Laboratoria nr 1

Sortowanie

1. Sortowanie bąbelkowe (BbS)
2. Sortowanie przez wstawianie (IS)
3. Sortowanie przez wybieranie (SS)
4. Sortowanie przez zliczanie (CS)
5. Sortowanie kubełkowe (BS)
6. Sortowanie przez scalanie ciągów (MS)
7. Sortowanie stogowe (HS)
8. Sortowanie szybkie (QS)

Materiały

Wyróżniamy następujące metody sortowania:

1. Przez prostą zamianę – bąbelkowe (Bubble Sort) – BbS

Sprawdzamy całą tablicę od dołu do góry (od prawej do lewej strony). Analizowane są zawsze dwa sąsiadujące ze sobą elementy. Jeżeli uporządkowane są one tak, że większy poprzedza mniejszy to zamieniane są one miejscami. Czynność powtarzana jest tak długo, aż podczas sprawdzania całej tablicy, nie zajdzie ani jedna zamiana elementów.

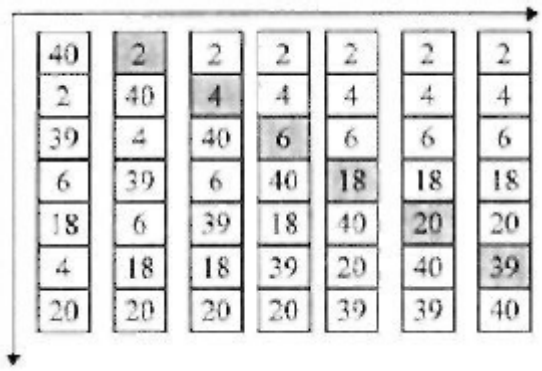
W trakcie pierwszego przebiegu na pierwszą pozycję tablicy (indeks *0)* przesuwa się element „najlżejszy", w trakcie drugiego przebiegu drugi najlżejszy wędruje na drugą pozycję tablicy (indeks 1) i tak dalej, aż do posortowania tablicy. Strefa pracy algorytmu zmniejsza się zatem o 1 w kolejnym przejściu dużej pętli - analizowanie za każdym razem całej tablicy byłoby niepotrzebne.

Algorytm nosi nazwę bąbelkowego przez analogię do pęcherzyków powietrza ulatujących w górę tuby wypełnionej wodą – o ile postawioną pionowo tablicę potraktować jako pojemnik z wodą a liczby jako pęcherzyki powietrza. Najszybciej ulatują do góry „bąbelki" najlżejsze - liczby o najmniejszej wartości (przyjmując sortowanie w kierunku wartości niemalejących).

Idea działania algorytmu, inaczej:

- Algorytm składa się z tzw. fal „bąbli" (powietrza). W każdej fali bierze się po kolei dwa sąsiednie elementy, tzn. 0 i 1, 1 i 2, 2 i 3, ... i zamienia się je miejscami („bąbel"), jeśli element o mniejszym indeksie ma większą wartość.
- Po każdej fali największy element zostaje przesunięty na koniec zbioru. Dlatego każda następna „fala" jest o jeden element krótsza.
- Jeśli w jakiejś „fali" nie wystąpił żaden „bąbel" to znaczy, że wszystkie elementy są uporządkowane od najmniejszego do największego.
- Kolejną „falę" przeprowadza się tylko wtedy kiedy w poprzedniej wystąpił „bąbel" oraz nie wszystkie „fale" zostały wykonane (dla zbioru N-elementowego wykonuje się N-1 fal).
- Przed dojściem do elementu największego „bąblowanie" prowadzi do wstępnego porządkowania elementów mniejszych. Potem największy przesuwa się na koniec zbioru.
- Jeśli zbiór jest uporządkowany, to zostaje wykonana tylko jedna fala „bąbli".

Poniżej znajduje się **przykład** dla nieuporządkowanego ciągu liczb *<40, 2, 39, 6, 18, 4, 20>*.



Inny przykład dla ciągu liczb *<4, 2, 5, 1, 7>*. Kolejne przebiegi algorytmu:

$$[\underbrace{4,2}_{4>2},5,1,7] \rightarrow [2,\underbrace{4,5}_{4<5},1,7] \rightarrow [2,4,\underbrace{5,1}_{5>1},7] \rightarrow [2,4,1,\underbrace{5,7}_{5<7}]$$

$$[\underbrace{2,4}_{2<4},1,5,7] \rightarrow [2,\underbrace{4,1}_{4>1},5,7] \rightarrow [2,1,\underbrace{4,5}_{4<5},7]$$

$$[\underbrace{2,1}_{2>1},4,5,7] \rightarrow [1,\underbrace{2,4}_{2<4},5,7]$$

$$[\underbrace{2,1}_{2>1},4,5,7] \rightarrow [1,\underbrace{2,4}_{2<4},5,7]$$

Tablica wejściowa A w poniższym pseudokodzie zawiera ciąg, który należy posortować. Po zakończeniu procedury tablica A zawiera posortowany ciąg wyjściowy.

