

Podstawowe metody sortowania

Algorytmy sortowania

Sortowanie danych jest tematem wielu prac z informatyki, a zwłaszcza z algorytmiki. Algorytmy sortowania danych znajdują swoje zastosowanie w wielu systemach obliczeniowych, bazach danych i symulacjach komputerowych. Budując algorytm bardzo często napotykamy problem braku uporządkowania w danych. Rozdział ten przedstawia propozycje praktycznej implementacji kilku metod sortowania danych. W literaturze [1,2, 21, 28, 39, 45-47, 61, 66-67, 71, 73, 84, 90, 100, 112, 115] przedstawiono wiele podobnych metod, które pozwalają porządkować posiadane informacje. Dobór algorytmu porządkowania elementów zbioru danych w dużej mierze zależy od sposobu przechowywania danych. Autorzy tej książki pokazali, w jaki sposób można sortować dane przechowywane w kopcach czy stosach. W literaturze [11, 21, 44-45, 66, 107] przedstawiono podobne, efektywne metody sortowania danych ułożonych na kopcach poprzez zastosowanie różnych modyfikacji algorytmów operujących na grafach i drzewach. Natomiast w literaturze [1-2, 37, 39, 66, 93, 102, 112] przedstawiono modyfikacje algorytmów operujących na stosach oraz innych strukturach danych.

Często mamy do czynienia również z innym schematem przechowywania danych. W rozdziale tym zaproponowano metody przeszukiwania danych leksykograficznych, co znajduje swoje zastosowanie w bazach danych. W literaturze [39, 84, 114, 115] znajdziemy podobne algorytmy, które pozwalają operować na bazach danych i efektywnie przeszukiwać ich rekordy. Problem sortowania danych jest często opisywany w literaturze i można znaleźć różne propozycje algorytmów sortujących. Autorzy niniejszej książki postanowili przedstawić swoje propozycje implementacji niektórych metod sortowania danych. Rozwiązania przedstawione w tym rozdziale mogą służyć programistom jako odrębne algorytmy sortujące lub dzięki zaproponowanej budowie można wykorzystać tylko ich fragmenty jako składowe nowych, większych metod sortujących zbiory danych. Wiedza pokazana w podrozdziałach pozwoli Czytelnikowi wybrać najwłaściwszy dla danego problemu algorytm sortowania danych, dzięki czemu tworzone oprogramowanie może działać w sposób bardziej efektywny.

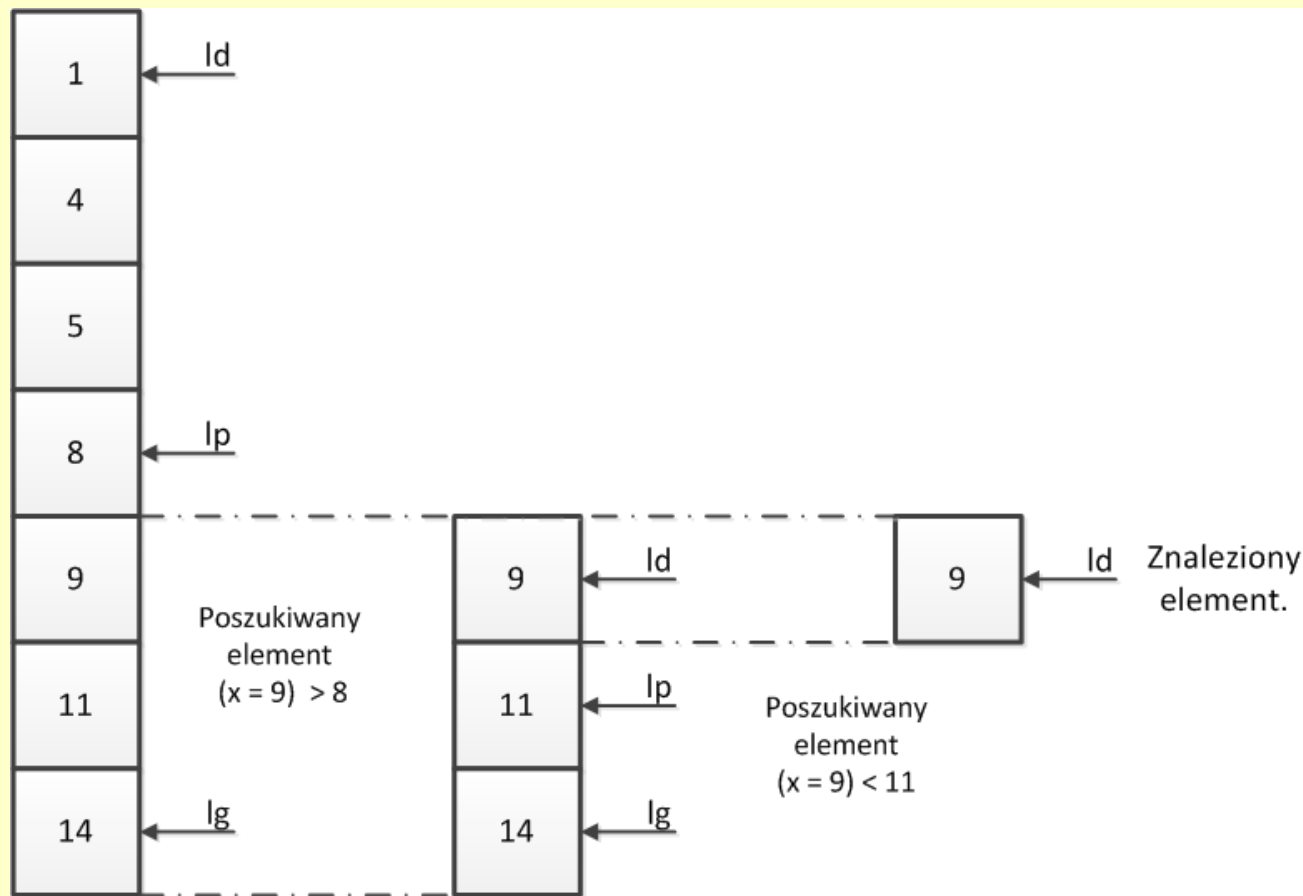
Wartością przedstawionych rozwiązań jest również możliwość ich zastosowania w praktycznej nauce budowania algorytmów. Czytelnik programując często będzie rozwiązywał problem porządkowania elementów w zbiorze. W tym celu ważnym jest poznanie algorytmów służących do porównywania i szeregowania wartości. Chcemy zaprezentować metody sortowania w praktycznych przykładach jako modelowe rozwiązania problemu sortowania danych różnymi sposobami. Pokazanie różnych algorytmów sortowania danych na konkretnym przykładzie pomoże zrozumieć zasadę budowania złożonych algorytmów, która będzie przydatna w następnych rozdziałach książki. W dalszych rozdziałach zostaną przedstawione implementacje skomplikowanych algorytmów rozwiązujących układy równań liniowych dowolnych rzędów, zatem wiedza zgromadzona na podstawie tego rozdziału pomoże Czytelnikowi zaczynającemu dopiero przygodę z algorytmiką na lepsze zrozumienia omawianych treści. W przedstawionych przykładach sortowania pokazane zostały sposoby implementacji kilku funkcji w programie, ich wzajemnej współpracy i przekazywania wartości.

