej kolejności z elementem $d[i+1]$. Sprawdzany warunek spowoduje posortowanie zbioru rosnąco.

Przy sortowaniu malejącym zmieniamy relację większości na relację mniejszości. Jeśli warunek jest spełniony, zamieniamy miejscami element $d[i]$ z elementem $d[i+1]$, po czym kontynuujemy pętlę nr 2 zwiększając o 1 indeks i.

Po każdym zakończeniu pętli nr 2 indeks j jest zmniejszany o 1. Ilość obiegów pętli wewnętrznej wynosi:

$$T_2(n) = (n-1) + (n-2) + \ldots + 2 + 1$$

$$T_2(n) = \frac{n(n-1)}{2}$$

$$T_2(n) = \frac{1}{2}\left(n^2 - n\right)$$

Otrzymane wyrażenie ma wciąż kwadratową klasę złożoności obliczeniowej, jednakże T2(n) < T1(n) dla n > 1. Osiągnęliśmy zatem większą efektywność działania dzięki wprowadzonym zmianom.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, n Є N

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, i, j Є N

tymcz - zmienna do przechowywania tymczasowego zmienianego elementu

Lista kroków

K01: Dla $j = n-1, n-2, \ldots, 1$, wykonuj K02

K02: Dla $i = 1, 2, \ldots, j$,

jeśli $d[i] > d[i+1]$, to $d[i] \leftrightarrow d[i+1]$

K03: Koniec

Wersja nr 2 sortowania bąbelkowego

Sortowanie bąbelkowe - wersja nr 3

Algorytm **sortowania bąbelkowego** wykonuje dwa rodzaje operacji:

- test bez zamiany miejsc elementów
- test ze zamianą miejsc elementów.

Pierwsza z tych operacji nie sortuje zbioru, jest więc **operacją pustą**. Druga operacja dokonuje faktycznej zmiany porządku elementów, jest zatem **operacją sortującą**.

Ze względu na przyjęty sposób sortowania algorytm bąbelkowy zawsze musi wykonać tyle samo operacji sortujących. Tego nie możemy zmienić. Jednakże możemy wpłynąć na eliminację operacji pustych. W ten sposób usprawnimy działanie algorytmu.

Jeśli dokładnie przyjrzałeś się wersjom 1 i 2, o powinieneś dokonać następujących spostrzeżeń:

1. Wersja pierwsza jest najmniej optymalną wersją algorytmu bąbelkowego. Wykonywane są wszystkie możliwe operacje sortujące i puste.
2. Wersja druga redukuje ilość operacji pustych poprzez ograniczanie liczby obiegów pętli wewnętrznej (sortującej).

Możliwa jest dalsza redukcja operacji pustych, jeśli będziemy sprawdzać, czy w pętli wewnętrznej były przestawiane elementy (czyli czy wykonano operacje sortujące). Jeśli nie, to zbiór jest już posortowany (dlaczego?) i możemy zakończyć pracę algorytmu.

Teraz rośnie trudność wyznaczenia **czasowej złożoności obliczeniowej**, ponieważ ilość faktycznie wykonywanych operacji porównań i przestawień zależy od rozkładu elementów w sortowanym zbiorze. Zadanie komplikuje dodatkowo fakt, iż operacja pusta jest zwykle wykonywana kilkakrotnie szybciej od operacji sortującej.

Na pewno można powiedzieć, iż dla zbioru posortowanego algorytm wykona tylko $n - 1$ operacji pustych, zatem w przypadku najbardziej optymistycznym czasowa złożoność obliczeniowa redukuje się do klasy $O(n)$ - liniowej. W przypadku najbardziej niekorzystnym algorytm wykona wszystkie operacje puste i sortujące, zatem będzie posiadał klasę czasowej złożoności obliczeniowej $O(n^2)$.

Przykład:

Posortujmy zbiór { 3 1 0 7 9 } zgodnie z wprowadzoną modyfikacją.