PSEUDOKOD

Bubble-Sort (A)

1 **for** $i \leftarrow 1$ **to** length[A]

3 **for** j $\leftarrow$ length[A] -1 **to** i

4 **if** $A[j] < A[j\text{-}1]$

5 **then do** zamień $A[j] \leftrightarrow A[j\text{-}1]$

Algorytm wykonuje n przejść (fale), przy czym w każdej dokonuje n-1 porównań. **Jego złożoność pesymistyczna to $O(n^2)$.**

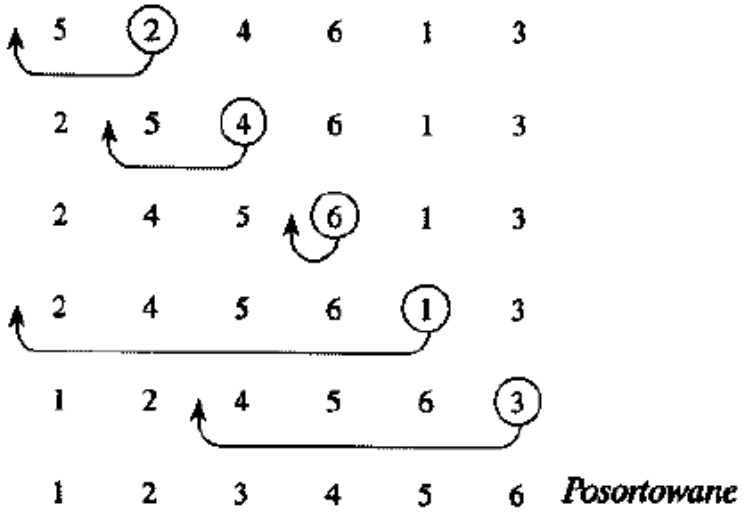
2. Przez proste wstawianie (Insertion Sort) – IS

Algorytm ten działa w taki sposób w jaki ludzie często porządkują talię kart. Zaczynamy od „pustej" lewej ręki, po czym bierzemy ze stołu kolejne karty i wstawiamy je we właściwe miejsca w talii kart, trzymanej w lewej ręce. Aby znaleźć właściwe miejsce dla danej karty, porównujemy ją z kartami, które już mamy w ręce, przesuwając się od strony prawej do lewej.

Opis słowny algorytmu:

1. Utwórz zbiór elementów posortowanych i przenieś do niego dowolny element ze zbioru nieposortowanego.
2. Weź dowolny element ze zbioru nieposortowanego:
    a. Wyciągnięty element porównujemy z kolejnymi elementami zbioru posortowanego póki nie napotkamy elementu równego lub elementu większego (jeśli chcemy otrzymać ciąg niemalejący) lub nie znajdziemy się na początku/końcu zbioru uporządkowanego.
    b. Wyciągnięty element wstaw w miejsce gdzie zakończono porównywanie.
3. Jeśli zbiór elementów nieuporządkowanych jest niepusty wróć do punkt 2 (pobieranie elementu nieposortowanego).

Na poniższym rysunku widać **działanie algorytmu dla tablicy** A=<*5, 2, 4, 6, 1, 3*>. Indeks *j* wskazuje „kartę bieżącą", która jest właśnie wstawiana do talii kart w ręce. Elementy tablicy A[1...j-1] reprezentują karty trzymane w ręce, a elementy A[j+1..n] odpowiadają stosowi kart na stole. Indeks *j* przesuwa się od strony lewej do prawej. W każdej iteracji zewnętrznej pętli **for** element A[j] jest pobierany z tablicy (wiersz 2 w pseudokodzie). Następnie, począwszy od pozycji *j-1*, elementy są sukcesywnie przesuwane o jedną pozycję w prawo, aż zostanie znaleziona właściwa pozycja dla A[j] (wiersze 4-6 w pseudokodzie) i wtedy element ten zostaje tam wstawiony (wiersz 7 w pseudokodzie).



Rys. 1.2. Działanie procedury INSERTION-SORT dla tablicy $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Pozycja o indeksie j jest oznaczona kółkiem

Inny przykład:

Zbiór	Opis operacji
7 3 8 5 **2**	Ostatni element jest zalążkiem listy uporządkowanej.
7 3 8 5 **2**	Ze zbioru wybieramy element leżący tuż przed listą uporządkowaną.
7 3 8 5 **2**	Wybrany element porównujemy z elementem listy.
7 3 8 **2** 5 <<<	Ponieważ element listy jest mniejszy od elementu wybranego, to przesuwamy go na puste miejsce.
7 3 8 **2** 5	Na liście nie ma już więcej elementów do porównania, więc element wybrany wstawiamy na puste miejsce. Lista uporządkowana zawiera już dwa elementy.
7 3 8 **2** 5	Ze zbioru wybieramy 8
7 3 8 **2** 5	8 porównujemy z 2
7 3 **2** <<< 8 **5**	2 jest mniejsze, zatem przesuwamy je na puste miejsce.
7 3 **2** 8 **5**	8 porównujemy z 5
7 3 **2** 5 8 <<<	5 jest mniejsze, przesuwamy je na puste miejsce
7 3 **2** **5** 8	Lista nie zawiera więcej elementów, zatem 8 wstawiamy na puste miejsce. Na liście uporządkowanej mamy już trzy elementy.