W przykładach praktycznych pokazano, w jaki sposób można sortować dane znajdujące się w pliku zewnętrznym, a następnie chronologiczne wartości zamieścić w innym pliku wynikowym. Do przykładów zostały dołączone rysunki, które krok po kroku pokazują zasady przyjęte podczas realizacji poszczególnych omawianych algorytmów. Zamieszczone komentarze objaśniają poszczególne operacje, jakie wykonujemy podczas sortowania danych poszczególnymi metodami i pomogą początkującym programistom zrozumieć omawiane procedury.

Autorzy mają nadzieję, że przedstawione w tym rozdziale rozwiązania praktyczne pomogą w realizacji zadań programistom i inżynierom. Natomiast dla początkujących programistów będą stanowić dobry materiał dydaktyczny, który pozwoli im opanować sztukę programowania i budowy algorytmów oraz nabrać potrzebnego doświadczenia do samodzielnej pracy.

Wyszukiwanie binarne

Zaczniemy od sytuacji ciągu danych, które sąsiadują ze sobą. Sortowanie binarne odbywa się dla odpowiednich uporządkowanych poprzez indeksowanie części ciągu danych. Algorytm poszukiwania binarnego zawęża przedział poszukiwania w każdym kroku o połowę korzystając z informacji o uporządkowaniu ciągu. Niech ld oznacza dolny wskaźnik ciągu, lg oznacza górny wskaźnik ciągu oraz $lp = (ld + lg) / 2$. Stąd poszukiwanie liczby 9 w ciągu 1, 4, 5, 8, 9, 11, 14 możemy zilustrować następująco.



Powyższa ilustracja pokazuje, w jaki sposób można przeszukać binarnie ciągi liczbowe. Natomiast dla ciągu literowego przeszukiwanie to będzie wyglądało identycznie uwzględniając funkcję porównywania ciągów znakowych. Zobaczmy teraz jak może wyglądać implementacja takiego algorytmu w postaci kodu.

Przykład 44

Napisać program realizujący przeszukiwanie binarne ciągu liczbowego wczytanego z pliku.

```
#include<stdio.h>
#include<conio.h>
struct dane
{
    int id;
    char nazwisko[24], praca[24];
};
int porownaj(char *s,char *t)
{
    while(*s==*t)
    {
        if(*s=='\0') return 0;
        s++;
        t++;}
    return *s-*t;
}
```

// Deklaracja bibliotek.

// Deklaracja struktury.

/* Funkcja *porownaj()* zwraca: <0 jeśli s<t, 0 jeśli s==t, >0 jeśli s>t. */

/* Porównuj do momentu, gdy znaki w obu tekstach są równe. */

/*Teksty są różne, bo doszliśmy do ostatnich znaków obu tekstów. */

/* Zwracamy różnicę na pierwszej pozycji, na której znaki są różne. */


```
int main()
{
    struct dane pracownik[1024];
    int i, n=0, ld, lp, lg;
    char x[1024];
    FILE *f;
    if((f = fopen("dane.txt", "r")) == NULL)
    {
        printf("Nie moge otworzyc pliku dane.txt\n");
        _getch();
        return 0;
    }
    while(fscanf(f,"%d %s %s",
                &(pracownik[n].id),
                pracownik[n].nazwisko,
                pracownik[n].praca)!=EOF)
        n++;
    for(i = 0; i < n; i++)
        printf("%d %s %s\n",pracownik[i].id,
                pracownik[i].nazwisko,pracownik[i].praca);
    printf("Podaj szukane nazwisko:");
    scanf("%s",x);
```

// Funkcja główna programu.

// Otwarcie pliku.

// Wczytanie całej tabeli.

/* Wydruk przeczytanych rekordów.*/

```

ld=0;
lg=n-1;
while((lg-ld) > 0)
{
    lp = (ld + lg)/2;
    if(porownaj(pracownik[lp].nazwisko,x) == 0)
    {
        printf("%d %s %s\n",pracownik[lp].id,
        pracownik[lp].nazwisko,pracownik[lp].praca);
        _getch();
        return 0;
    }
    if(porownaj(x,pracownik[lp].nazwisko)<0) lg=lp-1;
    else ld=lp+1;
}
if(porownaj(pracownik[ld].nazwisko,x)==0) {
    printf("%d %s %s\n", pracownik[ld].id,
    pracownik[ld].nazwisko, pracownik[ld].praca);
    _getch();
    return 0;
}
printf("Nie znalazlem pracownika!\n");
_getch();
}

```

```

/* Algorytm wyszukiwania
binarnego. */

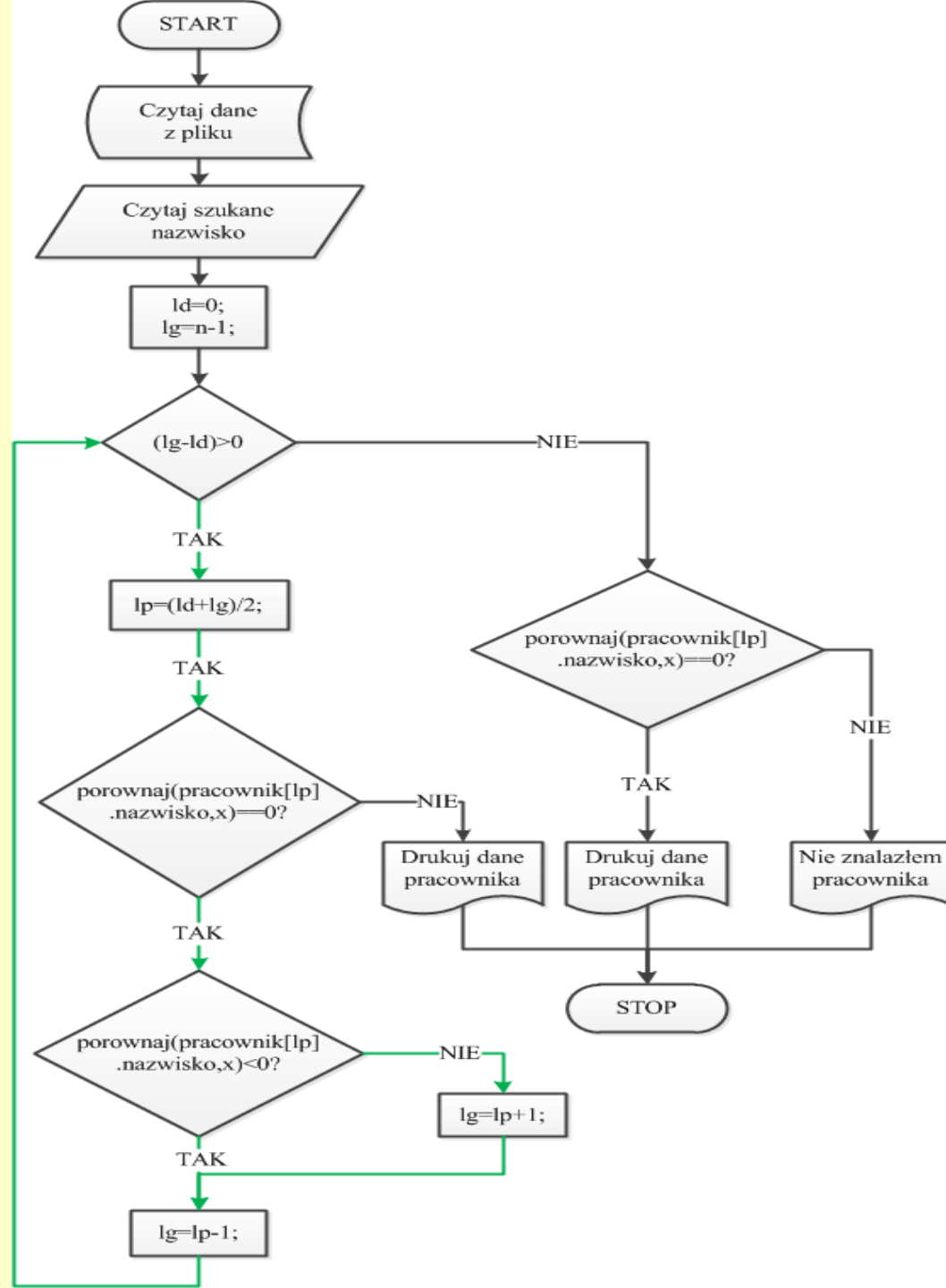
```

```

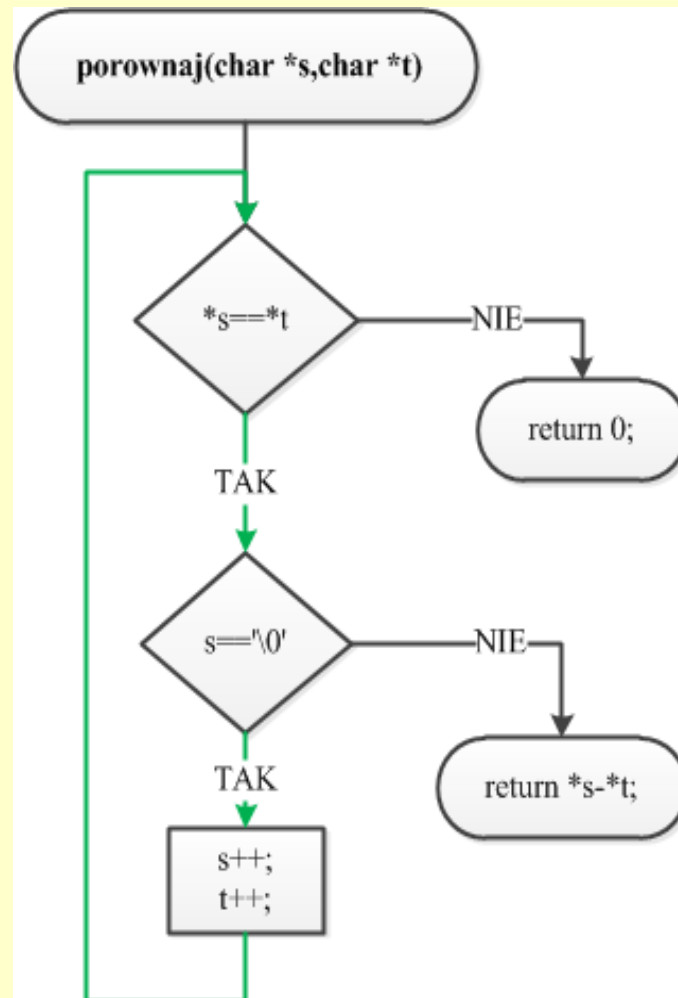
/*      Sprawdzenie      czy
doszliśmy do końca. */

