Zawansowane algorytmy sortowania

Sortowanie rekurencyjne quick sort

Zastanówmy się teraz nad wpływem zastosowania metod rekurencyjnych w kodzie programu. Ogólnie rekurencja nie przyspiesza działania programu, ani nie zmniejsza ilości zajmowanego przez program miejsca w pamięci operacyjnej. Jednak postać rekurencyjna jest często bardziej czytelna niż program napisany za pomocą pętli. Bardzo dobrym przykładem jest właśnie algorytm szybkiego sortowania, które napisane za pomocą funkcji rekurencyjnej jest bardzo czytelne. Przedstawmy szybkie sortowanie dobrze znane z literatury [1-2, 37, 39, 66, 73, 93, 102, 112].

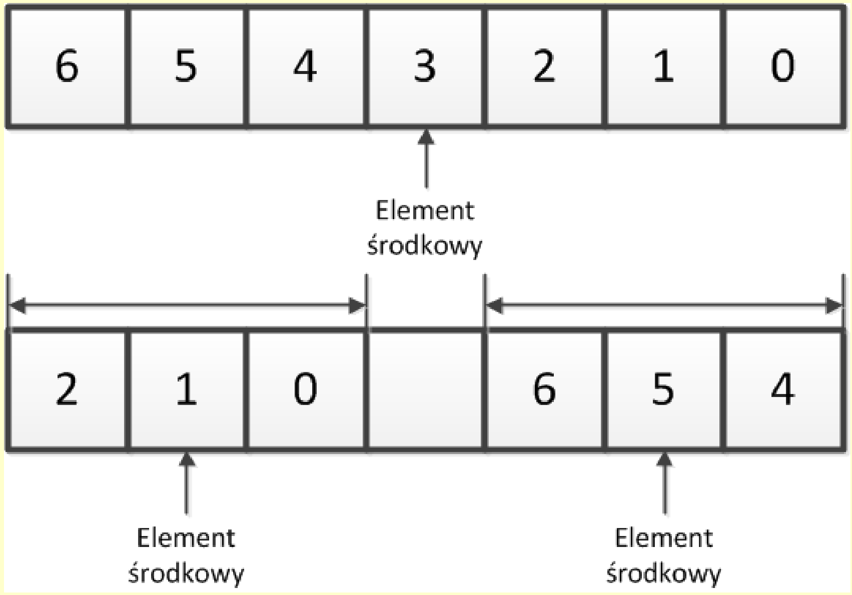
Omówmy najpierw sposób uporządkowania ciągu liczbowego. Metoda quick sort polega na wyborze elementu środkowego. Następnie układamy wszystkie elementy mniejsze od środkowego z lewej strony i jednocześnie wszystkie elementy większe lub równe elementowi środkowemu z prawej jego strony.

Niestety koszt wyznaczenia elementu środkowego bezpośrednio przed rozpoczęciem algorytmu szybkiego sortowania jest zależny od , czyli złożoności czasowej tej metody. Przyjęcie takiego rozwiązania spowodowałoby zwiększenie czasu działania algorytmu, dlatego przyjmuje się indeks elementu środkowego jako wartość losową, wybraną spośród wszystkich możliwych indeksów.

Przyjmijmy, że nasz ciąg przyjmuje następujące indeksy: $a_{lewy}, a_{lewt+1}, \ldots, a_{prawy-1}, a_{prawy}$. Jako indeks elementu środkowego ustalimy średnią arytmetyczną w postaci najmniejszej całkowitej dolnej wartości: $\left\lfloor \frac{a_{lewy}+a_{prawy}}{2} \right\rfloor$. Jeżeli indeksy ciągu rozpoczynają się od zera i kończą na indeksie *n-1*, to indeksem elementu środkowego jest $\left\lfloor \frac{0+n-1}{2} \right\rfloor = \left\lfloor \frac{n-1}{2} \right\rfloor$

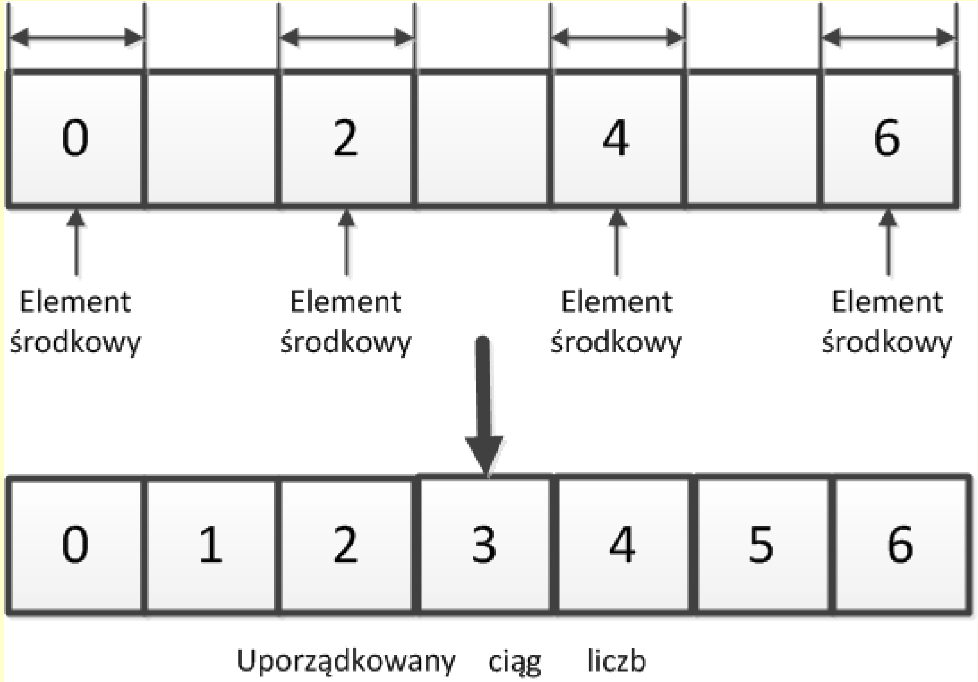
Ciąg elementów poddawany szybkiemu sortowaniu

Przedstawmy porządkowanie następującego ciągu 6,5,4,3,2,1,0 za pomocą szybkiego sortowania.