Obieg nr 1

3 1 0 7 9 Zamiana
1 **3 0** 7 9 Zamiana
1 0 **3 7** 9 Brak zamiany
1 0 3 **7 9** Brak zamiany
1 0 3 7 **9** Koniec pierwszego obiegu.

Ponieważ były przestawienia elementów, sortowanie kontynuujemy

Obieg nr 2

1 0 3 7 9 Zamiana
0 **1 3** 7 9 Brak zamiany
0 1 **3 7 9** Brak zamiany
0 1 3 **7 9** Koniec drugiego obiegu.

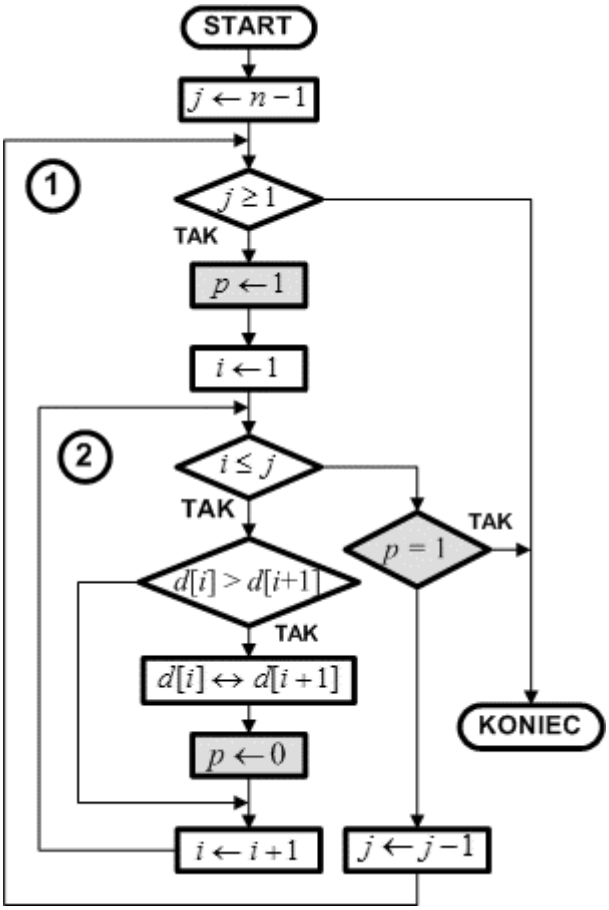
Było przestawienie elementów, sortowanie kontynuujemy

Obieg nr 3

0 1 3 7 9 Brak zamiany
0 **1 3 7 9** Brak zamiany
0 1 3 7 9 Koniec trzeciego obiegu.

Nie było przestawień elementów, kończymy sortowanie. Wykonaliśmy o 1 obieg sortujący mniej.

Schemat blokowy



Wprowadzona do algorytmu sortowania bąbelkowego modyfikacja ma na celu wykrycie posortowania zbioru. Zmiany zaznaczyliśmy blokami o odmiennym kolorze.

Zbiór będzie posortowany, jeśli po wykonaniu wewnętrznego obiegu sortującego nie wystąpi ani jedno przestawienie elementów porządkowanego zbioru.

Przed wejściem do pętli sortującej nr 2 ustawiamy zmienną pomocniczą p. Jeśli w pętli zajdzie potrzeba przestawienia elementów, to zmienna p jest zerowana. Po wykonaniu pętli sortującej sprawdzamy, czy zmienna p jest ustawiona. Jeśli tak, to przestawienie elementów nie wystąpiło, zatem kończymy algorytm. W przeciwnym razie wykonujemy kolejny obieg pętli nr 1.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, n Є N