```

W literaturze [1-2, 39, 45-47, 51, 61, 71, 73, 82, 83, 90, 100, 112, 114, 115] można znaleźć ciekawe modyfikacje opisanego algorytmu, które mogą ułatwić sortowanie w specyficznych przypadkach. Schematy blokowe wykonywanych operacji zostały pokazane na kolejnych ilustracjach.

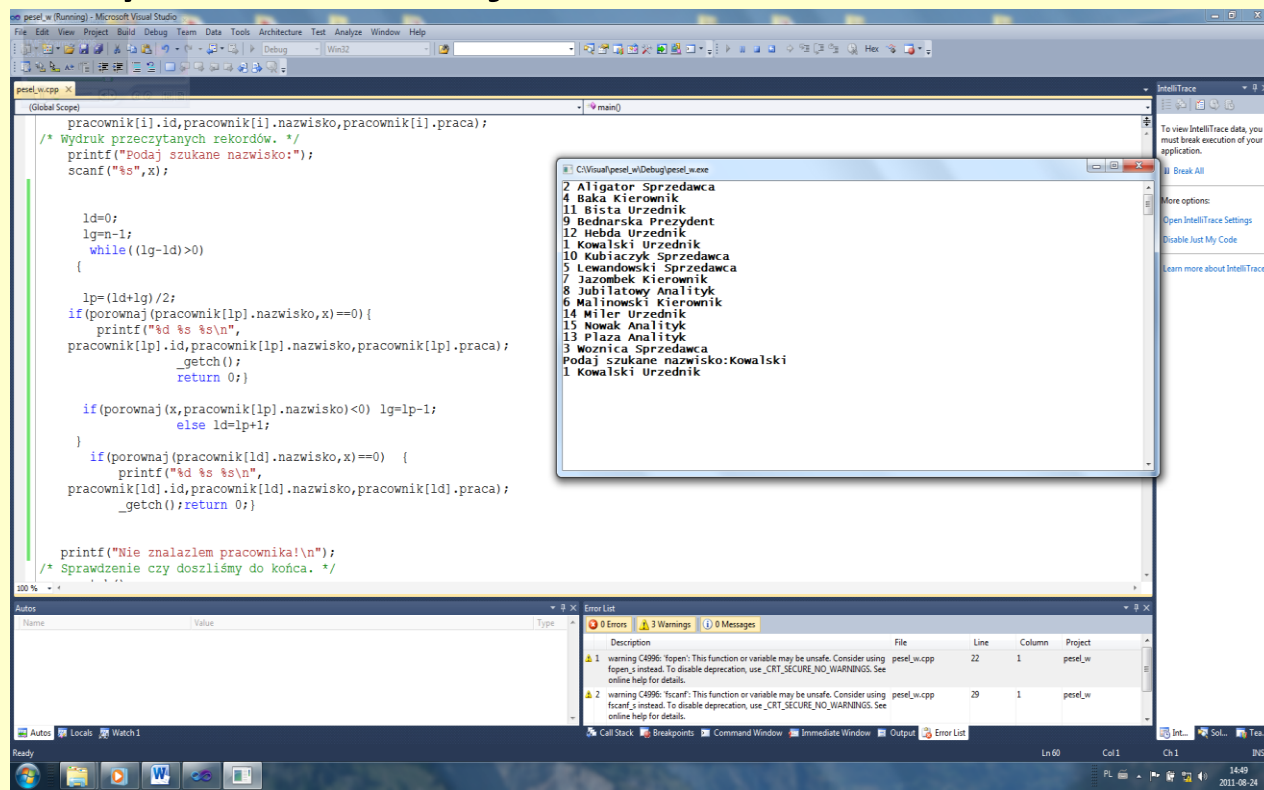


Schemat blokowy wyszukiwania binarnego.



Powyższy schemat blokowy pokazuje realizację operacji porównania wartości w ciągach. Wykorzystane zostały w niej operacje na wskaźnikach, co znacznie ułatwia implementację metody. Operacja porównywania elementów ciągu jest powtarzana odpowiednią ilość razy w całej procedurze sortującej. Czytelnik zechce prześledzić schemat całego algorytmu sortowania, jeszcze raz aby lepiej zrozumieć, w jaki sposób przedstawiony program dokonuje sortowania posiadanych informacji.