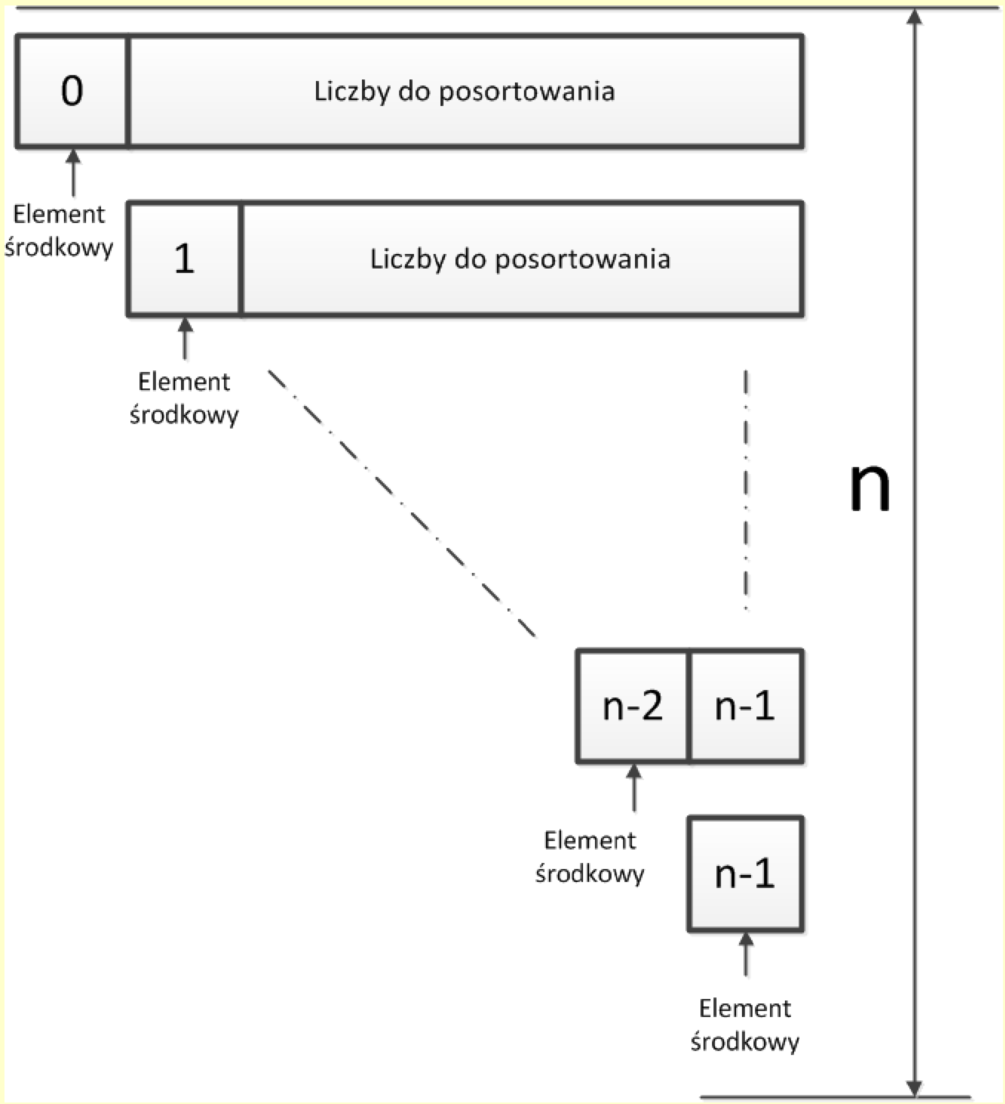


Kolejny etap w sortowaniu quick sort

Wykonując następne przejścia algorytmu szybkiego sortowania otrzymamy następujące wyniki przedstawione na poniższej ilustracji.



W literaturze [1-2, 37, 39, 66, 73, 93, 102, 112] znajdziemy informację, że średni czas sortowania ze względu na przychodzące dane do funkcji szybkiego sortowania wynosi $\vartheta(n \log_2 n)$. Zastanówmy teraz się nad możliwie najgorszym przypadkiem dla szybkiego sortowania. Przypadek taki wystąpi, gdy kolejno przestawiane liczby środkowe będą dosuwane do lewej albo prawej strony. Możemy to zilustrować następująco.



Ponieważ kolejne liczby mogą być dosuwane do lewej albo prawej strony, niekorzystnych dla nas ciągów liczbowych, w których czas sortowania wynosi $\vartheta(n^2)$ jest 2^{n-1}. Stąd prawdopodobieństwo wystąpienia najgorszego przypadku wynosi $P = \frac{2^{n-1}}{n!}$. Dla n=10 prawdopodobieństwo zaistnienia najgorszego przypadku jest bardzo małe i wynosi $P \approx 0{,}00014$. Jednak z praktyki wynika, że przypadek taki może zaistnieć. Należy wtedy zastosować metodę sortowania przez kopcowanie trójdzielne albo omówioną w podrozdziale następnym metodę sortowania przez scalanie. Przedstawmy twierdzenie o średnim czasie sortowania.

Twierdzenie

Algorytm szybkiego sortowania sortuje ciąg n elementowy w średnim czasie $\vartheta(n \log_2 n)$.

Dowód.

Oznaczmy przez $T(n)$ średni czas wykonania algorytmu szybkiego sortowania ciągu n – elementowego (dla $n=1,\ldots,n$). Przyjmijmy że, $T(0)=T(1)=b$ dla pewnej stałej b. Ponieważ element środkowy wybieramy losowo, stąd średnie czasy wykonania wywołania rekurencyjnego algorytmu są odpowiednio równe $T(i\text{-}1)$ oraz $T(n\text{-}i)$. Ponieważ i może być z równym prawdopodobieństwem dowolną wartością od 1 do n oraz przestawienie elementów względem elementu środkowego wymaga czasu cn dla pewnej stałej c, otrzymujemy zależność:

$$T(n) \le cn + \frac{1}{n}\sum_{i=1}^{n}[T(i-1) + T(n-i)] \;\; dla\, n \ge 2$$

Przekształcając otrzymujemy:

$$T(n) \leq cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \qquad (1)$$

Stosując indukcję względem n pokażemy, że dla $n \geq 2$ zachodzi nierówność $T(n) \leq kn\ln n$, gdzie $k = 2c + 2b$ oraz $b = T(0) = T(1)$.

