2.8. Sprawdzanie monotoniczności ciągu liczbowego

Definicja

Dla ciągu liczbowego można określić jego monotoniczność. Mówimy, że **ciąg jest monotoniczny**, jeśli każda para kolejnych wyrazów tego ciągu spełnia określone warunki dotyczące uporządkowania.

Załóżmy, że dany mamy n-wyrazowy ciąg liczbowy $(a_n) = (a_0, a_1, ..., a_{n-1})$. Ciąg ten jest monotoniczny, jeżeli dla $i = 0, 1, \ldots, n-2$ spełniony jest jeden z warunków podanych w tabeli 2.2.

Tabela 2.2. Typy ciągów monotonicznych

Typ ciągu monotonicznego	Warunek	Przykład ciągu liczbowego
rosnący	$a_i < a_{i+1}$ (czyli $a_{i+1} - a_i > 0$)	$(a_6) = (1, 2, 7, 9, 11, 15)$
malejący	$a_i > a_{i+1}$ (czyli $a_{i+1} - a_i < 0$)	$(a_7) = (8, 7, 6, 4, 3, 1, 0)$
nierosnący	$a_i \geq a_{i+1}$ (czyli $a_{i+1} - a_i \leq 0$)	$(a_8) = (11, 7, 7, 6, 5, 4, 4, 2)$
niemalejący	$a_i \leq a_{i+1}$ (czyli $a_{i+1} - a_i \geq 0$)	$(a_9) = (2, 3, 3, 4, 5, 7, 7, 7, 8)$

Ciąg liczbowy, którego nie można zaliczyć do żadnego z podanych typów, jest ciągiem niemonotonicznym. Przykładem takiego ciągu jest $(a_9) = (1, 4, 8, 3, 7, 1, 2, 3, 10)$.

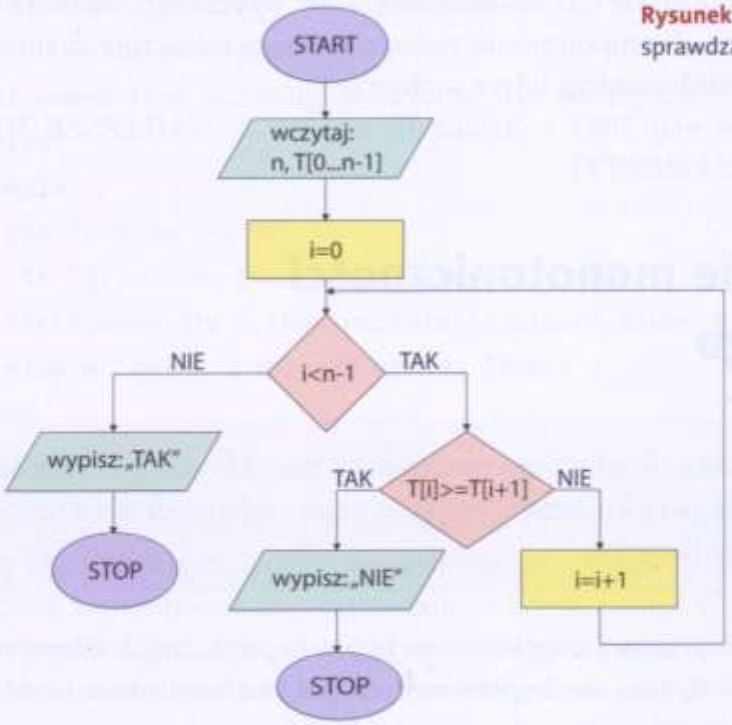
Skonstruujmy **algorytm sprawdzający, czy *n*-wyrazowy ciąg liczb całkowitych jest ciągiem rosnącym**, w postaci schematu blokowego (patrz rysunek 2.15) oraz programów w językach C++ i Pascal. Dodatkowo na płycie CD znajduje się realizacja tego algorytmu wykonana za pomocą arkusza kalkulacyjnego (*arkusz2_8.xls*, *arkusz2_8.ods*).

Specyfikacja:

Dane: Liczba naturalna: $n > 1$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby całkowite: $T[0...n-1]$.

Wynik: Komunikat informujący, czy ciąg zapisany w tablicy T jest rosnący.



Rysunek 2.15. Schemat blokowy algorytmu sprawdzającego, czy ciąg jest rosnący

Funkcja w języku C++ (prog2_20.cpp):

```
bool rosnacy (int T[], int n)
{
 for (int i=0;i<n-1;i++)
  if (T[i]>=T[i+1]) return false;
 return true;
}
```

Funkcja w języku Pascal (prog2_20.pas):

```
function rosnacy (T: tablica; n: integer): boolean;
var i: integer;
    pom: boolean;
```

```
begin
 pom:=true;
 i:=0;
 while (i<n-1)and(pom) do
 begin
  if T[i]>=T[i+1] then pom:=false;
  i:=i+1
 end;
 rosnacy:=pom
end;
```

Przeanalizujmy złożoność przedstawionego algorytmu. Operacją dominującą jest tutaj porównanie. Liczba tych działań wynosi dokładnie $n-1$ i nie zależy od wartości wyrazów sprawdzanego ciągu. Złożoność tego algorytmu jest więc liniowa, rzędu $O(n)$.

Zadanie 2.43. Podaj specyfikację zadania i skonstruuj algorytm w postaci listy kroków i programu sprawdzający, czy dany n-wyrazowy ciąg liczb całkowitych wprowadzanych z klawiatury jest malejący.

Zadanie 2.44. Podaj specyfikację zadania i skonstruuj algorytm w postaci schematu blokowego sprawdzający, czy dany n-wyrazowy ciąg liczb rzeczywistych wprowadzanych z klawiatury jest niemalejący.

Zadanie 2.45. Podaj specyfikację zadania i skonstruuj algorytm w postaci programu sprawdzający monotoniczność wprowadzanego z klawiatury n-wyrazowego ciągu liczb rzeczywistych. Wynikiem działania algorytmu powinny być komunikaty określające ten ciąg: „rosnący", „malejący", „nierosnący", „niemalejący", „niemonotoniczny". Na przykład dla ciągu (0, 0, 0, 0, 0) poprawną odpowiedzią będą komunikaty: „nierosnący", „niemalejący".

2.9. Sortowanie ciągu liczbowego

Sortowanie ciągu liczbowego polega na uporządkowaniu jego wyrazów rosnąco, malejąco, nierosnąco lub niemalejąco, a więc tak, aby ciąg niemonotoniczny został przekształcony w monotoniczny. Przedstawiając kolejne algorytmy sortowania, będziemy dążyć do przekształcenia ciągu $T[0...n-1]$ w ciąg monotoniczny, którego każda para kolejnych elementów będzie spełniać warunek $T[i] \leq T[i+1]$, dla $i = 0, 1, ..., n-2$. Celem naszym będzie więc uzyskanie ciągu niemalejącego, ewentualnie rosnącego.

Istnieją różne typy metod sortujących. Najpopularniejszą grupę algorytmów **stanowią metody oparte na porównywaniu elementów ciągu** i ich zamianie. Należą do nich: porządkowanie bąbelkowe, przez wybór, przez wstawianie, przez scalanie i metoda szybka.

Dwie ostatnie opierają się na technice „dziel i zwyciężaj", zostały więc omówione w podrozdziale 2.10, „Zastosowanie metody »dziel i zwyciężaj«". Pozostałe metody zaliczane do tej grupy przedstawiono w punkcie 2.9.1, „Metody sortowania przez porównania".

Złożoność czasowa algorytmów wykorzystujących operację porównania zawiera się w zakresie od $O(n^2)$ do $O(n \log n)$. Metody, których złożoność jest rzędu $O(n^2)$, to: sortowanie bąbelkowe, przez wybór, przez wstawianie i szybkie. Złożoność liniowo-logarytmiczną ma algorytm oparty na technice „dziel i zwyciężaj" — sortowanie przez scalanie.

Innym typem metod sortowania, które **nie stosują porównywania elementów ciągu**, jest porządkowanie realizowane z wykorzystaniem dodatkowej tablicy pomocniczej, w której na przykład zliczane są elementy ciągu. Do takich algorytmów należą porządkowanie przez zliczanie i sortowanie kubełkowe. Wymienione metody zostały opisane w punkcie 2.9.2, „Sortowanie w czasie liniowym". Złożoność czasowa tych algorytmów jest rzędu $O(n)$, pod warunkiem jednak, że liczba możliwych wartości wyrazów sortowanego ciągu nie przekracza n. Efektywność tych algorytmów wzrasta, gdy przedział wartości sortowanego ciągu maleje.

Definicja

Metody rozwiązujące problem porządkowania ciągu liczbowego z wykorzystaniem jednej tablicy nazywane są **algorytmami sortującymi w miejscu** (łac. *in situ*).

Oznacza to, że podczas realizacji algorytmu sortowanie wykonywane jest bezpośrednio w tablicy, w której zapisany jest ciąg. W tablicy tej ciąg źródłowy po zamianie wybranych par elementów przekształcany jest w posortowany ciąg wynikowy.

Do grupy algorytmów sortujących w miejscu nie należy sortowanie przez scalanie, które wymaga wykorzystania dodatkowej tablicy. Zaliczyć do tej grupy można jednak sortowanie przez wybór.

W niektórych algorytmach sortowania **efektywność działania zależy od danych wejściowych**. Istnieją również metody, których złożoność zawsze jest taka sama, nie wpływa na nią fakt, że wczytywany ciąg jest już posortowany (więc nie wymaga żadnych zamian wyrazów), czy też jest ciągiem posortowanym odwrotnie, co wymaga największej liczby zamian.