Działanie programu skompilowanego pod MS Visual Studio. Czytelnik może zaobserwować na poniższej ilustracji.



```
pracownik[i].id, pracownik[i].nazwisko, pracownik[i].praca);  
/* Wydruk przeczytanych rekordów. */  
printf("Podaj szukane nazwisko:");  
scanf("%s", x);  
  
    id=0;  
    lg=n-1;  
    while((lg-id)>0)  
    {  
        lp=(id+lg)/2;  
        if(porownaj(pracownik[lp].nazwisko, x)==0){  
            printf("%d %s %s\n",  
                pracownik[lp].id, pracownik[lp].nazwisko, pracownik[lp].praca);  
            _getch();  
            return 0;  
        }  
        if(porownaj(x, pracownik[lp].nazwisko)<0) lg=lp-1;  
        else id=lp+1;  
    }  
    if(porownaj(pracownik[id].nazwisko, x)==0) {  
        printf("%d %s %s\n",  
            pracownik[id].id, pracownik[id].nazwisko, pracownik[id].praca);  
        _getch(); return 0;  
    }  
  
    printf("Nie znalazlem pracownika!\n");  
/* Sprawdzenie czy doszliśmy do końca. */
```

2 Aligator Sprzedawca
4 Baka Kierownik
11 Bista Urzednik
9 Bednarska Prezydent
12 Hebda Urzednik
1 Kowalski Urzednik
10 Kubiacyk Sprzedawca
5 Lewandowski Sprzedawca
7 Jazonek Kierownik
8 Jubilatowy Analityk
6 Malinowski Kierownik
14 Miller Urzednik
15 Nowak Analityk
13 Plaza Analityk
3 Woznica Sprzedawca
Podaj szukane nazwisko: Kowalski
1 Kowalski Urzednik

Name	Value	Type
0 Errors	3 Warnings	0 Messages

Description	File	Line	Column	Project
warning C4996: 'fopen': This function or variable may be unsafe. Consider using fopen_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.	pesel_w.cpp	22	1	pesel_w
warning C4996: 'scanf': This function or variable may be unsafe. Consider using scanf_s instead. To disable deprecation, use _CRT_SECURE_NO_WARNINGS. See online help for details.	pesel_w.cpp	29	1	pesel_w

Złożoność obliczeniowa algorytmu wyszukiwania binarnego

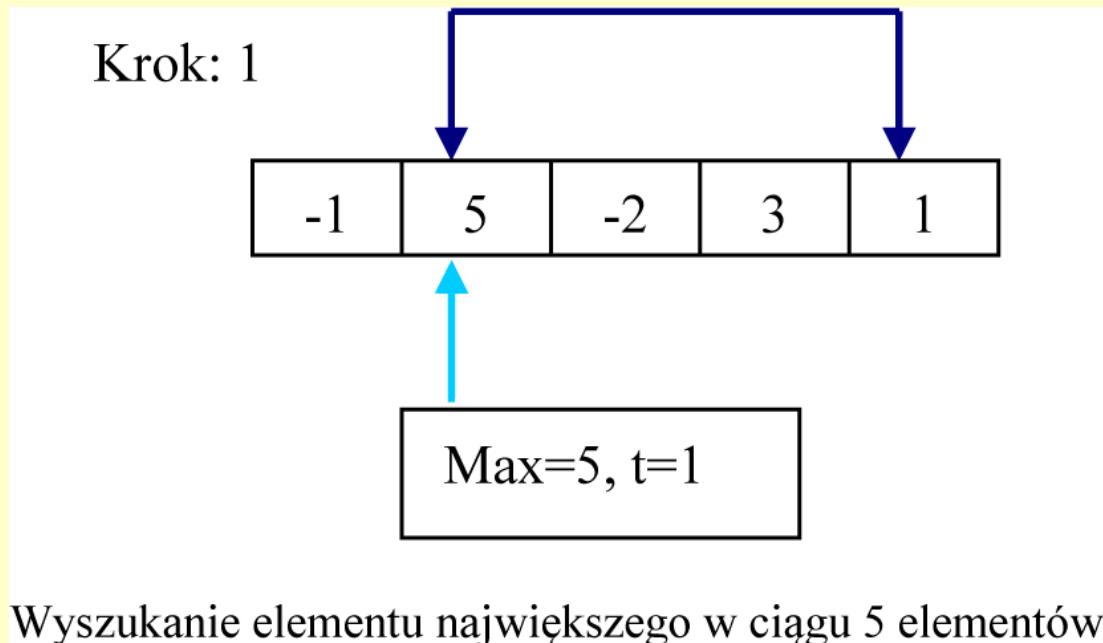
Algorytm wyszukiwania binarnego ma złożoność czasową $\vartheta(\log_2 n)$ i można go stosować dla każdego ciągu uporządkowanego. Oznacza to, że wyszukiwanie umożliwia znalezienie informacji w ciągu uporządkowanym o czasie $\vartheta(\log_2 n)$.

Sortowanie przez wstawianie na koniec ciągu

Pokażemy teraz metodę sortowania przez wstawianie. Ten rodzaj algorytmu sortowania danych porównuje poszczególne znaki sortowanego ciągu i wstawia największy znaleziony znak na koniec. Następnie sortowaniu poddany zostaje nowy ciąg. Ponieważ największy element znajduje się już na końcu tego ciągu procedura porównania odbywa się dla $n-1$ elementów tego ciągu. Taka procedura jest powtarzana do momentu uzyskania ciągu jednoelementowego rozumianego jako najmniejszy z porównywanych elementów.

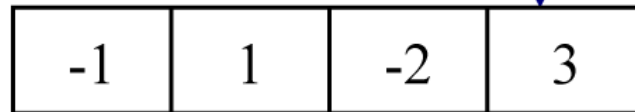
Przykład 45

Dla łatwiejszego zrozumienia procedury przyjęto następujące oznaczenia: Max – maksimum w przeszukiwanym ciągu, t – indeks elementu największego. Stosując omawiany algorytm wyszukiwania liczby największej możemy znaleźć element największy o indeksie t równym 1. Uwaga: Czytelnik musi pamiętać, że w językach programowania indeksowanie elementów następuje od 0. Po przestawieniu otrzymujemy ciąg: -1, 1, -2, 3, 5. W stworzonym ciągu element największy mamy już na pozycji ostatniej.



Zatem następnie w kroku 2 stosujemy algorytm wyszukiwania największej liczby jedynie dla ciągu czterech pierwszych elementów. W tym przypadku przestawiamy element ostatni sam ze sobą, ponieważ maksimum w tym przypadku znajduje się na ostatnim miejscu ciągu.

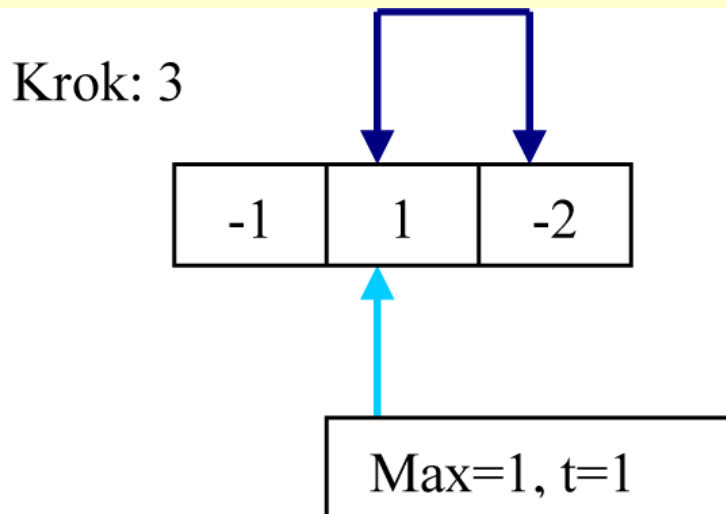
Krok: 2



Max=3, t=3

Wyszukanie elementu największego w ciągu 4 elementów.