Dla $n = 2$ mamy

$$T(2) \leq 2c + 2b \ \ (2)$$

Ponieważ $\ln 2 > \frac{1}{2}$ nierówność (2) możemy zapisać w postaci:

$$T(2) \leq (2c + 2b) \cdot 2\ln 2$$

Aby przeprowadzić krok indukcyjny, skorzystamy z założenia indukcyjnego i zapiszemy (1) w następującej postaci:

$$T(n) \leq cn + \frac{4b}{n} + \frac{2}{n}\sum_{i=2}^{n-1} ki \ln i \qquad (3)$$

Ponieważ funkcja $i \ln i$ jest wklęsła stąd

$$\sum_{i=2}^{n-1} i \ln i \leq \int_{2}^{n} x \ln x \, dx \leq \frac{n^2 \ln n}{2} - \frac{n^2}{4} \quad (4)$$

Podstawiając (4) do (3), dostajemy

$$T(n) \leq cn + \frac{4b}{n} + kn \ln n - \frac{kn}{2} \qquad (5)$$

Ze względu na to, że $n \geq 2$ i $k = 2c + 2b$ wnioskujemy, że $cn + \frac{4b}{n} \leq \frac{kn}{2}$. Stąd i zależność (5) wynika, że

$$T(n) \leq kn \ln n$$

Przykład 51

Implementacja prezentowanego algorytmu szybkiego sortowania.

#include<stdio.h>	// Deklaracja bibliotek

void zamiana(int *a,int i,int j)
{
   int tmp;
   tmp = a[i];
   a[i] = a[j];
   a[j] = tmp;
}
	// Deklaracja funkcji zamiany
	/* Deklaracja funkcji sortowania szybkiego wymieniającej element i-ty z elementem j-tym tablicy a. */

void sort_s(int *a, int lewy, int prawy)
{
   int i, ostatni;
   if(lewy >= prawy) return;
	/* Przerwanie funkcji sortowania, jeśli pierwszy element z badanych jest większy. */

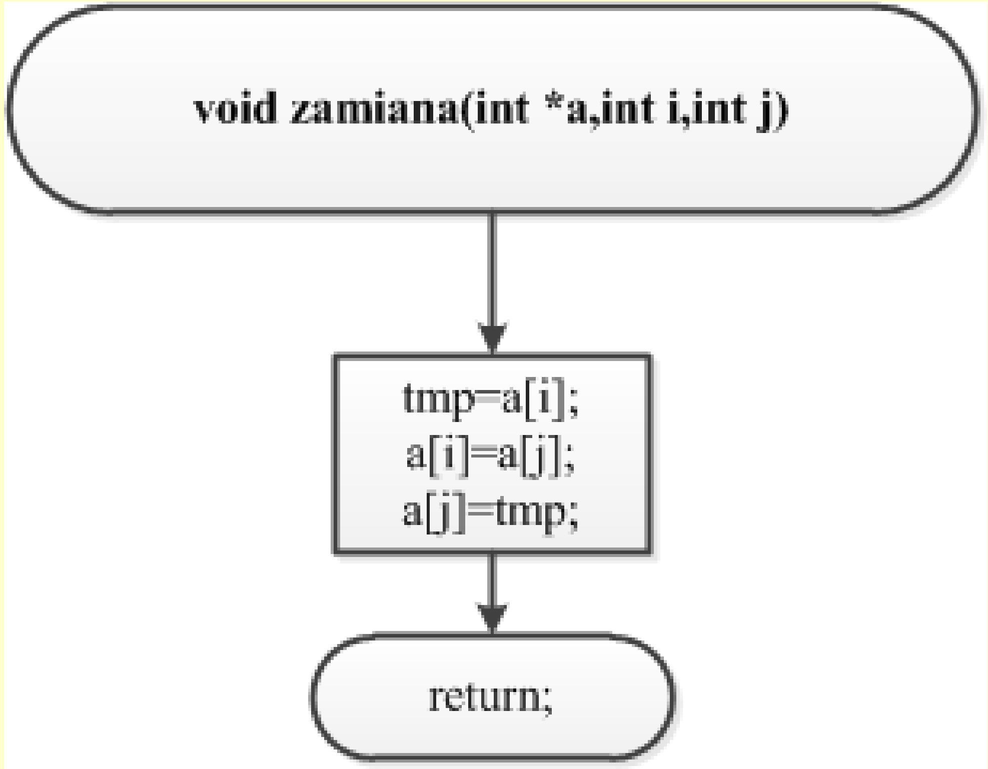
zamiana(a, lewy, (lewy+prawy) /2);	/* Wymiana elementu środkowego z elementem lewym tablicy poprzez funkcję zamiany. */
ostatni = lewy;	/* Przyjęcie lewego elementu jako ostatniego elementu mniejszego od środkowego. */

for(i = lewy + 1; i <= prawy; i++)
      if(a[i] <a [lewy])
            zamiana(a, ++ostatni, i);
   zamiana(a, lewy, ostatni);
	/* Uporządkowanie elementów z lewej strony -elementy mniejsze od środkowego, z prawej strony - elementy większe lub równe elementowi środkowemu. */