Definicja

Stabilnym algorytmem porządkowania nazywamy metodę, która nie zmienia kolejności względem siebie tych wyrazów ciągu, które mają tę samą wartość.

W realizacji wielu algorytmów okazuje się, że wcześniejsze posortowanie ciągu ułatwia wykonanie zadania. Na przykład binarne przeszukiwanie ciągu uporządkowanego, omó-

wione w punkcie 1.7.1, „Przeszukiwanie binarne ciągu uporządkowanego", ma niższą złożoność niż algorytm przeszukiwania liniowego podany w podrozdziale 2.5, „Przeszukiwanie ciągu liczbowego — metody liniowe", w którym musimy przejrzeć wszystkie wyrazy ciągu.

2.9.1. Metody sortowania przez porównania

Porządkowanie bąbelkowe

Sortowanie bąbelkowe (ang. *bubble sort*) realizowane jest przez porównywanie wszystkich kolejnych par wyrazów w ciągu w celu znalezienia maksimum. W tablicy $T[0...n-1]$ sprawdzane są więc pary elementów $T[i]$ z $T[i+1]$, dla $i = 0, 1, ..., n-2$. Po każdym przejściu maksymalny wyraz aktualnie przeglądanego ciągu przesuwany jest na koniec. Wyraz ten jest już posortowany, więc w kolejnej fazie przeglądamy już krótszy ciąg. W pierwszym kroku przeglądamy ciąg n-wyrazowy, wykonujemy więc $n-1$ porównań. W drugim kroku ciąg zawiera $n-1$ wyrazów, porównań będzie więc tylko $n-2$. Na końcu mamy tylko dwa wyrazy i jedno porównanie.

Omawiany algorytm jest stabilny i realizuje sortowanie ciągu metodą w miejscu.

Przykład 2.31.

Prześledźmy kolejne kroki sortowania na przykładzie liczbowym. Uporządkujemy rosnąco ciąg liczbowy: (7, 3, 2, 9, 1), stosując algorytm bąbelkowy.

	7	3	2	9	1
Krok 1.	**3**	**7**	2	9	1
	3	**2**	**7**	9	1
	3	2	7	**1**	**9**
Krok 2.	**2**	**3**	7	1	9
	2	3	**1**	**7**	9
Krok 3.	2	**1**	**3**	7	9
Krok 4.	**1**	**2**	3	7	9

Każdy krok algorytmu to kolejne przejście przez ciąg, który jest coraz krótszy. Szarym tłem wyróżniono posortowane wyrazy ciągu liczbowego. Niebieskim kolorem zaznaczono pary tych wyrazów ciągu, które zostały zamienione.

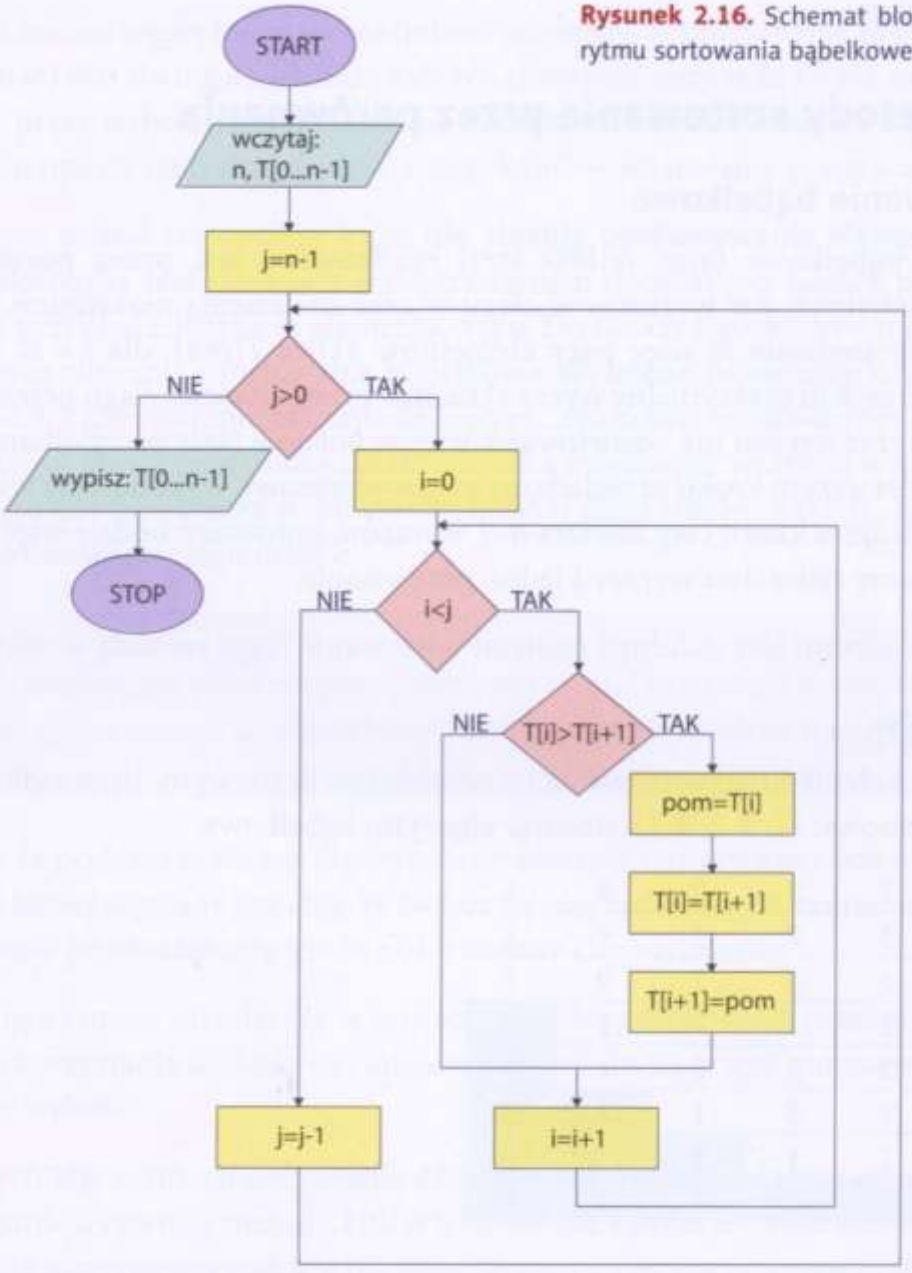
Skonstruujmy **algorytm realizujący sortowanie bąbelkowe** w postaci listy kroków, schematu blokowego (patrz rysunek 2.16) oraz programów w językach C++ i Pascal.

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0\ldots n-1]$.



Rysunek 2.16. Schemat blokowy algorytmu sortowania bąbelkowego

Lista kroków:

Krok 0. Wczytaj n, $T[0\ldots n-1]$.

Krok 1. Dla kolejnych wartości j: $n-1$, $n-2$, …, 1, wykonuj krok 2.

Krok 2. Dla kolejnych wartości i: 0, 1, …, $j-1$, wykonuj krok 3.

Krok 3. Jeśli $T[i] > T[i+1]$, zamień miejscami te elementy tablicy.

Krok 4. Wypisz elementy tablicy $T[0\ldots n-1]$. Zakończ algorytm.

Funkcja w języku C++ (prog2_21.cpp):

```
void sortuj (double T[], int n)
{
 double pom;
 for (int j=n-1;j>0;j--)
  for (int i=0;i<j;i++)
   if (T[i]>T[i+1])
   {
    pom=T[i];
    T[i]=T[i+1];
    T[i+1]=pom;
   }
}
```

Procedura w języku Pascal (prog2_21.pas):

```
procedure sortuj (var T: tablica; n: integer);
var j, i: integer;
    pom: real;
begin
for j:=n-1 downto 1 do
 for i:=0 to j-1 do
  if T[i]>T[i+1] then
  begin
   pom:=T[i];
   T[i]:=T[i+1];
   T[i+1]:=pom
  end
end;
```

Operacją dominującą w algorytmie sortowania bąbelkowego jest porównanie. Wyznaczmy więc złożoność czasową tej metody dla n-wyrazowego ciągu. Pętla zewnętrzna przedstawionego algorytmu powtarzana jest $n-1$ razy. W pętli wewnętrznej, realizującej przeglądanie ciągu w celu znalezienia i przesunięcia na koniec wyrazu maksymalnego, wykonywanych jest najpierw $n-1$ porównań, w kolejnym kroku $n-2$, a w ostatnim przebiegu tylko jedno porównanie. Łącznie otrzymujemy:

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2} \tag{2.23}$$

operacji dominujących. Wadą tego algorytmu jest to, że liczba porównań nie zależy od wartości wyrazów ciągu liczbowego. Złożoność omówionej metody jest więc rzędu

$O(n^2)$. Liczba wykonywanych zamian wyrazów sortowanego ciągu może być jednak tutaj różna:

$$\text{od } 0 \text{ do } \frac{n(n-1)}{2}.$$

Algorytm sortowania bąbelkowego można zmodyfikować w taki sposób, aby zmniejszyć liczbę porównań. Jedną z takich możliwości jest zakończenie algorytmu w przypadku stwierdzenia, że ciąg jest już posortowany. Warunek ten będzie spełniony, jeśli w kolejnym kroku nie zostanie wykonana żadna zamiana.

Zadanie 2.46. Wprowadź zmiany w algorytmie sortowania bąbelkowego prowadzące do polepszenia jego złożoności czasowej.

Porządkowanie przez wybór

Metoda sortowania przez wybór (ang. *selection sort*) polega na wyborze minimum w ciągu. Następnie znaleziona liczba zamieniana jest z pierwszym wyrazem przeglądanego ciągu. Przestawiony wyraz jest już posortowany, więc w kolejnym kroku analizujemy krótszy ciąg.