Tablica wejściowa A w poniższym pseudokodzie zawiera ciąg, który należy posortować. Po zakończeniu procedury tablica A zawiera posortowany ciąg wyjściowy.

PSEUDOKOD

Insertion-Sort(A)

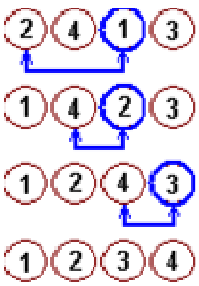
```
1       for i ←1 to length[A]
2           key ← A[i]
3           j = i - 1
4           while j > 0 i A[j] > key
5               A[j+1] ← A[j]
6               j = j - 1
7           A[j + 1] ← key
```

Jego złożoność pesymistyczna to $O(n^2)$.

3. Przez proste wybieranie/wymianę (Selection Sort) – SS

Na początku szukany jest najmniejszy element. Po jego znalezieniu jest on zamieniany z pierwszym elementem tablicy. Następnie szukany jest ponownie najmniejszy element, ale począwszy od elementu drugiego (pierwszy – najmniejszy jest już wstawiony na odpowiednie miejsce). Po jego znalezieniu jest on zamieniany z drugim elementem. Czynność ta jest powtarzana kolejno na elementach od trzeciego, czwartego, itd., aż do *n*-tego.

Poniżej znajduje się przykład zastosowania dla nieuporządkowanego ciągu liczb <*2, 4, 1, 3*>.



```
procedure prostewybieranie;
    var i, j, k: indeks; x: obiekt;
begin for i := 1 to n-1 do
        begin k := i; x := a[i];
            for j := j+1 to n do
                if a[j].klucz < x.klucz then
                begin k := j; x := a[j]
                end;
            a[k] := a[i]; a[i] := x;
            end
end
```

Selection-Sort(A)

```
1    for i ←1 to length[A]
2        key ← A[i]
3        k ← i
4        for j ←i +1 to length[A]
5            if key > A[j]
6                key ← A[j]
7                k ← j
8        A[k] ← A[i]
9        A[i] ← key
```

4. Przez zliczanie (Counting Sort) – CS

Zakładamy, że każdy z *n* sortowanych elementów jest liczbą całkowitą z przedziału od 1 do *k* dla pewnego ustalonego *k*.

Główna idea sortowania przez zliczanie polega na wyznaczeniu dla każdej liczby wejściowej *x*, ile elementów jest od niej mniejszych. Znając tę liczbę, znamy pozycję *x* w ciągu posortowanym, wystarczy ją więc bezpośrednio na tej pozycji umieścić. Sytuacja nieco się zmienia, gdy dozwolonych jest więcej elementów o tej samej wartości, ponieważ nie chcemy, aby wszystkie elementy trafiły na tę samą pozycję.

W procedurze Counting-Sort przyjmujemy, że dane wejściowe są zawarte w tablicy A[1...n], więc *length*[A]=n. Potrzebne są jeszcze dwie dodatkowe tablice: w B[1...n] zostaną umieszczone posortowane dane wejściowe, a w tablicy C[1...k] zapamiętywane są tymczasowe dane pomocnicze:

Sposób sortowania polega na policzeniu najpierw ile razy dana liczba występuje w ciągu (wiersze 3-4 pseudokodu), który mamy posortować i umieszczenie tej informacji w tablicy C (np. C[4]=3, oznacza, że cyfra 4 wystąpiła w tablicy wejściowej *A* 3 razy). Następnie obliczane jest ile elementów jest mniejszych lub równych od elementów w wejściowej tablicy *A* – dodawanie po kolei wartości tablicy C (wiersze 6-7 pseudokodu). Na końcu (w wierszach 9-11 pseudokodu) wszystkie elementy tablicy *A* zostają umieszczone na właściwych pozycjach w tablicy *B*.

Np. mamy posortować ciąg: 3,6,3,2,7,1,7,1. Po zliczeniu (w jednym kroku) posiadamy informacje:

Liczba 1 występuje 2 razy
Liczba 2 występuje 1 raz
Liczba 3 występuje 2 razy
Liczba 4 występuje 0 razy
Liczba 5 występuje 0 razy
Liczba 6 występuje 1 raz
Liczba 7 występuje 2 razy

Na podstawie tych danych tworzymy ciąg: 1,1,2,3,3,6,7,7. Jest to ciąg wejściowy, ale posortowany. Należy zauważyć trzy ważne rzeczy:

- Proces zliczania odbył się w jednym kroku
- Nie doszło do ani jednej zamiany elementów
- Proces tworzenia tablicy wynikowej odbył się w jednym kroku

<u>Algorytm ten posiada jednak wady:</u>

- Do przechowywania liczby wyrazów ciągu musimy użyć tablicy, o liczbie elementów równej największemu elementowi ciągu
- Sortować można jedynie liczby całkowite

<u>Sortowanie przez zliczanie ma jedną potężną zaletę i jedną równie potężną wadę:</u>

- Zaleta: działa w czasie liniowym (jest szybki)
- Wada: może sortować wyłącznie liczby całkowite, do tego najlepiej, żeby największa liczba nie była zbyt duża. Chodzi o to, że największa liczba wyznacza przy okazji ile pamięci będziemy potrzebować na posortowanie przy użyciu tablic statycznych.