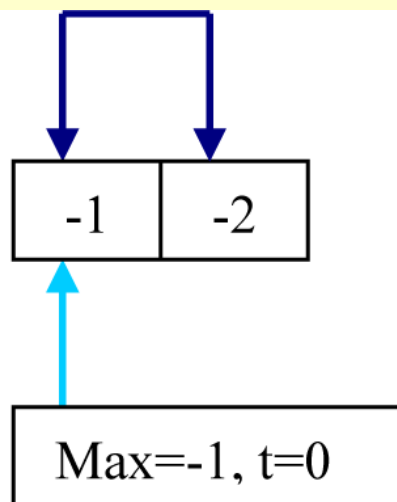
W kroku następnym mamy już tylko trzy elementy w przeszukiwanym ciągu: -1, 1, -2. Czytelnik zauważył, że na kolejnej ilustracji największy element znajduje się na pozycji iterowanej indeksem $t=1$. Zostaje on zamieniony z elementem znajdującym się na ostatniej pozycji w ciągu. W wyniku wykonania tej operacji w ostatnim czwartym kroku rozważaniu zostanie poddany ciąg zawierający już tylko dwie liczby: -1, -2. Zatem znalezienie elementu największego sprowadza się już tylko do porównania wartości dwóch kolejnych elementów ciągu, co pokazano na ostatniej ilustracji iteracji metody sortowania przez wstawianie.



Wyszukanie elementu największego w ciągu 3 elementów.

Jednocześnie wykonanie ostatniego kroku 4 w algorytmie spowoduje posortowanie całego omawianego ciągu według wartości rosnących. W wyniku zastosowania algorytmu przez wstawianie otrzymujemy posortowany ciąg liczb: -2, -1, 1, 3, 5.

Krok: 4



Wyszukanie elementu największego w ciągu 2 elementów.

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int n, i, j, a[1024], t, max;
```

```
    printf("Podaj n:");
```

```
    scanf("%d",&n);
```

```
    for(i=0;i<n;i++)
```

```
{
```

```
        printf("Podaj a[%d]=", i);
```

```
        scanf("%d",&a[i]);
```

```
}
```

```
// Deklaracja bibliotek
```

```
/* Deklaracja zmiennych i  
tablicy znaków*/
```

```
/*Wczytanie          wymiaru  
zadania.*/
```

```
/* Wczytanie elementów ciągu  
liczb.*/
```

```
for(i=0; i<n-1; i++)
{
    t = 0;
    max = a[0];

    for(j = 1; j < n-i; j++)
        if(a[j]>max)
        {
            max=a[j];
            t=j;
        }
    a[t] = a[(n-1)-i];
    a[(n-1)-i] = max;
}

for(i = 0; i < n; i++)
    printf("a[%d]=%d\n", i, a[i]);

return 0;
_getch();
}
```

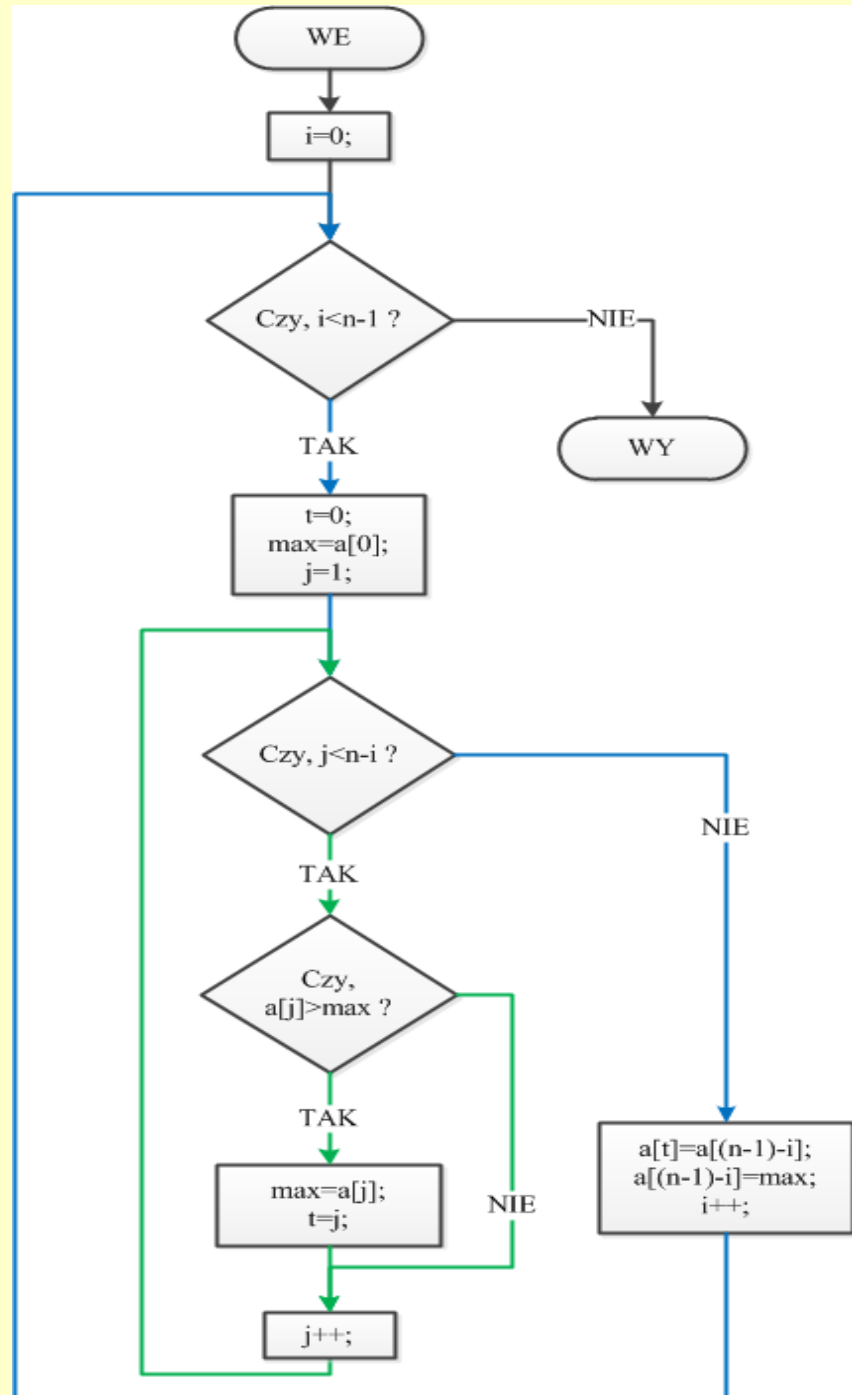
```
/* Iteracja zliczająca ilość elementów do
opuszczenia w ciągu liczb
porządkowanym. */
/* Ustawienie wskaźnika elementu
największego oraz zapamiętanie jego
wartości. */

/* Pętla przeszukująca ciąg w celu
znalezienia elementu największego i
zapamiętująca jego wskaźnik. */