Algorytm realizuje porządkowanie ciągu w miejscu. Stabilność tej metody zależy od sposobu realizacji. W przedstawionym w tym punkcie algorytmie kolejność wyrazów o tej samej wartości zmienia się w trakcie realizacji metody jednak w posortowanym ciągu jest już zachowana. Wynika stąd, że podany algorytm jest stabilny.

Przykład 2.32.
Prześledźmy działanie algorytmu na przykładzie liczbowym. Uporządkujmy rosnąco ciąg: (8; 2; 0; 1; 6).

	8	2	0	1	6
Krok 1.	**0**	2	**8**	1	6
Krok 2.	0	**1**	8	**2**	6
Krok 3.	0	1	**2**	**8**	6
Krok 4.	0	1	2	**6**	**8**

Szarym tłem zaznaczono wyrazy ciągu, które są już posortowane. Kolorem niebieskim wyróżniono wyrazy zamienione w danym kroku.

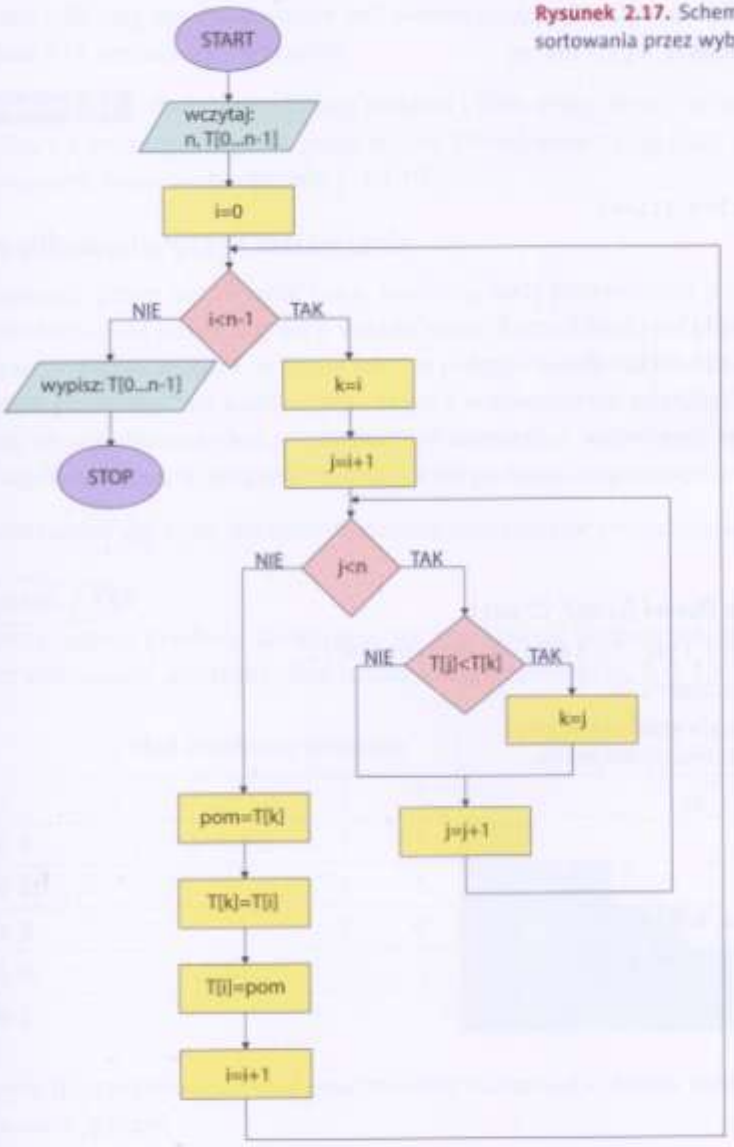
Skonstruujmy **algorytm wykonujący sortowanie przez wybór** w postaci listy kroków, schematu blokowego (patrz rysunek 2.17) oraz programów w językach C++ i Pascal.

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.



Rysunek 2.17. Schemat blokowy algorytmu sortowania przez wybór

Lista kroków:

Krok 0. Wczytaj n, $T[0...n-1]$.

Krok 1. Dla kolejnych wartości i: $0, 1, \ldots, n-2$, wykonuj kroki 2. – 3., a następnie przejdź do kroku 4.

Krok 2. Przypisz k = indeks minimalnego elementu w tablicy $T[i...n-1]$.

Krok 3. Zamień miejscami elementy tablicy $T[i]$ i $T[k]$.

Krok 4. Wypisz elementy tablicy $T[0\ldots n-1]$. Zakończ algorytm.

Funkcja w języku C++ *(prog2_22.cpp)*:

```
void sortuj (double T[], int n)
{
 int k;
 double pom;
 for (int i=0;i<n-1;i++)
 {
  k=i;
  for (int j=i+1;j<n;j++)
   if (T[j]<T[k]) k=j;
  pom=T[k];
  T[k]=T[i];
  T[i]=pom;
 }
}
```

Procedura w języku Pascal *(prog2_22.pas)*:

```
procedure sortuj (var T: tablica; n: integer);
var i, j, k: integer;
    pom: real;
begin
for i:=0 to n-2 do
 begin
  k:=i;
  for j:=i+1 to n-1 do
   if T[j]<T[k] then k:=j;
  pom:=T[k];
  T[k]:=T[i];
  T[i]:=pom
 end
end;
```

Złożoność metody sortowania przez wybór określamy przez wyznaczenie liczby wystąpień operacji dominującej, którą jest porównanie. W n-wyrazowym ciągu w kolejnych $n-1$ fazach algorytmu szukamy minimum w coraz krótszym ciągu, stąd liczba porównań wynosi tutaj:

$$(n-1)+(n-2)+\ldots+1=\frac{n(n-1)}{2}. \qquad (2.24)$$

Złożoność czasowa tej metody jest więc rzędu $O(n^2)$. Liczba zamian wyrazów w tym algorytmie może być jednak różna: od 0 do $n-1$. Zależne jest to od danych wejściowych. Jeśli sortujemy ciąg, który jest już uporządkowany, nie dokonujemy żadnej zamiany. Natomiast jeśli ciąg uporządkowany jest odwrotnie, czyli nierosnąco, to wykonujemy dokładnie $n-1$ zamian jego wyrazów.

Zadanie 2.47. Podaj specyfikację zadania i skonstruuj algorytm w postaci programu sortujący nierosnąco metodą przez wybór 10-wyrazowy ciąg liczb całkowitych wygenerowanych losowo z przedziału $[-10, 100]$.

Porządkowanie przez wstawianie

Sortowanie przez wstawianie (ang. *insertion sort*) przypomina porządkowanie kart. Po rozdaniu talii każdy z graczy układa swoje karty, biorąc po jednej i wstawiając od razu we właściwe miejsce. Właśnie na tym polega metoda sortowania przez wstawianie. Kolejno pobierany jest każdy wyraz ciągu i wstawiany we właściwym miejscu. W metodzie tej wykonywane jest porównanie w momencie wstawiania wyrazu do ciągu już uporządkowanego, w celu znalezienia dla niego właściwego miejsca.

Przedstawiony algorytm jest metodą stabilną oraz realizuje porządkowanie ciągu w miejscu.

Przykład 2.33.

Przeanalizujmy przebieg sortowania na przykładzie liczbowym. Stosując sortowanie przez wstawianie, uporządkujmy rosnąco ciąg liczbowy: (7, 3, 0, 1, 5).

	Stąd pobieramy elementy:					Tutaj wstawiamy elementy, jednocześnie sortując:				
	7	3	0	1	5					
Krok 1.		3	0	1	5	7				
Krok 2.			0	1	5	3	7			
Krok 3.				1	5	0	3	7		
Krok 4.					5	0	1	3	7	
Krok 5.						0	1	3	5	7

Szarym tłem wyróżniono ciąg posortowany, natomiast kolorem niebieskim zaznaczono wstawiane wyrazy.

Przejdźmy więc do realizacji tego algorytmu w tablicy jednowymiarowej dla n-wyrazowego ciągu.

Przykład 2.34.

Przyjrzyjmy się kolejnym krokom algorytmu sortowania przez wstawianie realizowanym w tablicy jednowymiarowej dla analizowanego wcześniej ciągu (7, 3, 0, 1, 5).

	7	3	0	1	5
Krok 1.	**3**	7	0	1	5
Krok 2.	**0**	3	7	1	5
Krok 3.	0	**1**	3	7	5
Krok 4.	0	1	3	**5**	7

Niebieskim kolorem zaznaczono wyrazy ciągu, które zostały w danym kroku wstawione do posortowanego już podciągu. Szarym tłem wyróżniono wyrazy ciągu, które zostały posortowane.

Załóżmy, że daną mamy tablicę $T[0\ldots n-1]$, w której zapisany jest ciąg do posortowania. Z powyższej analizy wynika, że sortowanie wykonywane jest w $n-1$ krokach. W każdym z tych etapów zwiększa się fragment ciągu już posortowanego. Wybierane są kolejne wyrazy o numerach z przedziału $[1, n-1]$ i przestawiane we właściwe miejsce w tym fragmencie ciągu, który jest już posortowany. Zaczynamy od elementu $T[1]$, który zapisujemy do zmiennej pomocniczej. Następnie, jeśli $T[1]$ jest mniejszy od elementu $T[0]$, wyraz ciągu $T[0]$ przesuwamy w prawo na miejsce o indeksie 1, natomiast wstawiany element $T[1]$, zapisany w zmiennej pomocniczej, umieszczamy w miejscu $T[0]$. W kolejnym kroku zapamiętujemy element $T[2]$ i wstawiamy go do posortowanego fragmentu ciągu, czyli $T[0\ldots1]$. Jeśli wstawiany element $T[2]$ jest mniejszy od kolejnych wartości posortowanego ciągu $T[1]$ i $T[0]$, dany wyraz ciągu przesuwany jest w prawo. Natomiast element $T[2]$ wstawiany jest w wolne miejsce, czyli pomiędzy element mniejszy i większy od niego lub na początek ciągu. Jeżeli element $T[2]$ nie jest mniejszy od $T[1]$, to pozostawiamy go na dotychczasowym miejscu, które staje się ostatnią pozycją uporządkowanej części ciągu. Kolejne kroki algorytmu powtarzane są tak długo, aż dojdziemy do elementu ostatniego $T[n-1]$, który będziemy wstawiać do ciągu $T[0\ldots n-2]$.