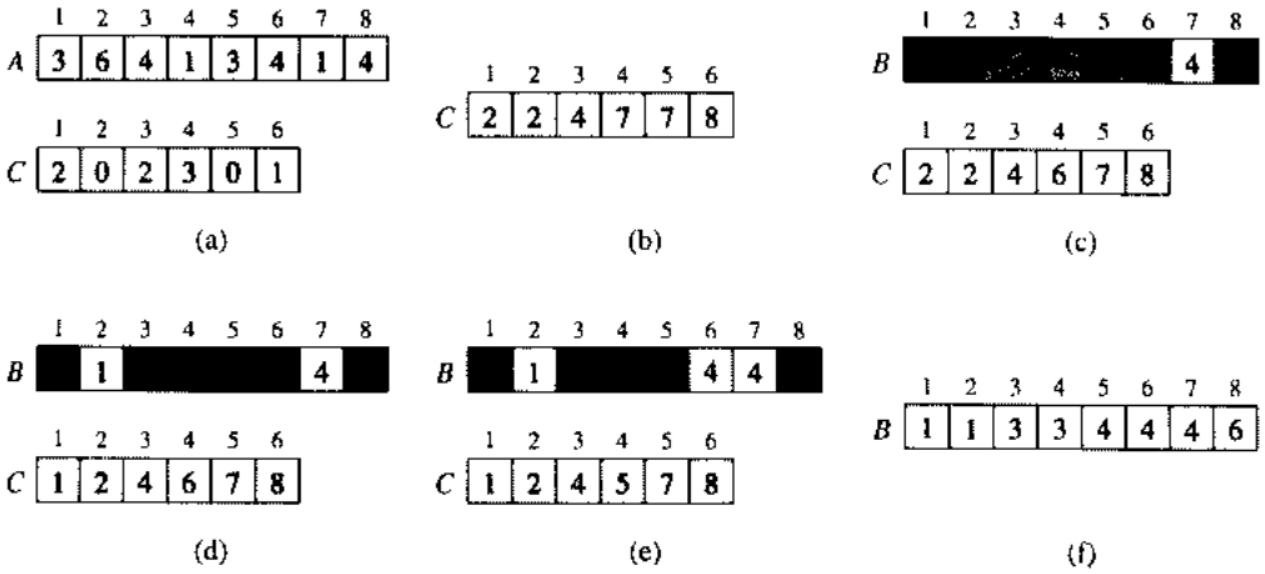
Obydwie te cechy wynikają ze sposobu sortowania. Polega ono na liczeniu, ile razy dana liczba występuje w ciągu, który mamy posortować.

PSEUDOKOD

Counting-Sort (A, B, k)

1 **for** $i \leftarrow 1$ **to** k

```
2          do  C[i] ← 0
3     for  i ←1 to length[A]
4          do C[A[i]] ← C[A[i]] + 1
5     // C[i] zawiera teraz liczbę elementów równych i
6     for  ← 2 to k
7          do  C[i] ← C[i] + C[i-1]
8     // C[i] zawiera liczbę elementów mniejszych lub równych i
9     for i ← length[A] downto 1
10         do B[C[A[i]]] ← A[i]
11            C[A[i]] ← C[A[i]] – 1
```



Rys. 9.2. Działanie procedury COUNTING-SORT dla tablicy wejściowej $A[1..8]$, w której każdy element jest dodatnią liczbą całkowitą nie większą od $k = 6$. **(a)** Tablica A oraz pomocnicza tablica C po wykonaniu wiersza 4. **(b)** Tablica C po wykonaniu wiersza 7. **(c)-(e)** Tablica B oraz pomocnicza tablica C po wykonaniu odpowiednio jednej, dwóch oraz trzech iteracji pętli w wierszach 9-11. Tylko jasnoszare elementy tablicy B zostały wypełnione. **(f)** Ostateczna zawartość tablicy B

5. Kubełkowe (Bucket Sort) – BS

Zakładamy, że dane wejściowe są liczbami rzeczywistymi wybieranymi losowo z przedziału *[0, 1)*, zgodnie z rozkładem jednostajnym.

Sortowanie opiera się na triku polegającym na podziale przedziału *[0, 1)* na *n* podprzedziałów

jednakowych rozmiarów, tzw. *kubełków*, a następnie "rozrzuceniu" *n* liczb do kubełków, do których należą. Ponieważ liczby są jednostajnie rozłożone w przedziale *[0, 1)*, więc oczekujemy, że w każdym z kubełków nie będzie ich zbyt wiele. W celu uzyskania ciągu wynikowego, sortujemy najpierw liczby w każdym z kubełków, a następnie łączy się je wszystkie, przeglądając po kolei kubełki.

Schemat algorytmu:

- Podziel zadany przedział liczb na *n* podprzedziałów (kubełków) o równej długości.
- Przypisz liczby z sortowanej tablicy do odpowiednich kubełków.
- Sortuj liczby w niepustych kubełkach (np. poprzez sortowanie przez wstawianie).
- Wypisz po kolei zawartość niepustych kubełków.