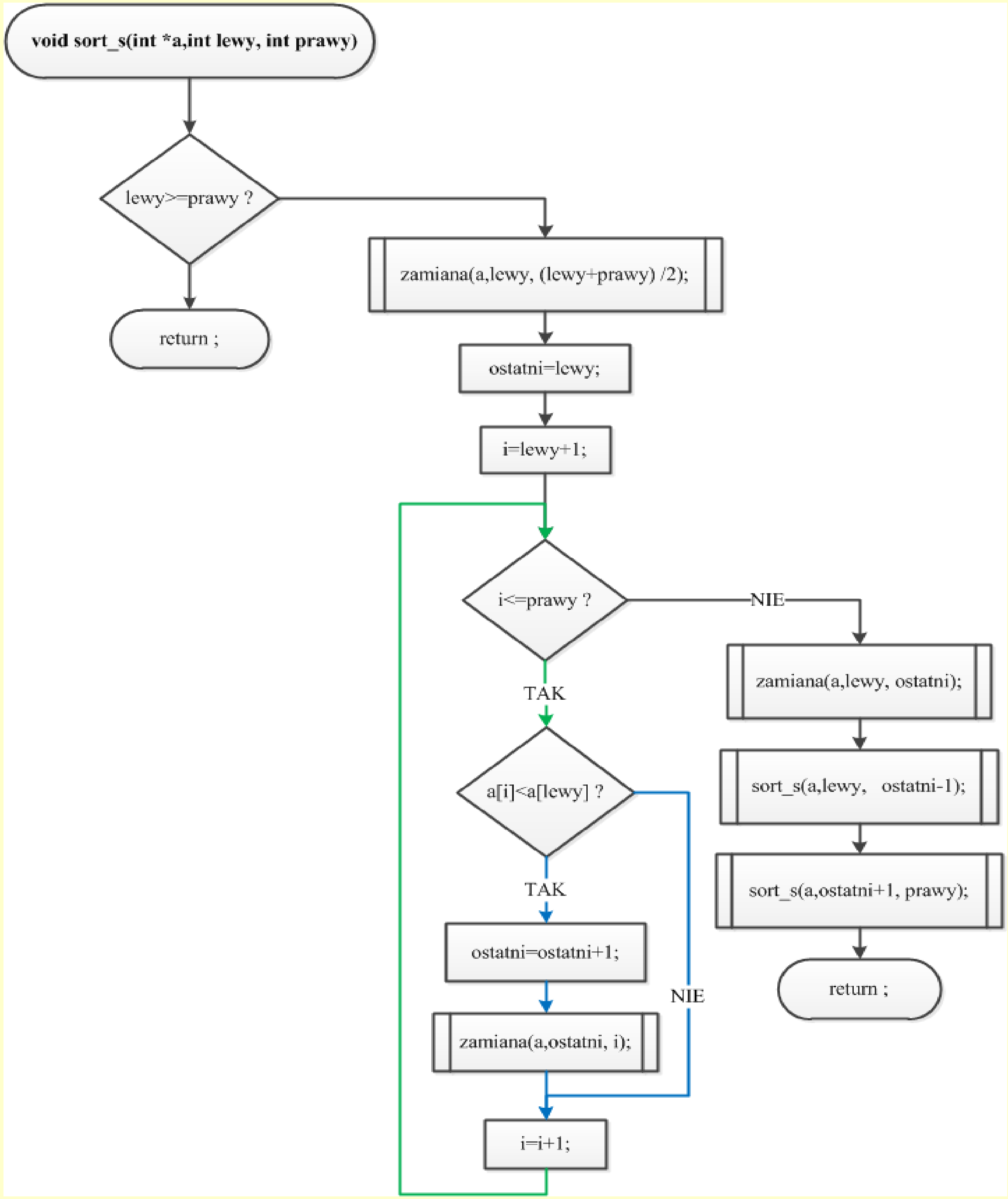
sort_s(a, lewy, ostatni-1);	// Sortowanie lewej części ciągu.
sort_s(a, ostatni+1, prawy); }	// Sortowanie prawej części ciągu.

`int main()`	// Silnik programu.
`{`	
`int n, i;`	
`printf("Podaj n:");`	// Wczytanie ilości elementów ciągu.
`scanf("%d", &n);`	
`int *a=new int [n];`	
`for(i=0;i<n;i++)`	
`{`	// Wczytanie ciągu
`printf("Podaj a[%d]=", i);`	
`scanf("%d", a+i);`	
`}`	
`sort_s(a,0,n-1);`	/* Wywołanie sortowania na wczytanym ciągu do tablicy a. */
`for(i=0;i<n;i++)`	
`printf("a[%d]=%d\n", i, a[i]);`	// Wypisanie posortowanego ciągu
`}`	

Schemat blokowy zamiany elementów w metodzie quick sort



Schemat blokowy sortowania quick sort

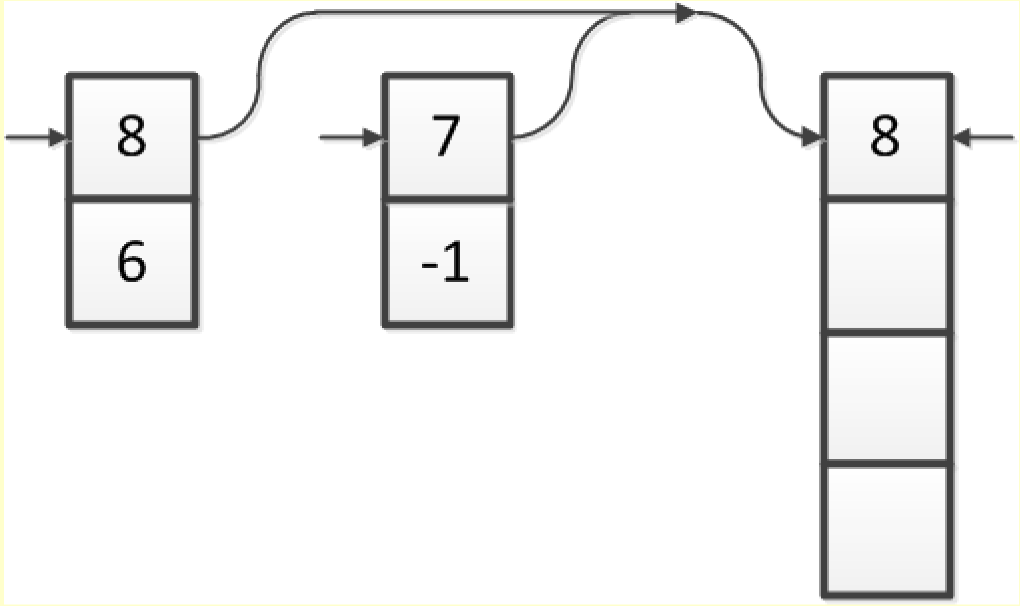


Sortowanie przez scalanie

Na koniec rozdziału przedstawimy algorytm sortowania przez scalanie. Pokażemy sortowanie przez scalanie na przykładzie dwóch uporządkowanych dwuelementowych stosów liczbowych. Ponieważ oba stosy są uporządkowane największe elementy znajdują się na górze każdego ze stosów i kolejno maleją przechodząc w dół stosu. Algorytm sortowania przez scalanie polega na porównaniu elementów na szczycie stosów i wstawianiu ich do innej tablicy. Otrzymujemy w ten sposób uporządkowany ciąg zawierający wszystkie elementy obu ciągów ułożone w porządku hierarchicznym. Popatrzmy na działanie tego algorytmu na przykładzie dwu uporządkowanych stosów. Oba stosy są pokazane na kolejnej ilustracji.