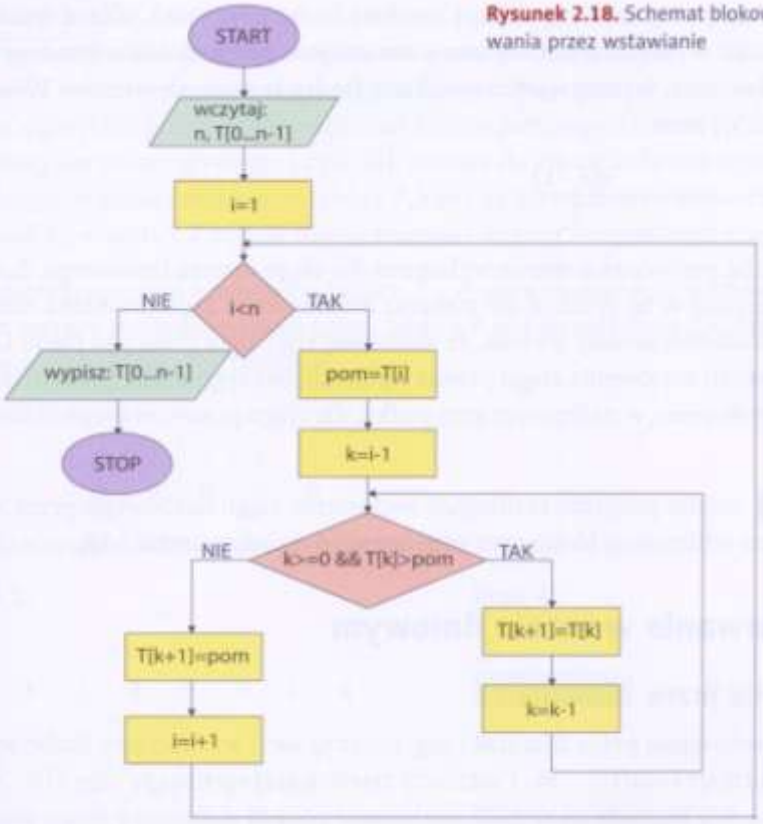
Skonstruujmy **algorytm realizujący sortowanie przez wstawianie** w postaci listy kroków, schematu blokowego (patrz rysunek 2.18) oraz programów w językach C++ i Pascal.

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0\ldots n-1]$.

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0\ldots n-1]$.



Rysunek 2.18. Schemat blokowy algorytmu sortowania przez wstawianie

Lista kroków:

Krok 0. Wczytaj n, $T[0...n-1]$.

Krok 1. Dla kolejnych wartości i: $1, 2, \ldots, n-1$, wykonuj kroki 2. – 3., a następnie przejdź do kroku 4.

Krok 2. Przypisz $pom = T[i]$ (pobranie elementu do wstawienia).

Krok 3. Porównuj element pom z kolejnymi wyrazami uporządkowanego podciągu $T[k]$, dla $k = i-1, i-2, \ldots, 0$, przesuń sprawdzone elementy, zwiększając im indeks o 1 w celu zrobienia miejsca dla pom, i umieść element pom przed pierwszym elementem $T[k]$, który będzie spełniał warunek $T[k] \leq pom$. Przyimek „przed" należy tu rozumieć jako „na pozycji o numerze o jeden wyższym".

(Wstawienie elementu $pom = T[i]$ do posortowanego podciągu $T[0\ldots i-1]$ w taki sposób, aby po dodaniu tego elementu podciąg nadal był uporządkowany).

Krok 4. Wypisz elementy tablicy $T[0\ldots n-1]$. Zakończ algorytm.

Wykonajmy analizę złożoności czasowej algorytmu. Operacją dominującą jest porównanie. W tym przypadku dane wejściowe mają wpływ na złożoność algorytmu. Wyznaczmy

złożoność pesymistyczną, czyli największą możliwą liczbę porównań, jaka w tym algorytmie może zostać wykonana. Taka sytuacja ma miejsce, gdy sortowany jest ciąg uporządkowany odwrotnie, wymagający największej liczby zamian elementów. Wówczas liczba porównań wynosi:

$$1+2+\ldots+n-1=\frac{n(n-1)}{2}. \tag{2.25}$$

Najmniejsza liczba porównań zostanie wykonana dla ciągu uporządkowanego, dokładnie $n-1$ razy. Ponadto w tej sytuacji nie zostanie zrealizowana żadna zamiana elementów. Z przeprowadzonej analizy wynika, że złożoność tego algorytmu jest rzędu $O(n^2)$. Jednak w przypadku sortowania ciągu prawie uporządkowanego liczba porównań maleje, przez co uzyskujemy, w najlepszym przypadku, dla ciągu posortowanego złożoność liniową $O(n)$.

Zadanie 2.48. Napisz program realizujący sortowanie ciągu liczbowego przez wstawianie zgodny ze schematem blokowym przedstawionym na rysunku 2.18.

2.9.2. Sortowanie w czasie liniowym

Porządkowanie przez zliczanie

W algorytmie sortowania przez zliczanie (ang. *counting sort*) wyznaczamy liczbę wystąpień każdej z wartości $i = 0, 1, \ldots, m-1$ w tablicy zawierającej sortowany ciąg $T[0\ldots n-1]$. Uzyskana informacja pozwala na szybkie znalezienie pozycji elementów ciągu w sortowanej tablicy. W omawianym algorytmie sortowania nie porównujemy elementów, tylko obliczamy liczbę ich wystąpień.

Przedstawioną metodę możemy zastosować, gdy ograniczona jest liczba wartości wyrazów porządkowanego ciągu, mianowicie tylko takich, które mogą być indeksami tablic. Algorytm sortowania przez zliczanie wymaga zastosowania dodatkowej tablicy liczników $P[0\ldots m-1]$, w której zapisywana jest liczba wystąpień danej wartości i (dla $i = 0, 1, \ldots, m-1$) w sortowanym ciągu. Na przykład wartość elementu $P[k]$ to liczba wystąpień wartości k w sortowanym ciągu.

Realizację algorytmu zaczynamy od wyzerowania tablicy liczników $P[0\ldots m-1]$. Następnie przeglądamy tablicę $T[0\ldots n-1]$ zawierającą elementy sortowanego ciągu i odpowiednio zwiększamy te liczniki $P[k]$, które spełniają warunek $k = T[i]$. Ostatnim krokiem jest zapisanie posortowanego ciągu w tablicy $T[0\ldots n-1]$, do której wpisujemy każdy indeks k tablicy $P[0\ldots m-1]$, dla $k = 0, 1, \ldots, m-1$, tyle razy, ile wynosi wartość elementu $P[k]$.

Sortowanie przez zliczanie, pomimo zastosowania tablicy pomocniczej, jest metodą porządkującą ciąg w miejscu. Wynika to stąd, że elementów sortowanego ciągu nie przepisujemy do innej tablicy. Algorytm ten jest również stabilny.

Przykład 2.35.

Przeanalizujmy kolejne etapy sortowania przez zliczanie ciągu $T = (4, 3, 3, 4, 0, 2, 1, 4)$. Wartości wyrazów sortowanego ciągu to liczby całkowite od 0 do 4 włącznie. Do wykonania algorytmu potrzebne są więc dwie tablice jednowymiarowe: $T[0\ldots7]$, zawierająca elementy sortowanego ciągu, i $P[0\ldots4]$, służąca do zliczania liczby wystąpień wyrazów tego ciągu. Wartości elementów tablicy P, które są licznikami, zwiększać będziemy wtedy, gdy indeks w tablicy P będzie równy wartości wyrazu sortowanego ciągu.

Przed rozpoczęciem sortowania tablice T i P mają następujące wartości: tablica T zawiera wyrazy porządkowanego ciągu, tablica P jest wyzerowana. Dodatkowo uwidocznione zostały indeksy w tablicy P.

P:

k	0	1	2	3	4
$P[k]$	0	0	0	0	0

Realizacja algorytmu przebiega następująco:

Krok 1.

T:

4	3	3	4	0	2	1	4

P:

k	0	1	2	3	4
$P[k]$	0	0	0	0	**1**

Krok 2.

T:

4	**3**	3	4	0	2	1	4

P:

k	0	1	2	3	4
$P[k]$	0	0	0	**1**	1

Krok 3.

T:

4	3	**3**	4	0	2	1	4

P:

k	0	1	2	3	4
$P[k]$	0	0	0	**2**	1

Krok 4.

T:

4	3	3	**4**	0	2	1	4

P:

k	0	1	2	3	4
$P[k]$	0	0	0	2	**2**

Krok 5.

T:

4	3	3	4	**0**	2	1	4

P:

k	0	1	2	3	4
$P[k]$	**1**	0	0	2	2

Krok 6.

T:

4	3	3	4	0	**2**	1	4

P:

k	0	1	2	3	4
$P[k]$	1	0	**1**	2	2

Krok 7.