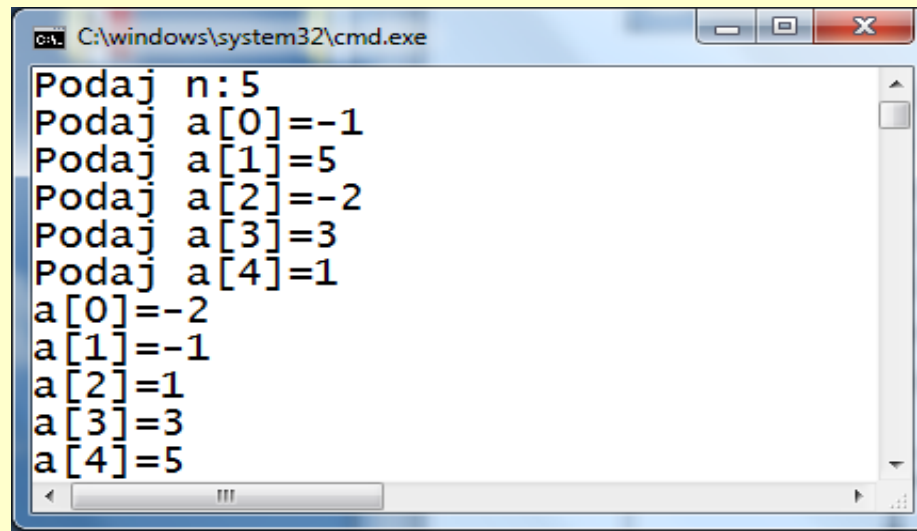
/*Przestawienie elementu największego na
koniec ciągu.*/

/* Pętla wypisująca na ekran
uporządkowany ciąg.*/
```

Schemat blokowy sortowania przez wstawianie.



Czytelnik zechce porównać operacje kodu z przedstawionym na wcześniejszej ilustracji schematem blokowym operacji wykonywanych w trakcie realizacji sortowania za pomocą wstawiania. Na schemacie blokowym warto przeanalizować kolejność wykonywanych podstawień i pętli w trakcie realizacji procedury sortowania. Czytelnik zapewne zwrócił uwagę, że schemat różni się od przedstawianych dotychczas. Oznaczenie wejścia poprzez „We” oraz wyjścia poprzez „Wy” jest charakterystyczne dla procedur stanowiących jedynie część całego programu. Nie zmienia to faktu, że zamieniając te oznaczenia na znane Czytelnikowi *START* i *STOP* otrzymamy schemat blokowy odrębnego programu. Natomiast wyniki otrzymane po kompilacji kodu są przedstawione na kolejnej ilustracji.