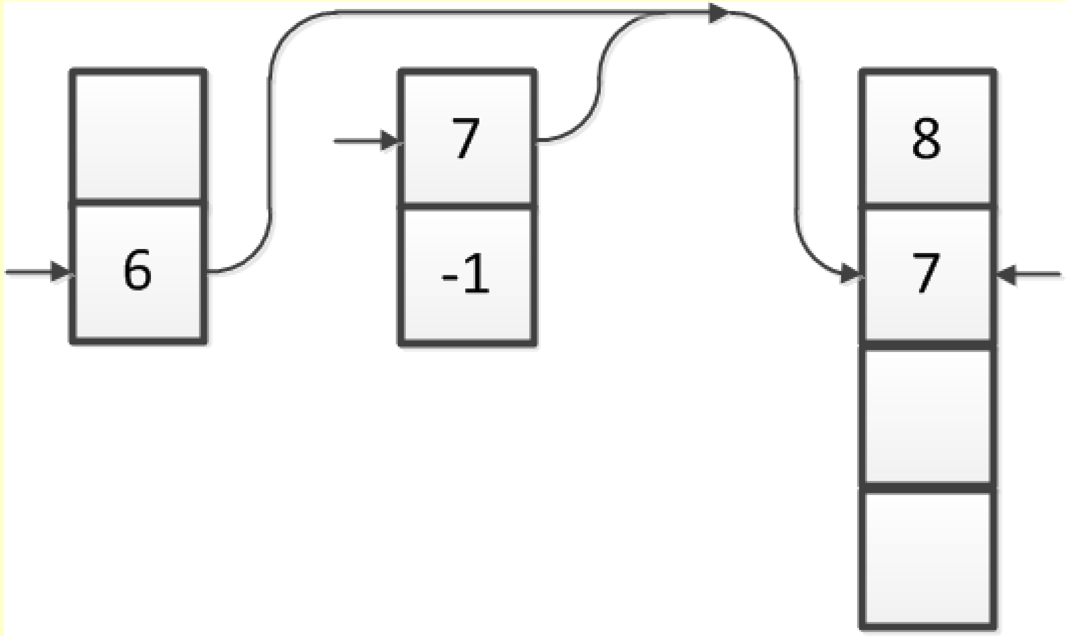
Porównanie stosów i wybór maksimum w kroku pierwszym.

Dokonujemy porównania pierwszych elementów stosów. Wybieramy element 8, ponieważ jest ona największy spośród porównywanych i jednocześnie spośród wszystkich elementów.



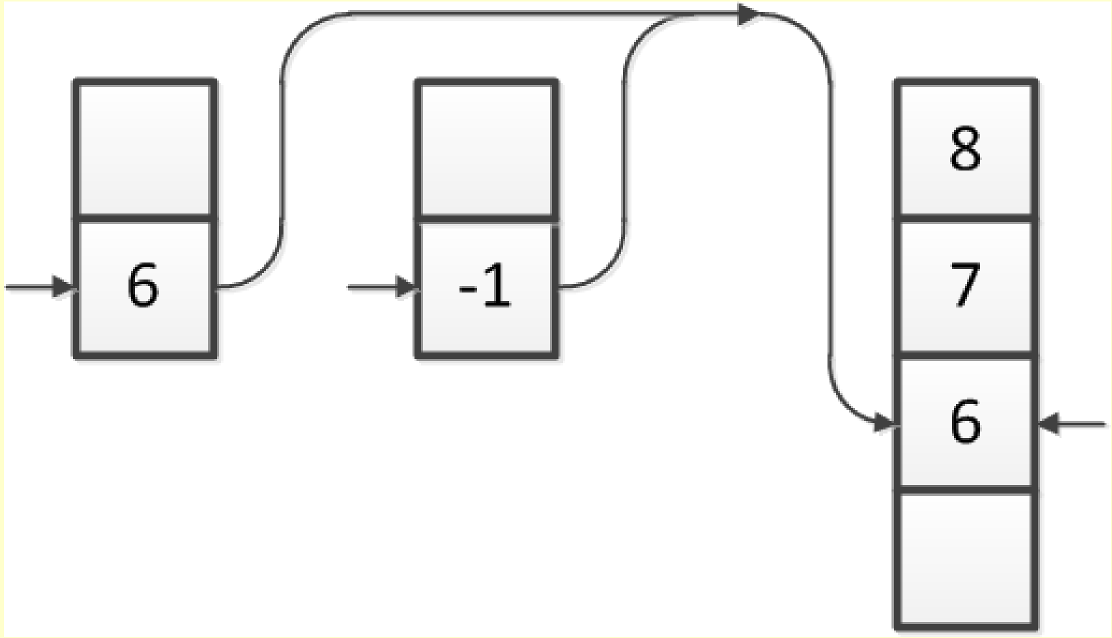
Porównanie elementów na stosach w następnym kroku

Po wyborze maksimum porównujemy kolejne elementy znajdujące się na górze stosów. W tym przypadku największym jest tym razem element z drugiego stosu.



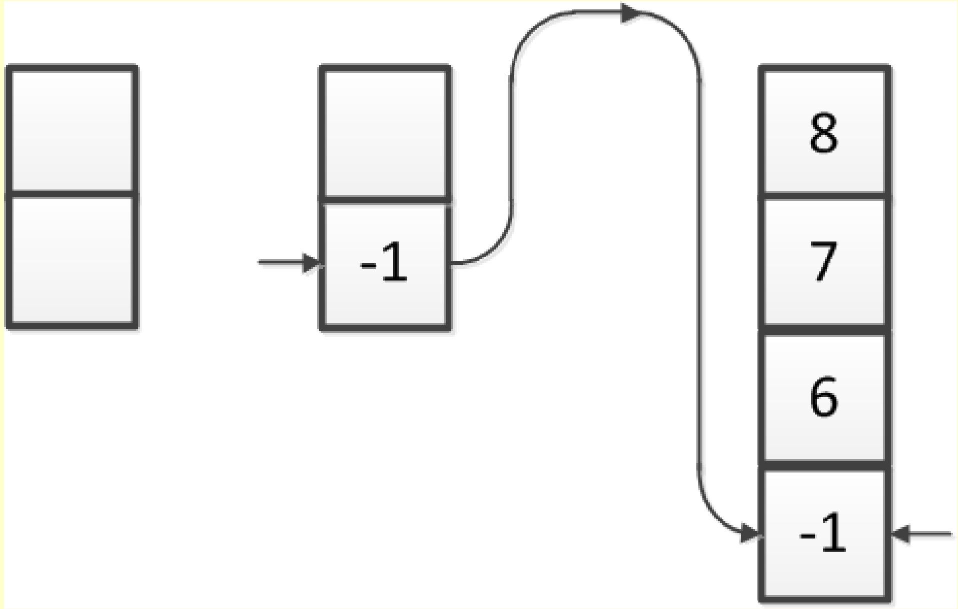
Kolejny krok w algorytmie sortowania przez scalanie