T:

4	3	3	4	0	2	**1**	4

P:

k	0	1	2	3	4
P[*k*]	1	**1**	1	2	2

Krok 8.

T:

4	3	3	4	0	2	1	**4**

P:

k	0	1	2	3	4
P[*k*]	1	1	1	2	**3**

Szarym tłem wyróżniono wyrazy już policzone, natomiast kolorem niebieskim zaznaczono aktualnie pobierane wartości i zwiększany licznik.

Po zliczeniu wszystkich elementów tablicy T (czyli porządkowanego ciągu) w tablicy P przepisujemy do tablicy T wyrazy posortowanego ciągu. Do tablicy wynikowej T wpisujemy kolejne indeksy tablicy P, każdy numer tyle razy, ile wynosi wartość tego wyrazu. Na przykład $P[4] = 3$, więc do tablicy T wpiszemy trzy razy wartość 4.

Po posortowaniu tablica T będzie miała następującą postać: $T = (0, 1, 2, 3, 3, 4, 4, 4)$.

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

Liczba naturalna: $m > 0$ (wartość, od której wszystkie elementy tablicy T są mniejsze).

n-elementowa tablica jednowymiarowa zawierająca liczby całkowite: $T[0...n-1]$, gdzie $0 \le T[i] < m$, dla $i = 0, 1, ..., n-1$ (ciąg do posortowania).

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby całkowite: $T[0...n-1]$.

Funkcja w języku C++ (prog2_23.cpp):

```
void sortuj (int T[], int n, int m)
{
 int P[MAX]={0};
 for (int i=0;i<n;i++) P[T[i]]++;
 int k=0;
 for (int i=0;i<m;i++)
  for (int j=P[i];j>=1;j--)
  {
   T[k]=i;
   k++;
  }
}
```

Procedura w języku Pascal (prog2_23.pas):

```
procedure sortuj (var T: tablica; n, m: integer);
var P: tablica;
    i, j, k: integer;
begin
 for i:=0 to m-1 do P[i]:=0;
 for i:=0 to n-1 do P[T[i]]:=P[T[i]]+1;
 k:=0;
 for i:=0 to m-1 do
  for j:=P[i] downto 1 do
  begin
   T[k]:=i;
   k:=k+1
  end
end;
```

W metodzie sortowania przez zliczanie z tablicy pobierane są kolejno wszystkie wyrazy ciągu, co wymaga wykonania n operacji przypisania i porównania. Wyzerowanie tablicy liczników $P[0\ldots m-1]$, wykonywane na początku, to m powtórzeń. Natomiast wpisanie posortowanego ciągu do tablicy $T[0\ldots n-1]$ stanowi n wykonanych przypisań. Uzyskujemy razem $m+n+n = m+2n$ operacji przypisania i porównania, co daje złożoność $O(m+n)$. Dane wejściowe nie mają w tym przypadku wpływu na liczbę wykonywanych działań. Przy założeniu, że m jest co najwyżej równe n, otrzymujemy złożoność liniową $O(n)$.

Porządkowanie kubełkowe

Sortowanie kubełkowe ciągu $T[0\ldots n-1]$ wymaga wykorzystania dodatkowej tablicy $P[0\ldots m-1]$, której elementy nazywamy kubełkami. Do kubełka $P[i]$ wrzucane są wszystkie elementy sortowanej tablicy $T[0\ldots n-1]$, które mają wartość i. Po przejrzeniu całego ciągu do tablicy $T[0\ldots n-1]$ wpisywane są wartości zawarte w kubełkach. Sortowanie wykonywane jest więc przez gromadzenie wyrazów porządkowanego ciągu w odpowiednich kubełkach reprezentujących ich wartości, a nie z wykorzystaniem porównań.

Omawiany algorytm realizowany jest z wykorzystaniem programowania dynamicznego. Kubełki $P[0\ldots m-1]$ reprezentowane są jako listy jednokierunkowe (patrz punkt 3.7.3, „Lista").

Przedstawiona metoda nie wykonuje sortowania w miejscu, jest jednak metodą stabilną.

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

Liczba naturalna: $m > 0$ (wartość, od której wszystkie elementy tablicy T są mniejsze).

n-elementowa tablica jednowymiarowa zawierająca liczby całkowite: $T[0...n-1]$, gdzie $0 \le T[i] < m$, dla $i = 0, 1, ..., n-1$ (ciąg do posortowania).

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby całkowite: $T[0...n-1]$.

Lista kroków:

Krok 0. Wczytaj n, $T[0...n-1]$.

Krok 1. Dla kolejnych wartości i: $0, 1, ..., m-1$, wykonuj krok 2., a następnie przejdź do kroku 3.

Krok 2. Utwórz listę pustą $P[i]$.

Krok 3. Dla kolejnych wartości i: $n-1, n-2, ..., 0$, wykonuj krok 4., a następnie przejdź do kroku 5.

Krok 4. Wstaw $T[i]$ na początek listy $P[T[i]]$.

Krok 5. Wykonaj scalenie list $P[0...m-2]$ w jedną listę P, łącząc kolejno $P[i+1]$ z $P[i]$.

Krok 6. Przepisz kolejne elementy listy P do tablicy $T[0...n-1]$.

Krok 7. Wypisz elementy tablicy $T[0...n-1]$. Zakończ algorytm.

Przeprowadźmy analizę złożoności czasowej algorytmu. Wiemy, że m to liczba wartości wyrazów sortowanego ciągu. Operacją dominującą jest tutaj porównanie wykonywane w pętli oraz operacja przypisania realizowana podczas sortowania. W krokach pierwszym i drugim wykonywanych jest m operacji, w trzecim i czwartym — n, w piątym — $m-1$, a w szóstym — n. Liczba wykonywanych działań wynosi więc $m+n+(m-1)+n = 2n+2m-1$. Uzyskujemy złożoność rzędu $O(n+m)$. Przy założeniu, że m ma wartość co najwyżej n, możemy stwierdzić, że mamy złożoność liniową $O(n)$.

Zadanie 2.49. Podaj przykładowe metody sortowania zaliczane do:

a) algorytmów stabilnych,

b) algorytmów niestabilnych,

c) algorytmów sortujących w miejscu,

d) algorytmów, które sortują nie w miejscu,

e) algorytmów, których złożoność jest zależna od danych wejściowych,

f) algorytmów, których złożoność nie jest zależna od danych wejściowych.

Odpowiedzi uzasadnij.

Zadanie 2.50. Podaj specyfikację i skonstruuj algorytm w postaci programu wpisujący do tablicy jednowymiarowej n liczb całkowitych wygenerowanych losowo z przedziału $[0, 28]$ oraz porządkujący je nierosnąco z wykorzystaniem metody:

a) stabilnej i sortującej w miejscu,

b) stabilnej i sortującej nie w miejscu.

Wybrane zadania maturalne

Na załączonej do podręcznika płycie CD zamieszczono treść wybranego zadania maturalnego. Zaproponowane zadanie wymaga od ucznia znajomości zagadnień związanych z sortowaniem ciągów liczbowych.

- *matura2.16.pdf* (*Egzamin styczeń 2006 r. Arkusz I, zadanie 2.*).

2.10. Zastosowanie metody „dziel i zwyciężaj"

2.10.1. Jednoczesne znajdowanie minimalnego i maksymalnego elementu

Dany n-wyrazowy ciąg, gdzie n jest liczbą naturalną większą od 0, dzielimy na dwa podciągi: *MIN* i *MAX*. Pierwszy, *MIN*, zawierać będzie wyrazy, wśród których jest minimum, a drugi, *MAX* — wyrazy obejmujące maksimum. Podziału dokonujemy, porównując ze sobą parami kolejne wyrazy ciągu. Liczby mniejsze dołączamy do podciągu *MIN*, a większe lub równe do *MAX*. Korzystając z optymalnego algorytmu na znajdowanie minimum lub maksimum w ciągu liczbowym (patrz podrozdział 2.6, „Znajdowanie minimalnego lub maksymalnego elementu"), wyznaczamy w pierwszym podciągu minimum, a w drugim maksimum. W przypadku gdy sprawdzany ciąg ma parzystą liczbę wyrazów, dzieli się on dokładnie na dwie połowy. W sytuacji gdy liczba wyrazów jest nieparzysta, ostatni wyraz ciągu przypisujemy do obydwu podciągów.

Wskazówka

Algorytm ten można zastosować przy wyznaczaniu rozpiętości zbioru, która jest równa różnicy minimalnego i maksymalnego elementu w tym zbiorze.

Przykład 2.36.

Przeanalizujmy realizację **algorytmu iteracyjnego** dla ciągu liczbowego (2, 3, 4, 3, 6, 7, 1, 0, 3, 3, 6, 9, 2).

Rozpoczynamy od podziału ciągu liczb na dwa podciągi: *MIN* — wyrazy zawierające minimum, i *MAX* — wartości obejmujące maksimum.

Podciąg *MIN*	2	3	6	0	3	6	2
Podział ciągu liczb	$2 \leq 3$	$4 > 3$	$6 \leq 7$	$1 > 0$	$3 \leq 3$	$6 \leq 9$	**2**
Podciąg *MAX*	3	4	7	1	3	9	2

Po podziale wiemy, że minimum znajduje się w podciągu *MIN* = (2, 3, 6, 0, 3, 6, 2), natomiast maksimum w *MAX* = (3, 4, 7, 1, 3, 9, 2).

Korzystając z klasycznych algorytmów wyznaczających najmniejszy lub największy element w ciągu, wyznaczamy minimum dla podciągu *MIN* oraz maksimum dla podciągu *MAX*. Otrzymujemy następujące wyniki: 0 — minimum, 9 — maksimum.