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, i, j Є N

tymcz - zmienna do przechowywania tymczasowego zmienianego elementu

p - znacznik zamiany miejsc elementów w zbiorze. p Є N

Lista kroków

K01: Dla $j = n-1, n-2, ..., 1$, wykonuj K02...K04

K02: $p \leftarrow 1$

K03: Dla $i = 1, 2, ..., j$,

jeśli $d[i] > d[i+1]$, to

$d[i] \leftrightarrow d[i+1]$

$p \leftarrow 0$

K04: Jeśli $p = 1$, to koniec

K04: Koniec

Kod programu

```
// Sortowanie B¹belkowe - Wersja nr 3
//--------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>

const int N = 20; // Liczebnosc zbioru.

// Program glówny
//---------------

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

  printf(" Sortowanie babelkowe\n       WERSJA NR 2\n\n")          ;

// Najpierw wypelniamy tablice d[] liczbami pseudolosowymi
// a nastepnie wyswietlamy jej zawartosc

  for(i = 1; i <= N; i++) d[i] = rand() % 100;

// wyswietlamy tablice
   printf ("Tablica nieposortowana\n\n   i   d[i]  \n");
   for(i = 1; i <= N; i++)
   printf ("  %d %d \n",i, d[i]);
```

```
// Sortujemy

  for(j = N -1; j >= 1 ;  j--)
  {
       p=1;   //wskaznik przestawień

      for(i = 1; i <= j; i++)
      if(d[i] > d[i + 1]) {
      tymcz=d[i];
      d[i]=d[i+1];
      d[i+1]= tymcz;
      p = 0;
      }
      if(p) break;
  }
// Wyœwietlamy wynik sortowania

// wyswietlamy tablice
  printf ("\n\nTablica posortowana\n  i  d[i]\n");
   for(i = 1; i <= N; i++)
   printf ("  %d %d\n",i,d[i]);
  return 0;
}
```

Sortowanie bąbelkowe - wersja nr 4

Czy algorytm **sortowania bąbelkowego** można jeszcze ulepszyć? Tak, ale zaczynamy już osiągać kres jego możliwości, ponieważ ulepszenia polegają jedynie na redukcji **operacji pustych**.

Wykorzystamy informację o miejscu wystąpienia **zamiany elementów** (czyli o miejscu wykonania operacji sortującej).

Jeśli w **obiegu sortującym** wystąpi pierwsza zamiana na pozycji i-tej, to w kolejnym obiegu będziemy rozpoczynali sortowanie od pozycji o jeden mniejszej (chyba że pozycja *i*-ta była pierwszą pozycją w zbiorze). Dlaczego? Odpowiedź jest prosta. Zamiana spowodowała, iż młodszy element znalazł się na pozycji i-tej.

Ponieważ w obiegu sortującym młodszy element zawsze przesuwa się o 1 pozycję w kierunku początku zbioru, to nie ma sensu sprawdzanie pozycji od 1 do i-2, ponieważ w poprzednim obiegu zostały one już sprawdzone, nie wystąpiła na nich zamiana elementów, zatem elementy na pozycjach od 1 do i-2 są chwilowo w dobrej kolejności względem siebie. Nie mamy tylko pewności co do pozycji i-1-szej oraz i-tej, ponieważ ostatnia zamiana elementów umieściła na i-tej pozycji młodszy element, który być może należy wymienić z elementem na pozycji wcześniejszej, czyli i-1.

W ten sposób określimy początkową pozycję, od której rozpoczniemy sortowanie elementów w następnym obiegu sortującym.

Ostatnia zamiana elementów wyznaczy pozycję końcową dla następnego obiegu. Wiemy, iż w każdym obiegu sortującym najstarszy element jest zawsze umieszczany na swojej docelowej pozycji. Jeśli ostatnia zamiana elementów wystąpiła na pozycji i-tej, to w następnym obiegu porównywanie elementów zakończymy na pozycji o 1 mniejszej - w ten sposób nie będziemy sprawdzać już najstarszego elementu z poprzedniego obiegu.

Sortowanie prowadzimy dotąd, aż w obiegu sortującym nie wystąpi ani jedna zamiana elementów.

Teoretycznie powinno to zoptymalizować algorytm, ponieważ są sortowane tylko niezbędne fragmenty zbioru - pomijamy obszary posortowane, które tworzą się na końcu i na początku zbioru.

Oczywiście zysk nie będzie oszałamiający w przypadku zbioru nieuporządkowanego lub posortowanego odwrotnie (może się zdarzyć, iż ewentualne korzyści czasowe będą mniejsze od czasu wykonywania dodatkowych operacji). Jednakże dla zbiorów w dużym stopniu uporządkowanych możemy uzyskać całkiem rozsądny algorytm sortujący prawie w czasie liniowym O(n).