Ciąg wejściowy:

0,78	0,15	0,26	0,02	0,64	0,76	0,63	0,59	0,25	0,74	0,15	0,18	0,82	0,79	0,39

Przydział liczb do kubełków:

Liczba	0,78	0,15	0,26	0,02	0,64	0,76	0,63	0,59	0,25	0,74	0,15	0,18	0,82	0,79	0,39
Numer Kubełka:	7	1	2	0	6	7	6	5	2	7	1	1	8	7	3

Liczby w kubełkach:

przed sortowaniem

Kubełek	Liczby
0	0,02
1	0,15; 0,15; 0,18
2	0,26; 0,25
3	0,39
4	
5	0,59
6	0,64; 0,63;
7	0,78; 0,76; 0,74; 0,79
8	0,82
9	

po sortowaniu

Kubełek	Liczby
0	0,02
1	0,15; 0,15; 0,18
2	0,25; 0,26
3	0,39
4	
5	0,59
6	0,63; 0,64;
7	0,74; 0,76; 0,78; 0,79
8	0,82
9	

Po połączeniu:

0,02	0,15	0,15	0,18	0,25	0,26	0,39	0,59	0,63	0,64	0,74	0,76	0,78	0,79	0,82

Poniżej znajduje się przykład zastosowania dla nieuporządkowanego ciągu liczb *<0.78, 0.15, 0.26, 0.02, 0.64, 0.76, 0.63, 0.59, 0.25, 0.74, 0.15, 0.18, 0.82, 0.79, 0.39>.*

Inny przykład:

Dane:		0.32	0.74	0.15	0.23	0.21	0.17	0.45	0.28	0.43	0.90	SORTOWANIE	
KUBEŁKI	L[0]	→					L[0]	→					
	L[1]	→	.15	.17			L[1]	→	.15	.17			
	L[2]	→	.23	.21	.28		L[2]	→	.21	.23	.28		
	L[3]	→	.32				L[3]	→	.32				
	L[4]	→	.45	.43			L[4]	→	.43	.45			
	L[5]	→					L[5]	→					
	L[6]	→					L[6]	→					
	L[7]	→	.74				L[7]	→	.74				
	L[8]	→					L[8]	→					
	L[9]	→	.90				L[9]	→	.90				
po rozrzuceniu							po posortowaniu						
Po połączeniu kubełków				.15	.17	.21	.23	.28	.32	.43	.45	.74	.90

PSEUDOKOD:

BUCKET-SORT(A)

1 $n \leftarrow length[A]$

2 **for** $i \leftarrow 1$ **to** n

3 **do** wstaw $A[i]$ do listy $B[\lfloor nA[i] \rfloor]$

4 **for** $i \leftarrow 0$ **to** $n - 1$

5 **do** posortuj listę $B[i]$ przez wstawianie

6 połącz listy $B[0]$, $B[1]$,..., $B[n - 1]$ dokładnie w tej kolejności

Ustalenie który element trafi do którego kubełka jest realizowane w 'czasie liniowym' – **O(n)**
Insertion Sort ma teoretycznie złożoność pesymistyczną równą $O(n^2)$, więc pesymistyczna złożoność dla *Bucket Sort* to właśnie **$O(n^2)$**
Nie jest jednak aż tak źle – ponieważ elementy trafiają do różnych kubełków, oczekiwana liczba operacji to **O(n) + O(2 – 1/n)**, czyli ogólnie – liniowa.

6. Przez scalanie (Merge Sort) – MS

Scalanie ciągów polega na łączeniu posortowanych ciągów w jeden ciąg posortowany. Ciągi scala się parami, poczynając od pierwszych dwóch. Ustawiamy liczniki obu ciągów na 1 (co wskazuje na pierwszy element ciągu). Sprawdzamy, który z elementów jest mniejszy i ten element przenosimy do ciągu wynikowego, a licznik ciągu, którego element był mniejszy zwiększamy o 1. Następnie sprawdzamy kolejne elementy wskazywane przez liczniki i robimy to tak długo, aż liczniki będą wskazywać na ostatnie elementy swoich ciągów. Gdy tak się stanie, w ciągu wynikowym będziemy mieli scalone dwa ciągi o liczbie wyrazów będącej sumą

elementów ciągów scalanych. Po scaleniu dwóch pierwszych ciągów scalamy ciąg wynikowy i trzeci. Po tej operacji w ciągu wynikowym będziemy mieli scalone 3 pierwsze ciągi. Następnie scalamy ciąg wynikowy z ciągiem 4, itd...