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.

Wynik: Najmniejszy element *minimum* i największy element *maksimum* w tablicy $T[0...n-1]$.

Funkcja iteracyjna w języku C++ (prog2_24.cpp):

```
void oblicz (double T[], int n, double &minimum, double &maksimum)
{
 int dl, i;
 if (n%2) dl=n-2; else dl=n-1;
 if (T[0]<=T[1])
 {
  minimum=T[0];
  maksimum=T[1];
 }
 else
 {
  minimum=T[1];
  maksimum=T[0];
 }
 i=2;
 while (i<dl)
 {
  if (T[i]<=T[i+1])
  {
   if (T[i]<minimum) minimum=T[i];
   if (T[i+1]>maksimum) maksimum=T[i+1];
  }
  else
  {
   if (T[i+1]<minimum) minimum=T[i+1];
```

```
    if (T[i]>maksimum) maksimum=T[i];
   }
   i+=2;
  }
  if (n%2)
  {
   if (T[n-1]<minimum) minimum=T[n-1];
   if (T[n-1]>maksimum) maksimum=T[n-1];
  }
 }
```

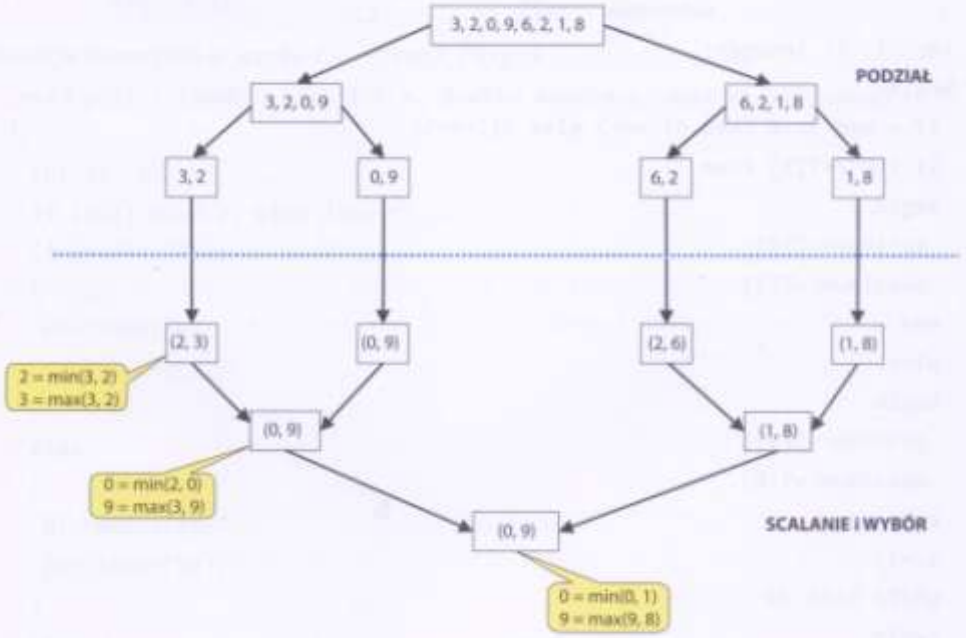
Procedura iteracyjna w języku Pascal (prog2_24.pas):

```
procedure oblicz (var T: tablica; n: integer; var minimum,
                  maksimum: real);
var dl, i: integer;
begin
 if n mod 2<>0 then dl:=n-2 else dl:=n-1;
 if T[0]<=T[1] then
 begin
  minimum:=T[0];
  maksimum:=T[1]
 end
 else
 begin
  minimum:=T[1];
  maksimum:=T[0]
 end;
 i:=2;
 while i<dl do
 begin
  if T[i]<=T[i+1] then
  begin
   if T[i]<minimum then minimum:=T[i];
   if T[i+1]>maksimum then maksimum:=T[i+1]
  end
  else
  begin
   if T[i+1]<minimum then minimum:=T[i+1];
   if T[i]>maksimum then maksimum:=T[i]
  end;
```

```
    i:=i+2;
  end;
  if n mod 2<>0 then
  begin
   if T[n-1]<minimum then minimum:=T[n-1];
   if T[n-1]>maksimum then maksimum:=T[n-1]
  end
 end;
```

Przykład 2.37.

Prześledźmy realizację **algorytmu rekurencyjnego** dla ciągu liczbowego (3, 2, 0, 9, 6, 2, 1, 8) przedstawioną na rysunku 2.19.



Rysunek 2.19. Przebieg rekurencyjnego algorytmu jednoczesnego znajdowania minimum i maksimum dla ciągu liczbowego (3, 2, 0, 9, 6, 2, 1, 8)

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.

Wynik: Minimalny element *minimum* i maksymalny element *maksimum* w tablicy $T[0...n-1]$.

Lista kroków:

Krok 0. Wczytaj n, $T[0...n-1]$.

Algorytm rekurencyjny (*T, lewy, prawy, minimum, maksimum*):

Krok 1. Jeśli *lewy* = *prawy* (ciąg jest 1-wyrazowy), przypisz *maksimum* = $T[lewy]$, *minimum* = $T[lewy]$ oraz zakończ algorytm, w przeciwnym wypadku przejdź do kroku 2.

Krok 2. Jeśli *lewy* = *prawy*–1 (ciąg jest 2-wyrazowy), przypisz *minimum* = wartość mniejszego elementu ciągu, *maksimum* = wartość większego elementu ciągu, a następnie zakończ algorytm, w przeciwnym wypadku przejdź do kroku 3.

Krok 3. Podziel ciąg liczb T na dwa podciągi: $T1$ i $T2$.

Krok 4. Uruchom ten algorytm z parametrami ($T1$, *lewy*, (*lewy*+*prawy*)/2, *minimum*1, *maksimum*1).

Krok 5. Uruchom ten algorytm z parametrami ($T2$, (*lewy*+*prawy*)/2+1, *prawy*, *minimum*2, *maksimum*2).

Krok 6. Przypisz *maksimum* = wartość większej z liczb *maksimum*1 i *maksimum*2, *minimum* = wartość mniejszej z liczb *minimum*1 i *minimum*2.

Zastosowanie w tym algorytmie zasady „dziel i zwyciężaj" jest widoczne przy podziale ciągu na dwa podciągi. **Zadanie realizowane dla danych rozmiaru *n* dzielone jest na dwa podzadania rozmiaru $n/2$.**

Przyjrzyjmy się złożoności rozwiązywanego problemu. Gdybyśmy zdecydowali się na wykonanie zadania przez skorzystanie z klasycznych metod wyznaczających minimum lub maksimum w n-wyrazowym ciągu liczbowym (omówionych w podrozdziale 2.6, „Znajdowanie minimalnego lub maksymalnego elementu"), to najpierw uruchomilibyśmy jeden z tych algorytmów, a potem drugi. Operacją dominującą w tych metodach jest porównanie. Każdy z tych algorytmów wykonuje $n-1$ działań dominujących. Liczba porównań w przypadku wyboru takiego rozwiązania, zwanego naiwnym, jest więc równa $2(n-1)$. Uzyskalibyśmy zatem złożoność liniową $O(n)$.

Przeanalizujmy teraz złożoność metody wykorzystującej technikę „dziel i zwyciężaj" omówionej w przykładzie 2.36. Przyjmijmy, że $n = 2^k$ (gdzie k jest liczbą naturalną). Spowoduje to, że w wyniku podziału uzyskujemy dwa równe podciągi. W pierwszym kroku podczas podziału wykonujemy $n/2$ porównań. W następnym etapie, podczas znajdowania minimum i maksimum w podciągach, liczba porównań dla każdego z wykorzystanych tutaj algorytmów wynosi $n/2-1$. Całkowita liczba tych operacji, przy założeniu, że n jest parzyste, wynosi więc $n/2+(n/2-1)+(n/2-1)=3n/2-2$. Liczba porównań wykonywanych w przypadku zastosowania techniki „dziel i zwyciężaj" jest znacznie mniejsza niż w sytuacji zastosowania algorytmów znajdujących minimum i maksimum dla całego ciągu, chociaż w tym przypadku również uzyskujemy złożoność liniową $O(n)$.

Zadanie 2.51. Przedstaw przebieg algorytmu rekurencyjnego realizującego jednoczesne znajdowanie minimum i maksimum dla podanych ciągów liczbowych (podobnie jak na rysunku 2.19):

a) (10, 9, 8, 7, 6, 5, 4, 3, 2, 1),

b) (10, 1, 9, 2, 8, 3, 7, 4, 6),

c) (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0).

Zadanie 2.52. Napisz program realizujący rekurencyjnie jednoczesne znajdowanie minimalnego i maksymalnego wyrazu w ciągu liczbowym zgodny z podaną listą kroków.

2.10.2. Sortowanie przez scalanie

Metoda sortowania przez scalanie (ang. *merge sort*) zaliczana jest do algorytmów wykorzystujących porównania. Jednocześnie jednak jest to metoda wykorzystująca omawianą w tym rozdziale technikę „dziel i zwyciężaj".

W metodzie sortowania przez scalanie wyróżniamy dwa etapy: **podział** i **scalanie**. Pierwsza faza wykonywana jest rekurencyjnie i polega na podzieleniu ciągu na podciągi zawierające jedną wartość. Druga faza realizowana jest podczas łączenia podciągów i polega na scalaniu ich, z jednoczesnym sortowaniem. Wynika stąd, że głównym celem jest tutaj scalenie dwóch uporządkowanych ciągów w jeden posortowany. Łączenie będzie wykonywane bezpośrednio w tablicy, w której zapisany jest sortowany ciąg. Jednak będzie nam potrzebna dodatkowa tablica, do której skopiujemy jeden ze scalanych podciągów.