Kolejne porównanie pokazane na poniższej ilustracji zajdzie między elementami

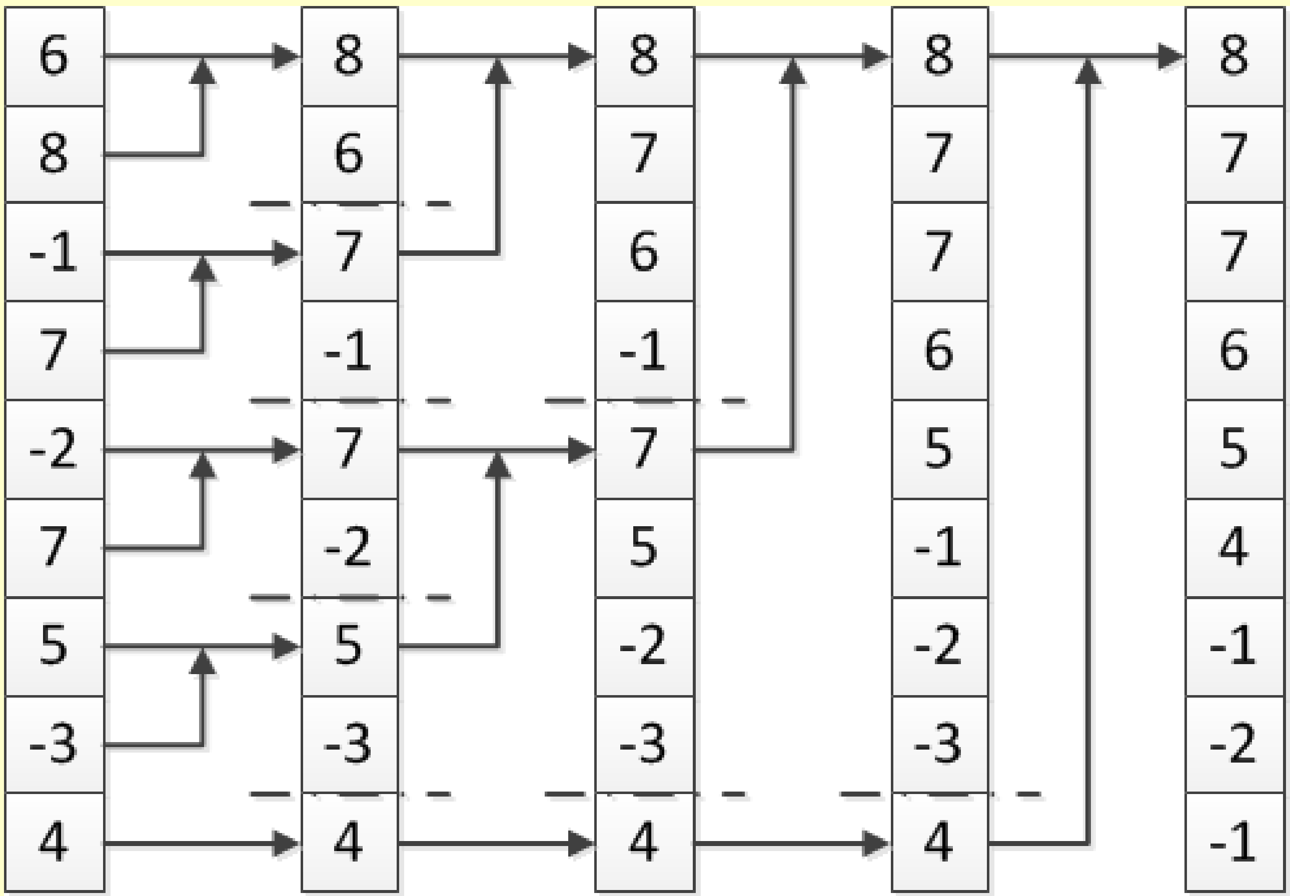


Ostatnie porównanie elementów na obu stosach i ułożenie ich według kolejności

Ostatecznie element najmniejszy spośród wszystkich znajdzie się na końcu nowej tablicy.



Prześledźmy teraz sortowanie ciągu liczb. Operacje możemy przedstawić na poniższym schemacie (elementy na szczycie stosu mają najstarsze indeksy):



Przykład 52

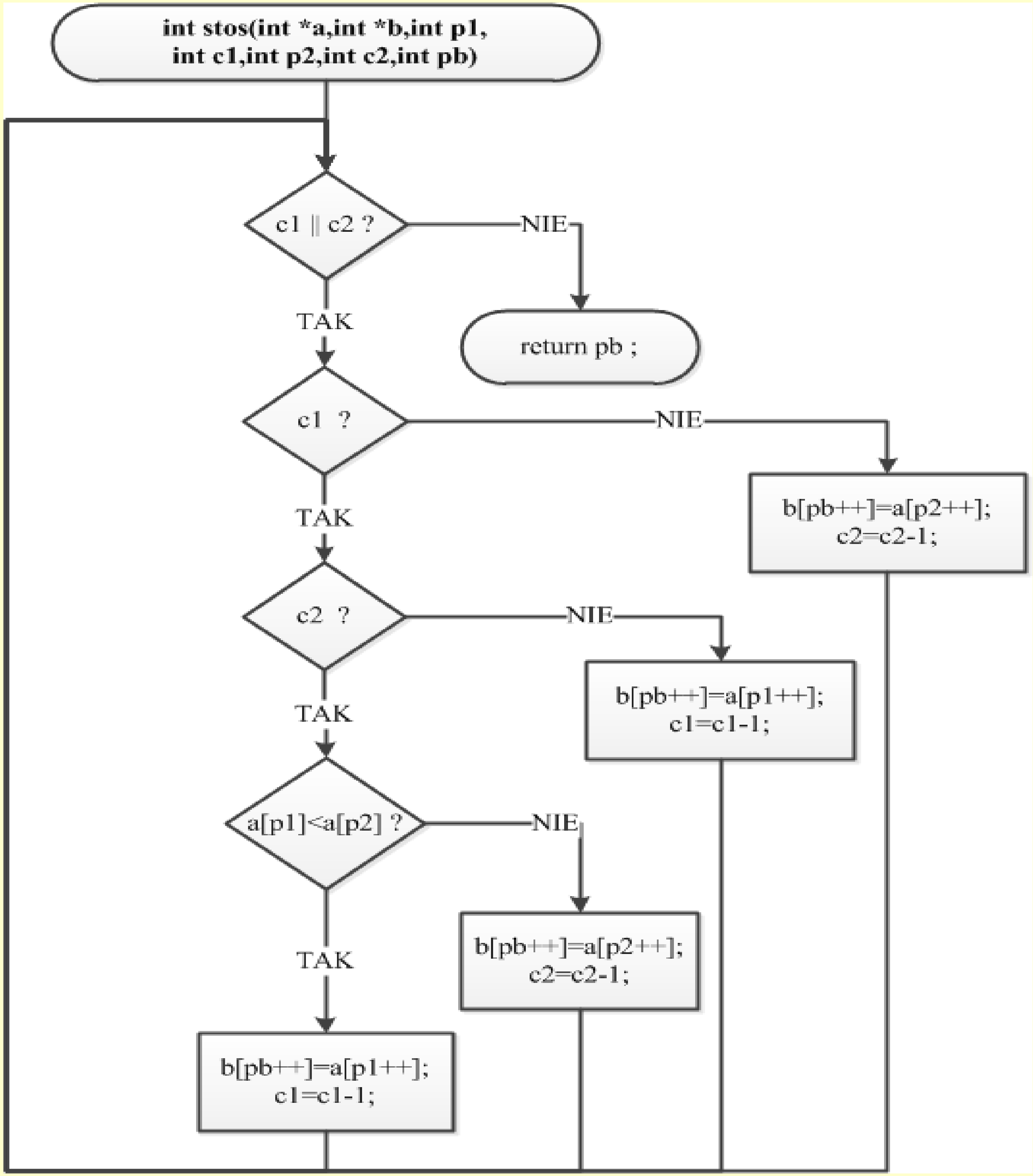
Napisać program czytający ze zbioru *we1.txt* ciąg liczb całkowitych i zapisujący uporządkowany ciąg liczb całkowitych od najmniejszej do największej liczby do zbioru *wy.txt*.