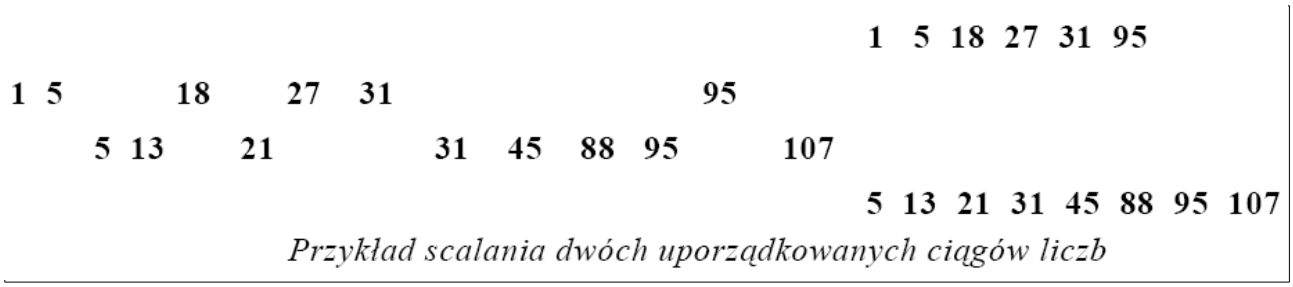
Ogólnie można wyróżnić można trzy podstawowe kroki:

- Podziel zestaw danych na dwie, równe części (w przypadku nieparzystej liczby wyrazów jedna część będzie o 1 wyraz dłuższa);
- Zastosuj sortowanie przez scalanie dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element;
- Połącz posortowane podciągi w jeden.

Procedura scalania dwóch ciągów A[1..n] i B[1..m] do ciągu C[1..m+n]:

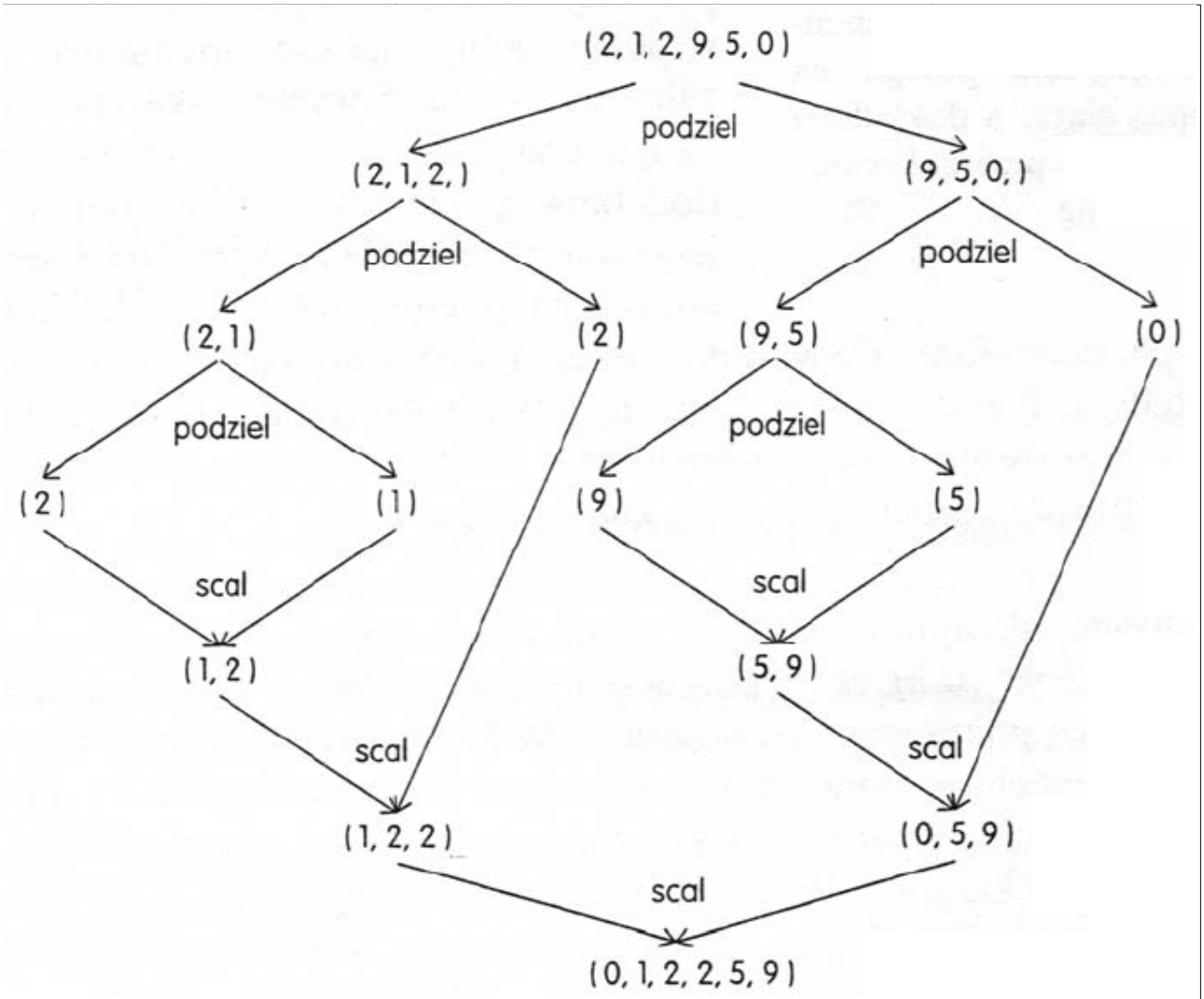
- Utwórz wskaźniki na początki ciągów A i B → i=1, j=1
- Jeżeli ciąg A wyczerpany (*i*>*n*), dołącz pozostałe elementy ciągu B do C i zakończ pracę.
- Jeżeli ciąg B wyczerpany (*j*>*m*), dołącz pozostałe elementy ciągu A do C i zakończ pracę.
- Jeżeli *A[i]* ≤ *B[j]* dołącz *A[i]* do *C* i zwiększ *i* o jeden, w przeciwnym przypadku dołącz *B[j]* do *C* i zwiększ *j* o jeden
- Powtarzaj od kroku 2 aż wszystkie wyrazy *A* i *B* trafią do *C*

Poniżej znajduje się przykład zastosowania dla nieuporządkowanych ciągów liczb <*1, 5, 18, 27, 31, 95*> oraz <*5, 13, 21, 31, 45, 88, 95, 107*>.