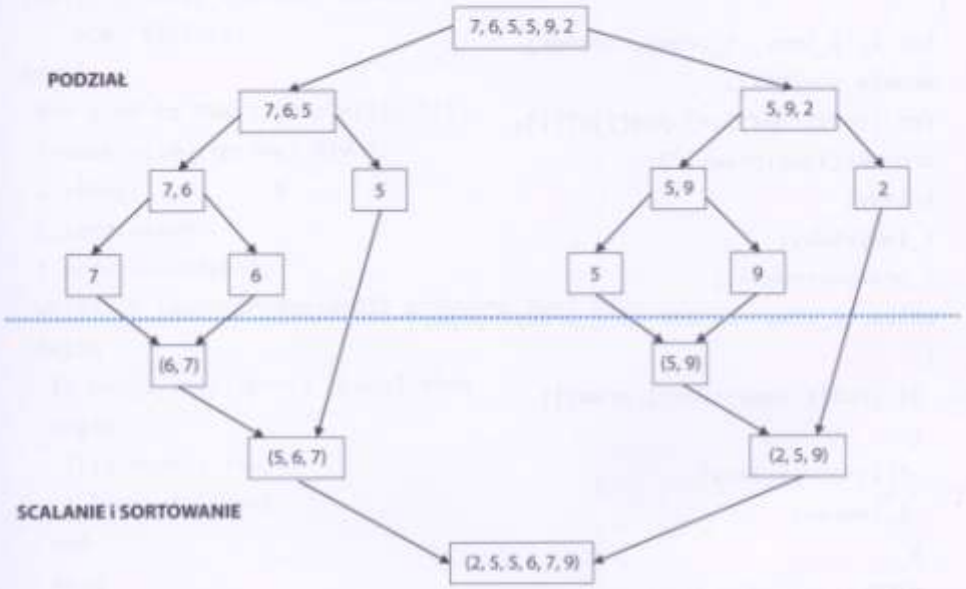
Załóżmy, że w tablicy mamy dwa uporządkowane podciągi, które chcemy scalić. Kopiujemy pierwszy z lewej strony tablicy podciąg (mający niższe indeksy) do dodatkowej tablicy. Następnie porównujemy parami kolejne wyrazy podciągów i wstawiamy do tablicy te, które są mniejsze. Przyjmijmy na przykład, że mamy tablicę $A[0\ldots4]$, a w niej dwa uporządkowane podciągi: $A[0\ldots2] = (2, 7, 8)$ i $A[3\ldots4] = (1, 5)$, które chcemy scalić. Pierwszy z lewej podciąg $A[0\ldots2]$ przepisujemy do tablicy dodatkowej $B[0\ldots2]$. Następnie przechodzimy do scalenia tych podciągów. Wykonujemy więc następujące działania:

- Porównujemy pierwszą parę elementów $B[0] = 2$ i $A[3] = 1$. Element $A[3]$ jest mniejszy, więc wstawiamy go do tablicy A w miejsce o numerze 0, czyli $A[0] = A[3]$. Pierwszym posortowanym wyrazem ciągu jest $A[0] = 1$.
- Porównujemy kolejną parę elementów: $B[0] = 2$ i $A[4] = 5$. $B[0]$ ma mniejszą wartość, więc wstawiamy ten element do tablicy A na miejsce $A[1]$. Drugim posortowanym wyrazem ciągu jest więc $A[1] = 2$.
- Następna para elementów to $B[1] = 7$ i $A[4] = 5$. Wartość elementu $A[4]$ jest mniejsza, więc do tablicy na miejscu $A[2]$ wstawiamy element $A[4]$. Uzyskujemy kolejny wyraz ciągu: $A[2] = 5$.
- Wyczerpaliśmy już wszystkie wyrazy podciągu $A[3...4]$, natomiast podciąg $B[0...2]$ nie został jeszcze przejrzany w całości. W takiej sytuacji przepisujemy do tablicy A pozostałe wyrazy podciągu $B[0\ldots2]$, czyli $A[3] = B[1]$ i $A[4] = B[2]$. Następnymi wyrazami uporządkowanego ciągu są więc $A[3] = 7$ i $A[4] = 8$.

Na skutek naszych działań w miejscu dwóch posortowanych podciągów $A[0\ldots2] = (2, 7, 8)$ i $A[3\ldots4] = (1, 5)$ pojawił się jeden uporządkowany ciąg $A[0\ldots4] = (1, 2, 5, 7, 8)$, będący efektem scalenia tych podciągów.

Przykład 2.38.

Przeanalizujmy realizację algorytmu sortowania przez scalanie na przykładzie liczbowym: (7, 6, 5, 5, 9, 2). Naszym celem jest uporządkowanie tego ciągu liczbowego w kolejności od najmniejszego do największego. Przebieg algorytmu został przedstawiony na rysunku 2.20.



Rysunek 2.20. Przebieg rekurencyjnego algorytmu sortowania przez scalanie dla ciągu (7, 6, 5, 5, 9, 2)

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0\ldots n-1]$.

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0\ldots n-1]$.

Lista kroków:

Krok 0. Wczytaj n, $T[0\ldots n-1]$.

Algorytm rekurencyjny (*T*, *n*, *lewy*, *prawy*):

Krok 1. Jeśli $lewy < prawy$, przypisz $srodek = \frac{lewy + prawy}{2}$ i przejdź do kroku 2., w przeciwnym wypadku zakończ algorytm.

Krok 2. Uruchom ten algorytm z parametrami (*T*, *lewy*, *srodek*).

Krok 3. Uruchom ten algorytm z parametrami (*T*, *srodek*+1, *prawy*).

Krok 4. Scal ciągi *T*[*lewy*, *srodek*] i *T*[*srodek*+1, *prawy*] w jeden ciąg *T*[*lewy*, *prawy*].

Istotną częścią przedstawionej metody jest scalanie uporządkowanych ciągów. Skonstruujmy algorytm realizujący scalanie w postaci programów w językach C++ i Pascal.

Funkcja w języku C++ (prog2_25.cpp):

```
void scalaj (double T[], int lewy, int prawy)
{
 int i, i_lewy, i_prawy, srodek;
 double pom[MAX];
 for (i=0;i<MAX;i++) pom[i]=T[i];
 srodek=(lewy+prawy)/2;
 i=lewy;
 i_lewy=lewy;
 i_prawy=srodek+1;
 while (i_lewy<=srodek && i_prawy<=prawy)
 {
  if (pom[i_lewy]<pom[i_prawy])
  {
   T[i]=pom[i_lewy];
   i_lewy++;
  }
  else
  {
   T[i]=pom[i_prawy];
   i_prawy++;
  }
  i++;
 }
 if (i_lewy>srodek)
  while (i_prawy<=prawy)
  {
   T[i]=pom[i_prawy];
   i_prawy++;
   i++;
  }
 else
  while (i_lewy<=srodek)
```

```
    {
     T[i]=pom[i_lewy];
     i_lewy++;
     i++;
    }
  }
```

Procedura w języku Pascal (prog2_25.pas):

```
procedure scalaj (var T: tablica; lewy, prawy: integer);
var i, i_lewy, i_prawy, srodek: integer;
    pom: tablica;
begin
 for i:=0 to MAX-1 do pom[i]:=T[i];
 srodek:=(lewy+prawy) div 2;
 i:=lewy;
 i_lewy:=lewy;
 i_prawy:=srodek+1;
 while (i_lewy<=srodek)and(i_prawy<=prawy) do
 begin
  if pom[i_lewy]<pom[i_prawy] then
  begin
   T[i]:=pom[i_lewy];
   i_lewy:=i_lewy+1
  end
  else
  begin
   T[i]:=pom[i_prawy];
   i_prawy:=i_prawy+1
  end;
  i:=i+1
 end;
 if i_lewy>srodek then
  while i_prawy<=prawy do
  begin
   T[i]:=pom[i_prawy];
   i_prawy:=i_prawy+1;
   i:=i+1
  end
 else
  while i_lewy<=srodek do
```

```
    begin
     T[i]:=pom[i_lewy];
     i_lewy:=i_lewy+1;
     i:=i+1
    end
  end;
```

Przedstawiona procedura-funkcja *scalaj(T, lewy, prawy)* wykonuje połączenie dwóch podciągów znajdujących się w tablicy *T*, mianowicie *T[lewy, srodek]* i *T[srodek+1, prawy]*. Możemy więc przystąpić do zdefiniowania algorytmu sortującego elementy tablicy z wykorzystaniem scalania.

Funkcja w języku C++ (prog2_25.cpp):

```
void sortuj (double T[], int lewy, int prawy)
{
 int srodek=(lewy+prawy)/2;
 if (lewy<srodek) sortuj(T,lewy,srodek);
 if (srodek+1<prawy) sortuj(T,srodek+1,prawy);
 scalaj(T,lewy,prawy);
}
```

Procedura w języku Pascal (prog2_25.pas):

```
procedure sortuj (var T: tablica; lewy, prawy: integer);
var srodek: integer;
begin
 srodek:=(lewy+prawy) div 2;
 if lewy<srodek then sortuj(T,lewy,srodek);
 if srodek+1<prawy then sortuj(T,srodek+1,prawy);
 scalaj(T,lewy,prawy)
end;
```

Wykonanie sortowania polega więc w tym algorytmie na podziale sortowanego ciągu na dwa mniejsze, których rozmiary są równe lub prawie równe. Czynność podziału powtarzana jest dopóty, dopóki nie otrzymamy pojedynczych wyrazów, następnie przechodzimy do fazy scalania, podczas której realizowane jest sortowanie.