#include<stdio.h>	// Deklaracja bibliotek
#include<stdlib.h>	
void merge(int n,int *a)	/* Funkcja sortowania poprzez scalanie stosów. */
{	
int i,m;	
int p1, p2, pb, c1, c2;	/* Deklarujemy rezerwacje miejsca w buforze. */
int *b = (int *)malloc(sizeof(int)*n);	
for(m = 1;m < n; m *= 2)	
{	
pb=0;	
for(i = 0; i < n; i += (m*2))	
{	

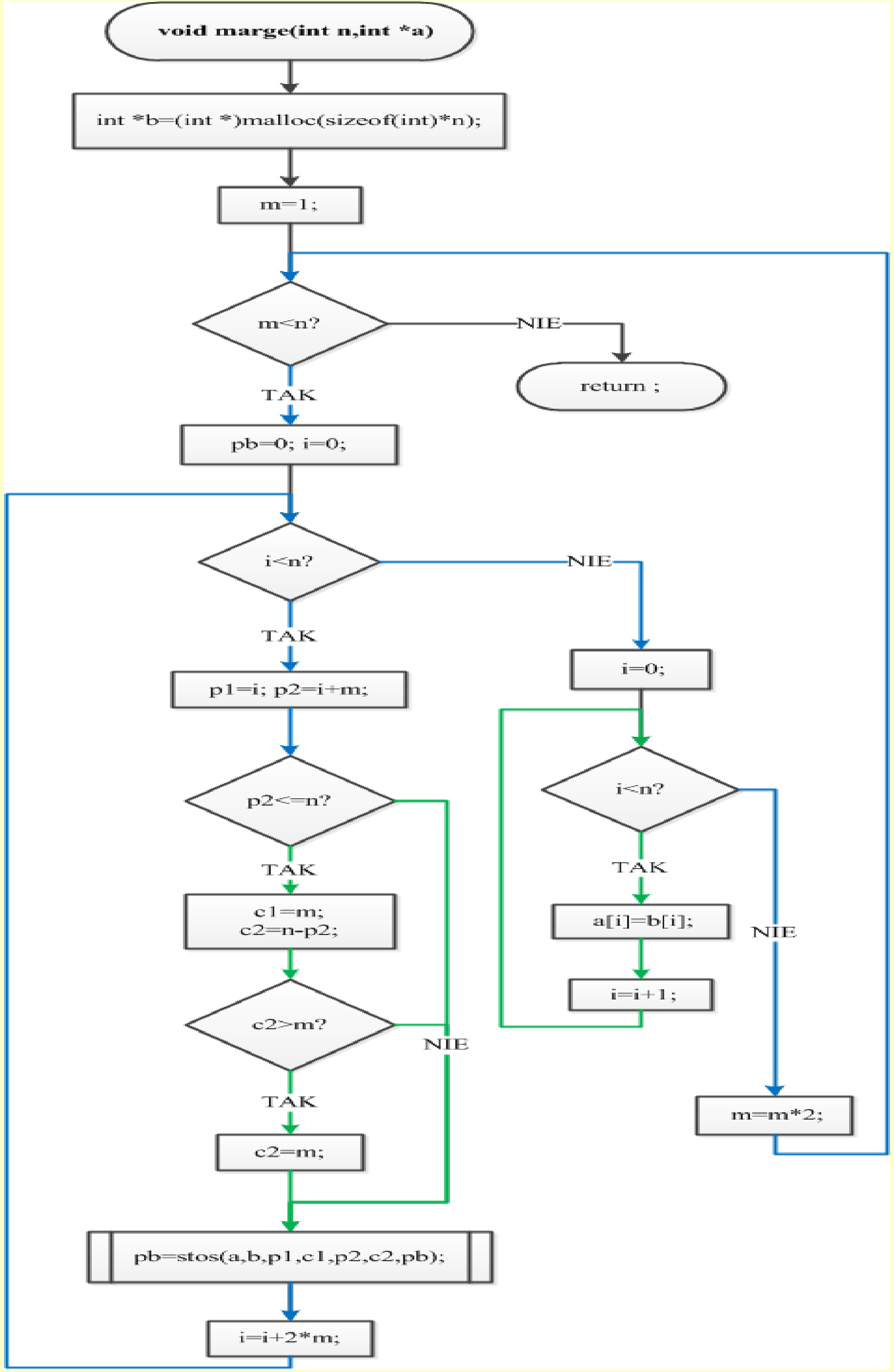
```
p1=i;
 p2=i+m;
 if (p2<=n) {
        c1=m;
         c2=n-p2;
        if (c2>m) c2=m;
        while(c1||c2) {                /* Porównanie elementów na stosie. */
        if (c1) {
        if (c2) {
        if (a[p1]<a[p2]) {
                 b[pb++]=a[p1++];
                 c1--;
                } else {
                 b[pb++]=a[p2++];
                 c2--;
                     }                  // Przepisanie stosu pierwszego.
             } else {
                 b[pb++]=a[p1++];
                 c1--;
                  }                     // przepisanie stosu drugiego
         } else {
             b[pb++]=a[p2++];
             c2--;
              }
      } }
 }
```