Przykład:

Według opisanej powyżej metody posortujmy zbiór { 0 1 2 3 **5 4** 7 9 }. W zbiorze tym są tylko dwa elementy nieuporządkowane - 5 i 4.

Obieg nr 1

[**0 1** 2 3 5 4 7 9] Pierwszy obieg, rozpoczynamy od pierwszej pozycji.
[**0 1 2** 3 5 4 7 9] Brak zamiany.
[**0 1 2 3** 5 4 7 9] Brak zamiany.
[**0 1 2 3 5** 4 7 9] Brak zamiany.
[**0 1 2** 3 **5 4** 7 9] Zamiana
[**0 1 2** 3 4 **5 7** 9] Brak zamiany.
[**0 1 2** 3 4 **5 7 9**] Brak zamiany.
[**0 1 2 3 4 5 7 9**] Koniec pierwszego obiegu.

Zbiór jest już uporządkowany, ale ponieważ była zamiana elementów, algorytm dla pewności musi wykonać jeszcze jeden obieg sortujący.

Obieg nr 2

[**0 1 2 3 4 5 7 9**]

Sortowanie rozpoczynamy od pozycji o 1 mniejszej od tej, na której wystąpiła w poprzednim obiegu wymiana elementów. Elementy są w dobrej kolejności. Dalszych sprawdzeń nie wykonujemy - kończymy na pozycji o 1 mniejszej, niż pozycja ostatniej zamiany w poprzednim obiegu.

[0 1 2 3 4 5 7 9]

Koniec, zbiór jest posortowany

Chociaż podany przykład jest troszeczkę tendencyjny, to jednak pokazuje wyraźnie, iż zoptymalizowany algorytm sortowania bąbelkowego może bardzo szybko posortować zbiory prawie uporządkowane.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze, n Є N

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i, *j* - zmienne sterujące pętli, i, j Є N

tymcz - zmienna do przechowywania tymczasowego zmienianego elementu

p - znacznik zamiany miejsc elementów w zbiorze. p Є N

Zmienne pomocnicze

i - zmienna sterująca pętli, $i \in N$

p_{min} - dolna granica pozycji sortowanych elementów, p_{min} Є N

p_{max} - górna granica pozycji sortowanych elementów, p_{max} Є N

p - numer pozycji zamiany elementów, p Є N

Lista kroków

K01: $p_{\min} \leftarrow 1; p_{\max} \leftarrow n-1$

K02: $p \leftarrow 0$

K03: Dla $i = p_{\min},\ldots, p_{\max}$, wykonuj K04...K07

K04: Jeśli $d[i] \leq d[i+1]$, to następny obieg pętli K03

K05: $d[i] \leftrightarrow d[i+1]$

K06: Jeśli $p = 0$, to $p_{\min} \leftarrow i$

K07: $p \leftarrow i$

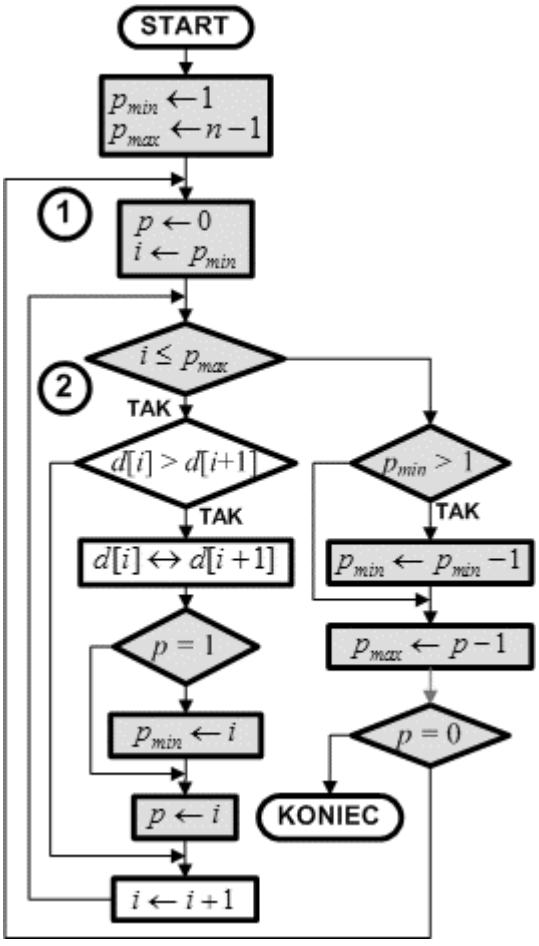
K08: Jeśli $p_{\min} > 1$, to $p_{\min} \leftarrow p_{\min} - 1$