Przedstawiony algorytm zrealizowany jest rekurencyjnie. Zakładamy, że liczba wyrazów sortowanego ciągu *n* jest postaci 2^k, gdzie *k* jest liczbą naturalną. Umożliwi to dzielenie ciągu w każdym kolejnym kroku na dwie równe części. Po każdym podziale liczba wyrazów podciągu będzie maleć dwukrotnie. Po pierwszym wywołaniu rekurencyjnym wyniesie 2^{k-1}, po drugim 2^{k-2}, natomiast na *k*-tym poziomie pozostanie tylko 1 wyraz. Po dojściu do pojedynczych wyrazów na *k*-tym poziomie wywołań rekurencyjnych mamy już tylko ciągi 1-wyrazowe, które są posortowane. Zaczynamy więc scalanie ciągów upo-

rządkowanych. Najpierw wykonamy $^{n}/_{2}$ porównań dla 1-wyrazowych ciągów, na kolejnym k–1 poziomie $^{3n}/_{4}$ porównań podczas łączenia ciągów 2-wyrazowych w 4-wyrazowe. W ostatnim kroku scalamy dwa ciągi w jeden n-wyrazowy, co wymaga n–1 porównań. Możemy więc wyznaczyć ogólną liczbę porównań dla tego algorytmu:

$$1\cdot\frac{n}{2}+3\cdot\frac{n}{4}+7\cdot\frac{n}{8}+\ldots+(n-1)\cdot\frac{n}{n}. \tag{2.26}$$

Uwzględniając fakt, że $n = 2^k$, otrzymujemy liczbę porównań wynoszącą: $n\cdot\log_2 n-n+1$. Złożoność czasowa tej metody jest więc rzędu $O(n \log n)$.

Zadanie 2.53. Przedstaw przebieg algorytmu rekurencyjnego realizującego sortowanie przez scalanie dla podanych ciągów liczbowych (podobnie jak na rysunku 2.20):

a) (9, 9, 8, 7, 5, 5, 4, 3, 2, 1),

b) (11, 2, 6, 3, 7, 3, 7, 5, 8),

c) (2, 12, 3, 4, 5, 4, 7, 8, 9, 9, 0).

Zadanie 2.54. Zmodyfikuj przedstawiony algorytm realizujący sortowanie przez scalanie tak, aby porządkował wyrazy ciągu nierosnąco.

2.10.3. Sortowanie szybkie

Metoda sortowania szybkiego (ang. *quick sort*) jest oparta na następującej własności: jeśli w tablicy $T[0\ldots n-1]$ istnieje element o indeksie k taki, że wszystkie elementy o mniejszych numerach mają wartość mniejszą od $T[k]$ oraz wszystkie elementy o większych indeksach mają wartość większą od $T[k]$, to, aby uzyskać posortowany ciąg, wystarczy osobno posortować elementy tablicy $T[0\ldots k-1]$ i $T[k+1\ldots n-1]$.

W każdym kolejnym kroku omawianej metody powtarzane są te same działania, zmienia się tylko fragment ciągu, na którym wykonujemy określone operacje. Indeks pierwszego wyrazu w podciągu oznaczymy jako *lewy*, ostatniego — *prawy*. Realizację każdego kroku algorytmu należy więc rozpocząć od wybrania wyrazu środkowego, którego wartość wyznaczamy następująco: *srodek* = $T[(lewy+prawy)/2]$. Znajdowanie wyrazów ciągu do zamiany rozpoczynamy od wyrazów skrajnych: *lewy* i *prawy*, a dalej przesuwamy się w stronę wyrazu środkowego *srodek*. Szukamy z lewej strony elementu $T[i] \geq srodek$, a z prawej elementu $T[j] \leq srodek$. Po znalezieniu pary spełniającej podane warunki wykonujemy zamianę elementów $T[i]$ z $T[j]$. Zauważmy, że w obu warunkach zastosowano nierówności nieostre („≥" oraz „≤" zamiast ostrych „>" oraz „<"), dzięki czemu również sam element *srodek* może zmieniać miejsce. Czynności te powtarzamy tak długo, aż indeksy i i j spotkają się, dochodząc z obu stron do elementu *srodek*. Wyraz ten może w tym momencie nie znajdować się już w połowie długości ciągu. Istotne jest natomiast, że znalazł się na właściwej (docelowej) pozycji w porządkowanym ciągu, gdyż na lewo od niego znajdują się wyłącznie wyrazy o wartości mniejszej lub równej *srodek*, a na prawo — o wartości większej lub równej *srodek*. Istotną czynnością w tym algorytmie jest rów-

nież wyznaczenie następnych podciągów do sortowania. Podział następuje po zakończeniu szukania wyrazów do zamiany, czyli po dojściu zmiennych *i* oraz *j* do wyrazu *srodek*. Wówczas, wyznaczając kolejne elementy o numerach *i* oraz *j*, dokonujemy podziału na dwa podciągi: *T[lewy...j]* i *T[i...prawy]*. Operacja dzielenia wykonywana jest dopóty, dopóki ciąg, który dzielimy, ma więcej niż dwa wyrazy.

Metoda sortowania szybkiego nie wymaga, aby wyraz, który służy do podziału ciągu, był wyrazem środkowym. Podział ciągu można przeprowadzić za pomocą dowolnego wyrazu porządkowanego ciągu. Jednak najczęściej stosuje się rozwiązanie omówione powyżej, polegające na wyznaczeniu wyrazu środkowego.

Przykład 2.39.

Przeanalizujmy realizację algorytmu sortowania szybkiego na przykładzie liczbowym: (4, 7, 5, 3, 2). Naszym celem jest uporządkowanie tego ciągu w kolejności od najmniejszego do największego. Przebieg algorytmu został przedstawiony na rysunku 2.21. Niebieskim kolorem zaznaczono wyrazy przeznaczone do zamiany, pogrubieniem — wyraz środkowy przeglądanego ciągu, natomiast szarym tłem wyróżniono wyrazy już posortowane.

Rysunek 2.21. Przebieg algorytmu sortowania szybkiego dla ciągu (4, 7, 5, 3, 2)

Specyfikacja:

Dane: Liczba naturalna: $n > 0$ (liczba elementów tablicy T).

n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.

Wynik: Posortowana niemalejąco n-elementowa tablica jednowymiarowa zawierająca liczby rzeczywiste: $T[0...n-1]$.

Lista kroków:

Krok 0. Wczytaj n, $T[0...n-1]$.

Algorytm rekurencyjny (*T*, *lewy*, *prawy*):

Krok 1. Jeśli *lewy* < *prawy*, przypisz *srodek* = $T[(lewy+prawy)/2]$, w przeciwnym wypadku zakończ algorytm.

Krok 2. Przegrupuj wyrazy ciągu $T[lewy, prawy]$ w dwa podciągi, wykorzystując wyznaczony wyraz środkowy *srodek*. Po podziale wyraz *srodek* znajdzie się na pozycji elementu $T[k]$, dla k spełniającego warunek $lewy \le k \le prawy$; elementy zawarte w podciągu $T[lewy, k]$ będą od niego nie większe, a w podciągu $T[k+1, prawy]$ — nie mniejsze.

Krok 3. Uruchom ten algorytm z parametrami $(T, lewy, k)$.

Krok 4. Uruchom ten algorytm z parametrami $(T, k+1, prawy)$.

Funkcja w języku C++ (prog2_26.cpp):

```
void sortuj (double T[], int lewy, int prawy)
{
 int i=lewy, j=prawy;
 double srodek=T[(lewy+prawy)/2], pom;
 do
 {
  while (T[i]<srodek) i++;
  while (T[j]>srodek) j--;
  if (i<=j)
  {
   pom=T[i];
   T[i]=T[j];
   T[j]=pom;
   i++;
   j--;
  }
 }
 while (i<=j);
 if (lewy<j) sortuj(T,lewy,j);
 if (prawy>i) sortuj(T,i,prawy);
}
```

Procedura w języku Pascal (prog2_26.pas):

```
procedure sortuj (var T: tablica; lewy, prawy: integer);
var i, j: integer;
    srodek, pom: real;
begin
 i:=lewy;
 j:=prawy;
 srodek:=T[(lewy+prawy) div 2];
 repeat
  while T[i]<srodek do i:=i+1;
  while T[j]>srodek do j:=j-1;
  if i<=j then
  begin
   pom:=T[i];
   T[i]:=T[j];
   T[j]:=pom;
   i:=i+1;
   j:=j-1
  end
 until i>j;
 if lewy<j then sortuj(T,lewy,j);
 if prawy>i then sortuj(T,i,prawy)
end;
```

Zastanówmy się nad złożonością czasową przedstawionego algorytmu. Operacją dominującą jest tutaj porównanie. Dane wejściowe mają wpływ na liczbę wykonywanych działań. W optymistycznym i przeciętnym przypadku złożoność jest liniowo-logarytmiczna, co wynika z zastosowania metody „dziel i zwyciężaj". Tylko w najgorszym przypadku, czyli gdy sortujemy ciąg, który jest uporządkowany odwrotnie, realizowanych jest n^2 porównań. Wynika stąd, że złożoność tego algorytmu jest kwadratowa rzędu $O(n^2)$.

Zadanie 2.55. Przedstaw przebieg algorytmu rekurencyjnego realizującego sortowanie szybkie dla podanych ciągów liczbowych (podobnie jak w przykładzie 2.39):

a) (10, 9, 8, 7, 6, 5, 4, 3, 2, 1),

b) (10, 1, 9, 2, 8, 3, 7, 4, 6),

c) (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0).

Zadanie 2.56. Zmodyfikuj przedstawiony algorytm realizujący sortowanie szybkie tak, aby porządkował wyrazy ciągu nierosnąco.