```
C:\windows\system32\cmd.exe
Poda n: 5
Poda a[0]=-1
Poda a[1]=5
Poda a[2]=-2
Poda a[3]=3
Poda a[4]=1
a[0]=-2
a[1]=-1
a[2]=1
a[3]=3
a[4]=5
```

Zastanówmy się teraz nad złożonością przedstawionego algorytmu sortowania przez wstawianie. Jeżeli uzależnimy czas działania programu od wymiaru zadania to mówimy, że wyznaczyliśmy złożoność czasową algorytmu. Analogicznie złożoność pamięciowa jest funkcją zajmowanego miejsca w pamięci komputera od wymiaru zadania.

Złożoność pamięciowa jest funkcją liniową postaci:

$$f(n) = a \cdot n + c$$

gdzie przez następujące symbole rozumiemy:

n – wymiar zadania. W naszym przypadku wymiarem zadania jest ilość liczb, jaką mamy do posortowania, czyli n elementów omawianego ciągu.

a - stała odpowiadająca ilości bajtów potrzebnych do zapamiętania liczby całkowitej. Dla liczb całkowitych typu *long int* jest ona równa 4.

c - stała zależną od wielkości programu, która dla różnych kompilatorów pracujących pod różnymi systemami operacyjnymi jest różna. Niemniej jako stałą dla naszych rozważań możemy ją pominąć.

Czas, jaki potrzebujemy do wykonania programu zapisujemy symbolem $\vartheta(n)$. Zapis taki oznacza, że ilość potrzebnego miejsca w pamięci komputera dla zadania dwa razy większego będzie również dwa razy większa. Do wyznaczenia złożoności czasowej należy zauważyć, że do wyszukania największej z liczb w ciągu n elementowym potrzeba wykonać $n-1$ porównań. Po przestawieniu elementu największego na koniec opuszczamy go w następnej iteracji i ilość potrzebnych porównań zmniejsza się o jeden. Zapisując ilość potrzebnych porównań w kolejnych iteracjach, a więc złożoność czasową otrzymujemy wzór:

$$g(n) = (n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n - 1}{2} \cdot n$$

gdzie n oznacza ilość elementów w rozpatrywanym zadaniu, a więc jego wymiar. Ograniczamy teraz przedstawionym równaniem funkcję złożoności czasowej w następujący sposób:

$$\frac{n^2}{4} \leq \frac{n(n-1)}{2} \leq \frac{n^2}{2}, \quad n \geq 2$$

Wnioskujemy, że czas działania programu wynosi . Oznacza to, że jeżeli weźmiemy dwa razy większy ciąg liczbowy będziemy potrzebowali cztery razy więcej czasu na jego posortowanie. Zatem wstawiając do powyższych równań dane, jakie mamy dla omawianego ciągu otrzymamy następujące wartości

$$n = 5$$

$$f(5) = 4 \cdot 5 = 20$$

$$g(n) = (5 - 1) + \dots + 1 = \frac{5 - 1}{2} \cdot 5 = 10$$

Omówiony w przykładzie algorytm sortowania przez wstawianie jest jednym z podstawowych algorytmów sortowania. W schematach blokowych złożoność obliczeniową można określić poprzez ilość występowania zapętlenia w algorytmie. Na schematach blokowych w tym i kolejnych rozdziałach zapętlenia w omawianych procedurach są odpowiednio oznaczone kolorami, aby ułatwić Czytelnikowi zrozumienie złożoności obliczeniowej poszczególnych omawianych metod.

Sortowanie bąbelkowe

Kolejnym algorytmem sortowania jest sortowanie bąbelkowe. Jest to jeden z ważniejszych algorytmów ze względu na jego sposób działania. Modyfikacje i odwołania do tej metody możemy znaleźć w literaturze [1-2, 39, 45-47, 51, 71, 73, 85, 90-91]. Działanie tego algorytmu podobnie jak poprzednio opiera się na zamianie elementów w rozważanym ciągu miejscami, tak aby element największy lub najmniejszy przesunąć na koniec ciągu w każdej z wykonywanych iteracji. Różnica polega jednak na tym, że zamiana elementów dokonana zostaje w poszczególniej iteracji poprzez wszystkie sąsiednie elementy. Nie zamieniamy, tak jak poprzednio jedynie elementu ostatniego w ciągu ze znalezionym największym. Zakładamy, że właśnie w elemencie ostatnim znajduje się wartość poszukiwana wartość ekstremalna, zatem nie wymaga ona zamiany miejscem. Prześledźmy teraz w jaki sposób będzie wyglądać sortowanie bąbelkowe ciągu liczbowego.

Przykład 46

Wykonać sortowanie bąbelkowe dla zadanego ciągu liczb -1, 5, -2, 3, 1.

Algorytm sortowania bąbelkowego łatwo można rozpisać dla ciągu liczb. Porównujemy w nim ze sobą kolejne elementy stykając ekstremum w każdej rozpatrywanej parze. Znalezioną wartość ekstremalną przesuwamy w ustalonym kierunku. Następnie porównujemy elementy w kolejnej utworzonej parze szukając największego i przesuwając go na koniec. Takie porównania wykonujemy, że dojdziemy do końca ciągu i element największy znajdzie się na ostatnim miejscu. Jak widzimy na poniższych ilustracjach sortowania bąbelkowego, po porównaniu kolejnych liczb w ciągu można dokonać ewentualnej zamiany „sąsiadów”. Ewentualnemu przestawieniu podlega para kolejnych liczb. Jeśli większa z nich jest „pierwszą” w parze zostaje ona wypchnięta na koniec. Następnie w kolejnym kroku analogicznie rozważamy kolejną parę w ciągu

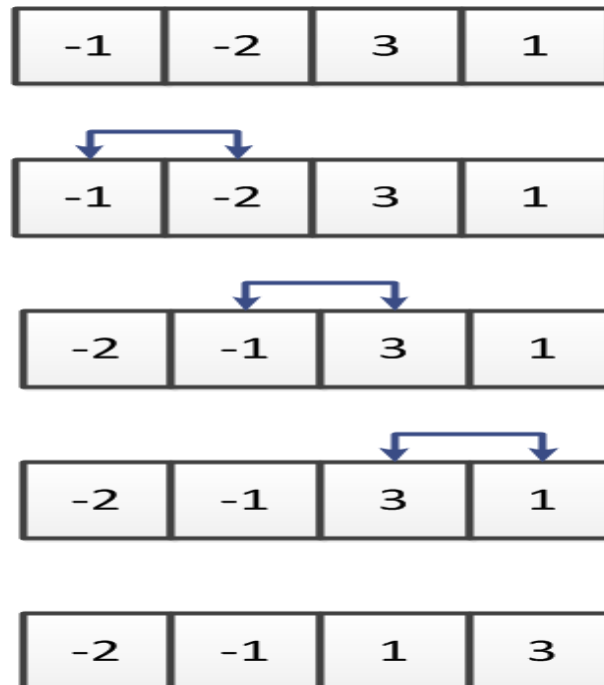
W ten sposób porównamy ze sobą wszystkie pary liczb, jakie znajdują się w rozważanym ciągu. W wyniku każdego kroku algorytmu bąbelkowego element największy zostanie umieszczony dokładnie na końcu rozważanego ciągu. Należy zauważyć, że w każdej iteracji rozważamy ciąg o jeden element mniejszy, ponieważ w wyniku poprzednich kroków wartości maksymalne są ustawione już w kolejności chronologicznej na samym końcu badanego ciągu. Zobaczmy teraz graficzną ilustrację wykonywanych operacji.

Krok 1:



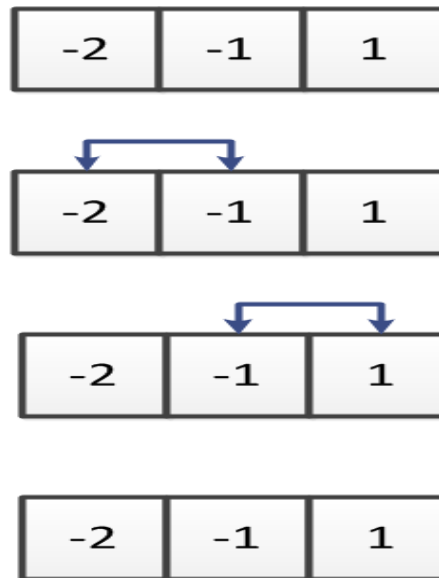
W następnym kroku bierzemy pod uwagę ciąg o jeden mniejszy i dokonujemy analogicznego porównania kolejnych par liczb. W naszym przypadku w kroku 2 zostają porównane wszystkie pary liczb w ciągu 4 elementów. Ciąg taki jest teraz jedynym słusznym do rozważenia, ponieważ $max=5$ zostało już umieszczone na końcu w poprzednim kroku. W wyniku porównania kolejnych par umiejscowimy element największy spośród badanych 4 elementów na końcu. Jest to pozycja 3 w badanym ciągu i jednocześnie 4 w całym zadaniu.

Krok 2:

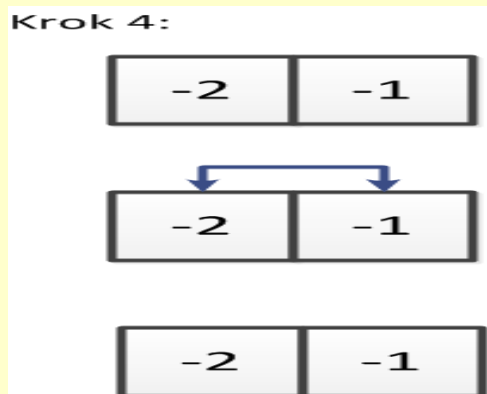


Następna iteracja/krok wypycha kolejny element na koniec ciągu. Bierzemy jednocześnie pod uwagę ciąg złożony z mniejszej liczby elementów, ponieważ w poprzedniej iteracji ustawiliśmy już element największy na jego końcu i nie wymaga on ani porównywania ani przestawiania. Operacje porównanie kolejnych par elementów zostały pokazane na ilustracji.

Krok 3:



Czytelnik zauważył zapewne, że ciąg jaki został poddany operacji porównania jest już odpowiednio posegregowany. Wykonanie porównań w poszczególnych parach pozostawiło badany ciąg 3 elementów bez zmian. Jednocześnie cały pozostały ciąg jest już ułożony w zadanej kolejności. Zatem w ostatnim kroku dokonujemy już tylko jednego porównania. Wynik pracy w tym kroku również musi pozostawić ciąg bez zmian.



Przedstawiony algorytm sortowania bąbelkowego ma dokładnie taką samą złożoność obliczeniową jak algorytm sortowania przez wstawianie. Przedstawiony algorytm sortowania bąbelkowego możemy zaimplementować w postaci kodu w następujący sposób.

```
#include<stdio.h>
```

```
int main()
```

```
{
```

```
    int n, i, j, temp, a[1024];
```

```
    printf("Podaj n:");
```

```
    scanf("%d", &n);
```

```
    for(i = 0; i < n; i++)
```

```
    {
```

```
        printf("Podaj a[%d]=", i);
```

```
        scanf("%d", &a[i]);
```

```
    }
```

```
    for( i = 0; i < n-1; i++)
```

```
        for( j = 1; j < n-i; j++)
```

```
            if(a[j-1]>a[j])
```

```
            {
```

```
                temp = a[j];
```

```
                a[j] = a[j-1];
```

```
                a[j-1] = temp;
```

```
            }
```

```
    for(i = 0; i < n; i++)
```

```
        printf("a[%d]=%d\n", i, a[i]);
```

```
    return 0;
```

```
    _getch();
```

```
}
```

```
// Deklaracja bibliotek
```

```
// Deklaracja zmiennych i tablicy
```

```
// Wczytanie wymiaru zadania
```

```
// Wczytanie elementów ciągu liczb
```

```
/* Iteracja zliczająca ilość elementów do  
opuszczenia w posortowanym ciągu liczb */
```

```
// Iteracja licząca przestawienia
```

```
// Porównanie kolejnych par liczb ze sobą
```

```
/* Przestawienie elementów w poszczególnej  
parze, jeżeli jest to konieczne */
```

```
// Wypisanie uporządkowanego ciągu
```

Spróbujmy teraz udoskonalić napisaną wersję naszego algorytmu sortowania bąbelkowego. Napisany algorytm zmienimy tak, aby operował na tablicy wykorzystując do tego odpowiednią funkcję zmiany elementów na podstawie wskaźników.

Przykład 47

Napisać sortowanie bąbelkowe wykorzystując funkcję zamiany elementów w tablicy.