K09: $p_{\max} \leftarrow p - 1$

K10: Jeśli $p > 0$, to idź do K02

K11: Koniec

Schemat blokowy



Tym razem wprowadzonych zmian do algorytmu sortowania bąbelkowego jest dużo, zatem opiszemy cały algorytm od początku.

Zmienna p_{min} przechowuje numer pozycji, od której rozpoczyna się sortowanie zbioru. W pierwszym obiegu sortującym rozpoczynamy od pozycji nr 1. Zmienna p_{max} przechowuje numer ostatniej pozycji do sortowania. Pierwszy obieg sortujący kończymy na pozycji n-1, czyli na przedostatniej.

Pętla numer 1 wykonywana jest dotąd, aż w wewnętrznej pętli nr 2 nie wystąpi żadna zamiana elementów. Zmienna p pełni w tej wersji algorytmu nieco inną rolę niż poprzednio. Mianowicie będzie przechowywała numer pozycji, na której algorytm ostatnio dokonał wymiany elementów. Na początku wpisujemy do p wartość 0, która nie oznacza żadnej pozycji w zbiorze. Zatem jeśli ta wartość zostanie zachowana, uzyskamy pewność, iż zbiór jest posortowany, ponieważ nie dokonano wymiany elementów.

Wewnętrzną pętlę sortującą rozpoczynamy od pozycji p_{min}. W pętli sprawdzamy kolejność elementu i-tego z elementem następnym. Jeśli kolejność jest zła, wymieniamy miejscami te dwa elementy. Po wymianie sprawdzamy, czy jest to pierwsza wymiana - zmienna p ma wtedy wartość 0. Jeśli tak, to numer pozycji, na której dokonano wymiany umieszczamy w p_{min}. Numer ten zapamiętujemy również w zmiennej p. Zwróć uwagę, iż dzięki takiemu podejściu p zawsze będzie przechowywało numer pozycji ostatniej wymiany - jest to zasada zwana "ostatni zwycięża".

Po sprawdzeniu elementów przechodzimy do następnej pozycji zwiększając i o 1 i kontynuujemy pętlę, aż do przekroczenia pozycji p_{max}. Wtedy pętla wewnętrzna zakończy się.

Jeśli w pętli nr 2 była dokonana zamiana elementów, to p_{min} zawiera numer pozycji pierwszej zamiany. Jeśli nie jest to pierwsza pozycja w zbiorze, p_{min} zmniejszamy o 1, aby pętla sortująca rozpoczynała od pozycji poprzedniej w stosunku do pozycji pierwszej zamiany elementów.

Pozycję ostatnią zawsze ustalamy o 1 mniejszą od numeru pozycji końcowej zamiany elementów.

Na koniec sprawdzamy, czy faktycznie doszło do zamiany elementów. Jeśli tak, to p jest większe od 0, gdyż zawiera numer pozycji w zbiorze, na której algorytm wymienił miejscami elementy. W takim przypadku pętlę nr 1 rozpoczynamy od początku. W przeciwnym razie kończymy, zbiór jest uporządkowany.