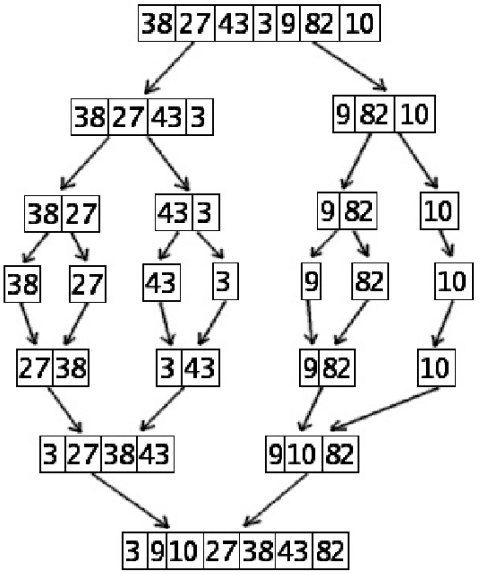
Przykład scalania dwóch uporządkowanych ciągów liczb

Poniżej znajduje się przykład zastosowania dla nieuporządkowanego ciągu liczb <*2, 1, 2, 9, 5, 0*>.



Ilustracja zasady działania algorytmu sortowania przez scalanie

Inny przykład:



PSEUDOKOD:

MERGE-SORT(*A, p, r*)

1 **if** $p < r$

2 **then** $q \rightarrow \lfloor (p + r)/2 \rfloor$

3 MERGE-SORT(*A, p, q*)

4 MERGE-SORT(*A, q*+1, *r*)

5 MERGE(*A, p, q, r*)

A = tablica

p, q, r = indeksy, takie, że $p \leq q < r$

Zakłada się, że podtablice *A[p..q], A[q + 1.. r]* są posortowane. Procedura MERGE-SORT scala te tablice w jedną posortowaną tablicę *A[p..r]*.

Złożoność obliczeniowa: O(n log n)

Wersja nierekurencyjna:

- Podstawową wersję algorytmu sortowania przez scalanie można uprościć. Pomysł polega na odwróceniu procesu scalania serii. Ciąg danych możemy wstępnie podzielić na n serii długości 1, scalić je tak, by otrzymać n/2 serii długości 2, scalić je otrzymując n/4 serii długości 4, itd..
- Złożoność obliczeniowa jest taka sama jak w przypadku klasycznym, jednak nie korzystamy wtedy z rekurencji, a więc zaoszczędzamy czas i pamięć potrzebną na jej obsłużenie.

Przez łączenie naturalne

Dane, które chcemy posortować są przechowywane w plikach. Nie mamy bezpośredniego dostępu do dowolnego elementu ciągu do posortowania (teoretycznie mamy, ale wyszukanie kolejnego elementu w ciągu znajdującego się przed elementem, który pobraliśmy ostatnio, wymaga przejścia pliku od początku). Wykorzystujemy dwa pliki pomocnicze. Zakładamy, że sortujemy niemalejąco.

1. Rozdzielamy seriami plik wejściowy na dwa pliki pomocnicze. To znaczy przechodząc plik wejściowy od początku, wyszukujemy ciągi posortowane i zamiennie zapisujemy na dwa pomocnicze pliki. W praktyce wygląda to tak, że pobiera się *n*-ty oraz *(n+1)*-wszy element z pliku i sprawdza się, czy *(n+1)*-wszy jest większy lub równy *n*-temu. Jeśli tak, to należy do ciągu i zapisujemy go do tego samego pliku pomocniczego co element *n*-ty. Jeśli nie, to zapisujemy go na inny (drugi) plik pomocniczy i tak w kółko, aż do zakończenia pliku wejściowego.

2. Łączymy serie z plików pomocniczych i kopiujemy do pliku wejściowego (wejściowy lub dla ochrony danych można sobie wziąć kolejny plik pomocniczy i na początku po prostu skopiować plik wejściowy do niego). Łączenie przebiega analogicznie do scalania ciągów, z tą różnicą, że tutaj oba pliki wejściowe niekoniecznie są całkowicie posortowane.
3. Wykonujemy zamiennie kroki 1 i 2, aż do momentu, w którym otrzymamy w pliku wejściowym tylko jedną serię, która jest naszym posortowanym ciągiem.

Poniżej znajduje się przykład zastosowania dla nieuporządkowanego ciągu liczb <*1, 9, 8, 7, 4, 5, 7, 6, 1, 4, 0, 9, 8, 1, 3, 4, 8, 1, 7*>.