```
#include<stdio.h>
```

```
void zamiana(int *pa,int *pb)
```

```
{
```

```
    int temp;
```

```
    temp = *pa;
```

```
    *pa = *pb;
```

```
    *pb= temp;
```

```
}
```

```
int main()
```

```
{
```

```
    int n,i,j;
```

```
    printf("Podaj n:");
```

```
    scanf("%d",&n);
```

```
    int *a = new int [n];
```

```
    for(i=0;i<n;i++)
```

```
    { printf("Podaj a[%d]=",i);
```

```
        scanf("%d",a+i); }
```

```
    for(i = 0; i < n-1; i++)
```

```
        for(j = 1; j < n-i; j++)
```

```
            if(a[j-1] > a[j]) zamiana(a+j-1, a+j);
```

```
    for(i=0;i<n;i++)
```

```
        printf("a[%d]=%d\n",i,a[i]);
```

```
}
```

```
// Deklaracja biblioteki
```

```
// Deklaracja funkcji własnej
```

```
/* Funkcja zamienia wartości dwóch  
zmiennych o podanych adresach pa i pb */
```

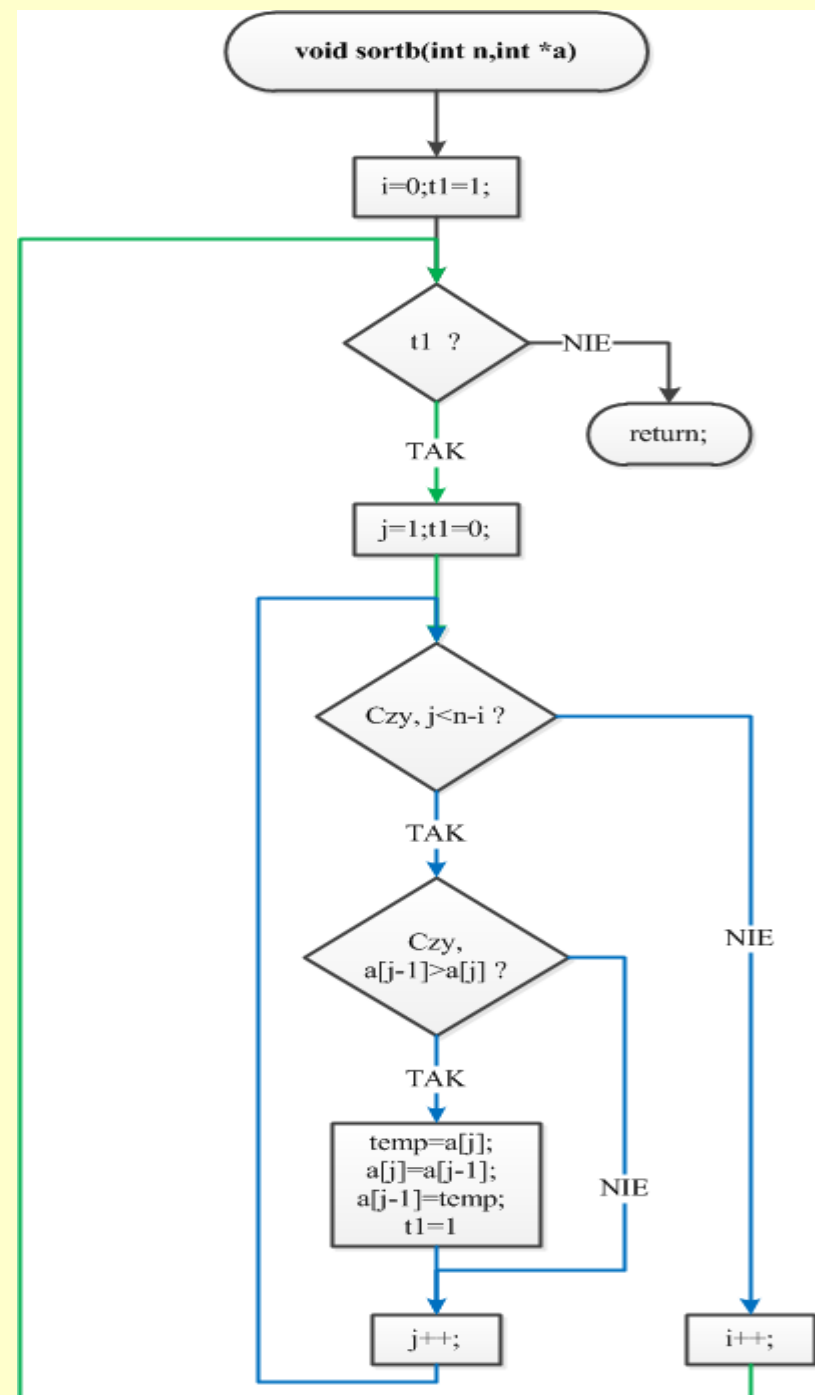
```
// Główna funkcja programu
```

```
// Pobranie ilości elementów
```

```
// Deklaracja tablicy
```

```
/* Wywołanie funkcji zamiana. Adresy  
podajemy wprost. Można to zrobić  
wpisując &a[j-1] i &a[j] */
```

Powyższy algorytm nie uwzględnia możliwości szybszego uporządkowania ciągu liczbowego w wyniku dokonywanych przestawień. Możemy wprowadzić zmienną logiczną kontrolującą proces przestawień i przerwać porządkowanie ciągu w momencie jego uporządkowania. Niestety opisana metoda kontrolująca uporządkowanie ciągu liczbowego dalej pozostaje metodą o złożoności $\vartheta(n^2)$ ze względu na możliwe ciągi do sortowania. Postaramy się teraz napisać program, który sortowałby podany ciąg w sposób bąbelkowy jednak z kontrolą wcześniejszego uporządkowania liczb, co zostanie zaimplementowane na podstawie schematu blokowego pokazanego na kolejnej ilustracji.



```
#include<stdio.h>
```

```
void sortb(int n,int *a)
```

```
{  
    int i, j, t, t1=1;  
    i=0;  
    while(t1)  
    {  
        t1 = 0;  
        for(j = 1; j < n-i; j++)  
            if( a[j-1] > a[j] )  
            {  
                t = a[j];  
                a[j] = a[j-1];  
                a[j-1] = t;  
                t1 = 1;  
            }  
        i++;  
    }  
}
```

```
// Deklaracja biblioteki
```

```
/* Zmienna t1 kontroluje dokonanie  
przestawienia podczas przestawiania  
elementów ciągu liczbowego */
```

```
// Algorytm sortowania bąbelkowego
```

```

int main()
{
    int n, i;
    printf("Podaj n:");
    scanf("%d", &n);
    int *a = new int [ n ];
    for(i = 0; i < n; i++)
    {
        printf("Podaj a[%d]=", i);
        scanf("%d", a+i);
    }
    sortb(n,a);
    for(i=0;i<n;i++)
        printf("a[%d]=%d\n", i, a[i]);
}

```

// Główna funkcja programu

// Wczytanie ilości elementów

// Deklaracja tablicy dla n elementów
 /* Procedura wczytania n elementów
 do zadeklarowanej tablicy */

// Wywołanie funkcji sortującej
 /* Procedura wypisująca posortowane
 elementy ciągu */

Należy zastanowić się, czy można zmniejszyć złożoność czasową algorytmu sortowania. Odpowiedź jest pozytywna i możemy podać algorytm o złożoności czasowej $\vartheta(n \log_2 n)$.