Dwukierunkowe sortowanie bąbelkowe
Bidirectional Bubble Sort

Dwukierunkowe sortowanie bąbelkowe oparte jest na spostrzeżeniu, iż każdy obieg wewnętrznej **pętli sortującej** umieszcza na właściwym miejscu **element najstarszy**, a **elementy młodsze** przesuwa o 1 pozycję w kierunku początku zbioru.

Jeśli pętla ta zostanie wykonana w kierunku odwrotnym, to wtedy najmłodszy element znajdzie się na swoim właściwym miejscu, a elementy starszy przesuną się o jedną pozycję w kierunku końca zbioru.

Połączmy te dwie pętle sortując wewnętrznie naprzemiennie w kierunku normalnym i odwrotnym, a otrzymamy algorytm dwukierunkowego sortowania bąbelkowego.

Wykonanie pętli sortującej w normalnym kierunku ustali maksymalną pozycję w zbiorze, od której powinna rozpocząć sortowanie pętla odwrotna.

Ta z kolei ustali minimalną pozycję w zbiorze, od której powinna rozpocząć sortowanie pętla normalna w następnym obiegu pętli zewnętrznej. Sortowanie możemy zakończyć, jeśli nie wystąpiła potrzeba zamiany elementów w żadnej z tych dwóch pętli.

Specyfikacja problemu

Dane wejściowe

n - liczba elementów w sortowanym zbiorze,

$$n \in N$$

d[ ] - zbiór *n*-elementowy, który będzie sortowany. Elementy zbioru mają indeksy od 1 do *n*.

Dane wyjściowe

d[ ] - posortowany zbiór *n*-elementowy. Elementy zbioru mają indeksy od 1 do *n*.

Zmienne pomocnicze

i - zmienna sterująca pętli,

$$i \in N$$

p_{min} - dolna granica pozycji sortowanych elementów,

$$p_{min} \in N$$

p_{max} - górna granica pozycji sortowanych elementów,

$$p_{max} \in N$$

p - numer pozycji zamiany elementów,

$$p \in N$$

Lista kroków

Operacja Sortuj

K01: Jeśli $d[i] \leq d[i+1]$, to koniec
K02: $d[i] \leftrightarrow d[i+1]$
K03: $p \leftarrow i$
K04: Koniec

Algorytm główny

K01: $p_{\min} \leftarrow 1; p_{\max} \leftarrow n-1$

K02: $p \leftarrow 0$

K03: Dla $i = p_{\min}, p_{\min}+1, \ldots, p_{\max}$, Sortuj

K04: Jeśli $p = 0$, to koniec

K05: $p_{\max} \leftarrow p-1; p \leftarrow 0$

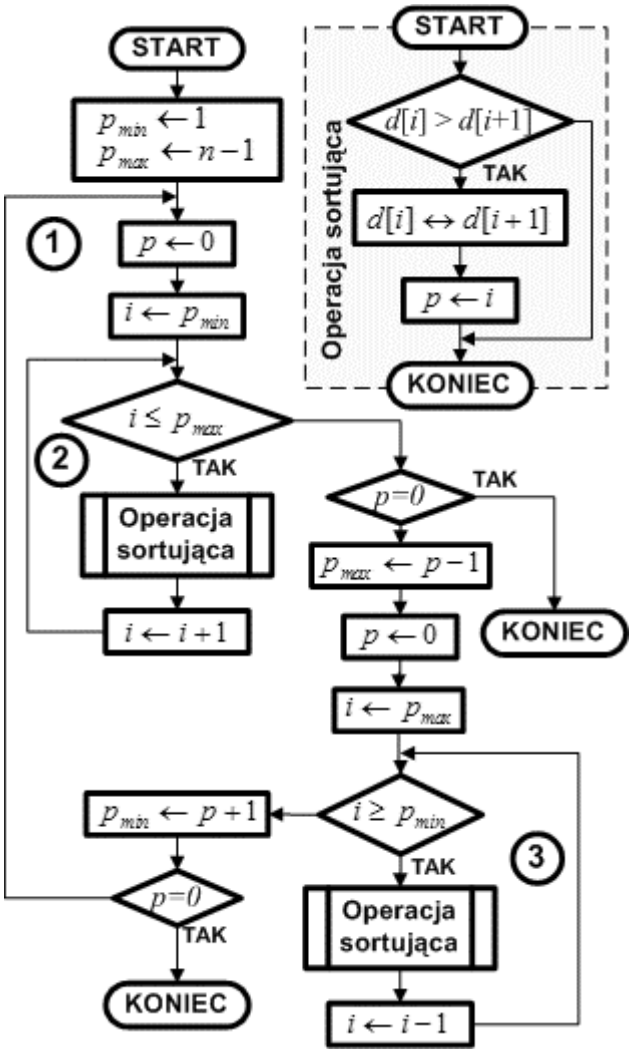
K06: Dla $i = p_{\max}, p_{\max}-1, \ldots, p_{\min}$, Sortuj