`for(i = 0; i < n; i++) a[i] = b[i];`	/* Przepisanie tablicy *b* do tablicy *a*. */
`}`	
`}`	
`int  main(int argc,int **argv)`	// Funkcja główna w programie.
`{`	
`int n = 0, i, e;`	
`FILE  *f1, *f2;`	/* Deklaracja plików tekstowych. */
`if((f1 = fopen("we.txt", "r")) == NULL)`	
`{`	/* Sprawdzamy czy owe pliki istnieją. */
`printf("Nie moge otworzyc pliku wejsciowego\n");`	
`return -1;   }`	/* Próba przeczytania elementu z pliku. */
`int *a = (int *)malloc(0);`	
`while(fscanf(f1, "%d", &e) != EOF)`	/* Dodanie elementu do ciągu liczbowego.
`{    n++;`	*/
`a = (int *)realloc(a,sizeof(int)*n);`	
`a[n-1]=e;  }`	
`merge(n,a);`	/* Wywołanie procedury sortowania. */
`if((f2 = fopen("wy.txt", "w")) == NULL)`	
`{`	// Otwarcie pliku wyjściowego.
`printf("Nie moge otworzyc pliku wyjsciowego\n");`	
`return -1;   }`	
`for (i = 0;  i < n; i++)`	
`fprintf(f2,"a[%d]=%d\n", i, a[i]);`	// Zapis danych do pliku.
`fclose(f1);`	
`fclose(f2);`	// Zamykamy pliki.
`}`	

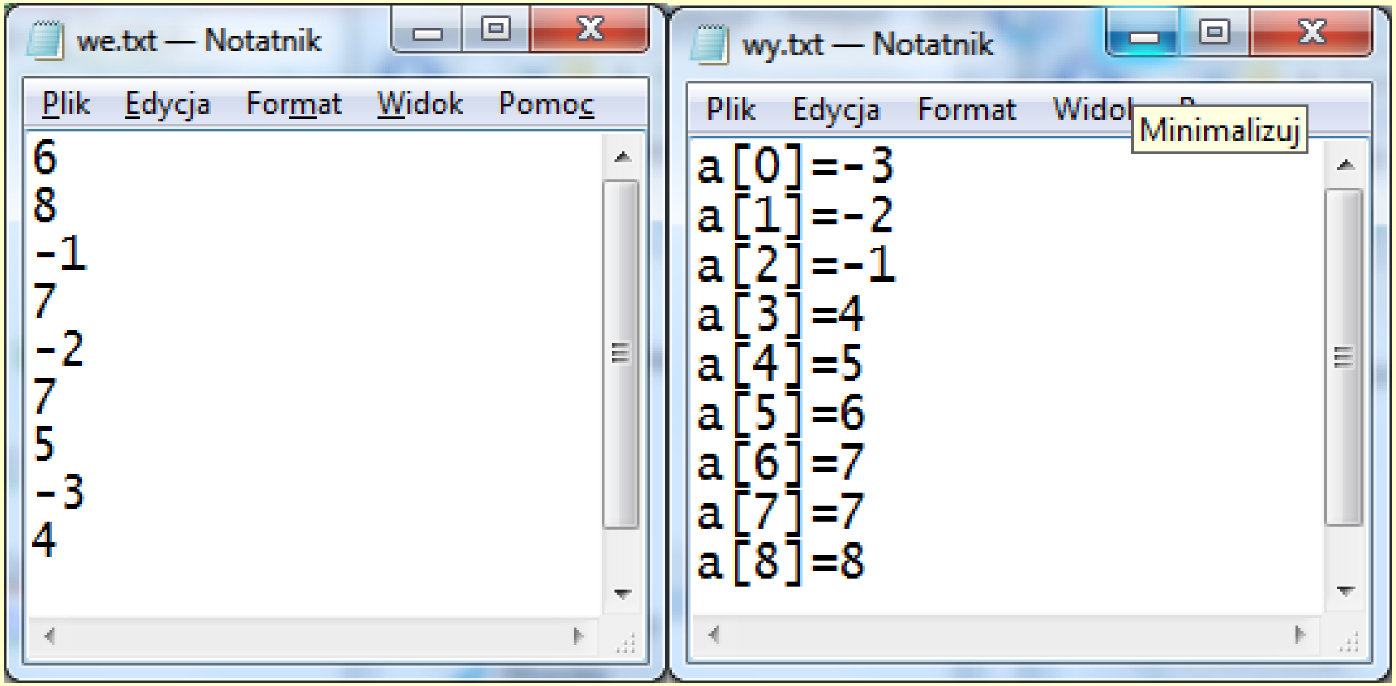
Schemat blokowy pobierania elementów ze stosu w algorytmie sortowania przez scalanie



Schemat blokowy procedury sortowania przez scalanie



Działanie programu sortującego algorytmem przez scalanie na plikach zewnętrznych pokazano na poniższej ilustracji.



Przedstawiony algorytm ma złożoność czasową , natomiast do wykonania operacji na stosach potrzebuje dodatkowej tablicy deklarowanej w funkcji sortującej. Na uwagę Czytelnika zasługuje zaprezentowany sposób czytania liczb z pliku, który automatycznie ustala ilość przeczytanych danych, czyli wymiar zadania. W algorytmie wykorzystano fakt, że funkcja *scanf()* zwraca ilość przeczytanych liczb albo *EOF* w przypadku napotkania końca zbioru.

`int n=0;` `int *a=(int *)malloc(0);`	
`while(fscanf(f1,"%d",&e)!=EOF)` `{`	// Próba przeczytania elementu z pliku.

`n++;`
`a=(int *)realloc(a,sizeof(int)*n);`
`a[n-1]=e;`
`}`
	/* Dodanie elementu do ciągu liczbowego.*/

Istotnym w algorytmie jest fakt, ze funkcja *realloc()* może przemieścić tablicę w pamięci i zwrócić inny adres niż instrukcja *malloc()*. Pisząc funkcję czytania ciągu liczbowego musimy zwrócić wskaźnik do tablicy i ilość elementów przechowywanych w tablicy. Problem ten zostanie poruszony w następnym rozdziale.