K07: $p_{\min} \leftarrow p+1$

K08: Jeśli $p > 0$, to idź do K02

K09: Koniec

Schemat blokowy



W algorytmie wydzieliliśmy powtarzający się fragment operacji i nazwaliśmy go operacją sortującą. Porównuje ona dwa kolejne elementy zbioru i zamienia je miejscami, jeśli są w złej kolejności. Po zamianie do zmiennej p trafia indeks pierwszego z elementów pary. Podany warunek sprawdza

uporządkowanie rosnące. Jeśli chcemy posortować zbiór malejąco, relację większości należy zastąpić relacją mniejszości.

W algorytmie występują trzy pętle. Pętla nr 1 jest pętlą warunkową i obejmuje dwie pętle wewnętrzne nr 2 i nr 3. Pętla ta wykonywana jest dotąd, aż w sortowanym zbiorze nie wystąpi w trakcie sortowania ani jedna zamiana miejsc elementów.

Pętla nr 2 jest pętlą sortującą w górę. Pętla nr 3 sortuje w dół.

Na początku algorytmu ustalamy dwie granice sortowania:

- dolną w p_{min}
- górną w p_{max}.

Granice te określają indeksy elementów zbioru, które będą przeglądały pętle sortujące nr 2 i nr 3. Początkowo granice są tak ustawione, iż obejmują cały sortowany zbiór.

Na początku pętli nr 1 zerujemy zmienną p. Zmienna ta będzie zapamiętywać pozycję ostatniej zamiany elementów. Jeśli po przejściu pętli sortującej nr 2 lub nr 3 zmienna p wciąż zawiera wartość 0, to nie wystąpiła żadna zamiana elementów. Zbiór jest wtedy uporządkowany i kończymy algorytm.

Pierwszą pętlę sortującą wykonujemy kolejno dla indeksów od p_{min} do p_{max}. Po zakończeniu pętli sprawdzamy, czy p jest równe 0. Jeśli tak, kończymy algorytm. W przeciwnym razie p zawiera pozycję ostatniej zamiany elementów. Ponieważ pętla nr 2 ustala pozycję najstarszego elementu, to elementy o indeksach od p do n są już właściwie uporządkowane. Dlatego dla kolejnego obiegu pętli przyjmujemy p_{max} o 1 mniejsze niż p.

Przed rozpoczęciem drugiej pętli zerujemy p. Jest to dosyć ważne. Załóżmy, iż zbiór został już uporządkowany w poprzedniej pętli nr 2, lecz wystąpiła tam zamiana elementów. Jeśli nie wyzerujemy p, to następna pętla sortująca nie zmieni zawartości tej zmiennej i p zachowa wartość większą od 0. Zatem pętla główna nr 1 nie zakończy się i algorytm wykona niepotrzebnie jeszcze jeden pusty obieg. Niby nic, a jednak...

Pętla nr 3 sortuje w kierunku odwrotnym od p_{max} do p_{min}. Po jej zakończeniu p zawiera indeks ostatniej zamiany elementów. Podobnie jak poprzednio zbiór jest uporządkowany od elementu nr 1 do p. Zatem dla kolejnych obiegów przyjmujemy p_{min} o 1 większe od p. Jeśli p jest równe 0, kończymy algorytm. W przeciwnym razie kontynuujemy pętlę nr 1.