Po pierwszym kroku:
Pierwszy plik pomocniczy: 19 7 6 09 134 8 (odstępy pomiędzy seriami)
Drugi plik pomocniczy: 8 457 14 8 17
Plik wejściowy: 1894577146089113478

Po drugim kroku:
Pierwszy plik pomocniczy: 189 146 113478
Drugi plik pomocniczy: 4577 089
Plik wejściowy: 1457789014689113478

Po trzecim kroku:
Pierwszy plik pomocniczy: 1457789 113478
Drugi plik pomocniczy: 014689
Plik wejściowy: 0114456778899 113478

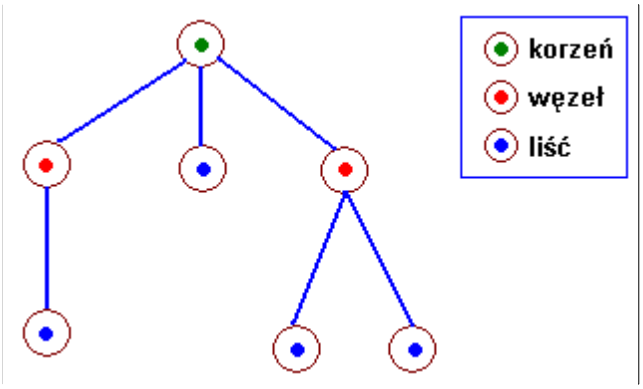
Po czwartym kroku:
Pierwszy plik pomocniczy: 0114456778899
Drugi plik pomocniczy: 113478
Plik wejściowy: 0111134445677788899

7. Przez kopcowanie – stogowe (Heapsort) – HS

Czas działania algorytmu wynosi O(n log n). Algorytm heapsort sortuje w miejscu, tzn. elementy wejściowej tablicy są przechowywane cały czas w tej samej tablicy, tylko stała liczba elementów tablicy jest w czasie działania algorytmu przechowywana poza tablicą wejściową. Używa się tu struktury danych, zwanej „kopcem”, do przetwarzania danych w czasie działania algorytmu.

- **Drzewo:**

Dla każdego drzewa wyróżniony jest jeden, charakterystyczny element – korzeń. Korzeń jest jedynym elementem drzewa, który nie posiada elementów poprzednich. Dla każdego innego elementu określony jest dokładnie jeden element poprzedni. Dla każdego elementu oprócz ostatnich, tzw. liści, istnieją co najmniej 2 elementy następne. Jeżeli liczba następnych elementów wynosi dokładnie 2 to drzewo nazywamy binarnym.

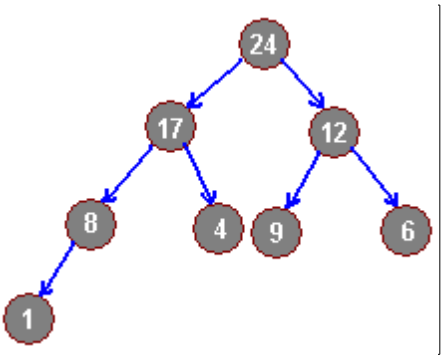


- **Kopiec (stóg):**

Kopiec (binarny), inaczej zwany stogiem, jest to tablicowa struktura danych, którą można rozpatrywać jako pełne *drzewo binarne*, które spełnia tzw. *warunek kopca (*każdy następnik jest niewiększy od swego poprzednika). Z warunku tego wynikają szczególne własności kopca:

  - w korzeniu kopca znajduje się największy element,
  - na ścieżkach (połączeniach między węzłami), od korzenia do liścia, elementy są posortowane nierosnąco.

Każdy węzeł drzewa odpowiada elementowi tablicy, w którym jest podana wartość węzła. Drzewo jest pełne na wszystkich poziomach z wyjątkiem być może najniższego, który jest wypełniony od strony lewej do pewnego miejsca.



Po zbudowaniu drzewa należy wykonać odpowiednie instrukcje, które zapewnią mu warunek kopca. Należy więc sprawdzać (poczynając od poprzednika ostatniego liścia schodząc w górę do korzenia), czy poprzednik jest mniejszy od następnika i jeżeli tak jest to zamienić je miejscami. Po wykonaniu tych czynności drzewo binarne zamienia się w stóg. Z jego własności wynika, że w korzeniu znajduje się największy element. Korzystając z tego faktu, możemy go pobrać na koniec tablicy wynikowej, a na jego

miejsce wstawić ostatni liść. Po pobraniu korzenia tablica źródłowa zmniejsza się o 1 element, a porządek kopca zostaje zaburzony (nie wiadomo, czy ostatni liść będący teraz korzeniem jest rzeczywiście największym elementem). By przywrócić warunek stogu należy ponownie uporządkować jego elementy, tym razem jednak zaczynając od korzenia (ponieważ to on jest nowym elementem). Po przywróceniu porządku w kopcu możemy ponownie pobrać korzeń i wstawić go do tablicy wynikowej (tym razem na drugie miejsce od końca), wstawić na jego miejsce liść i zmniejszyć rozmiar tablicy źródłowej o 1. Wykonujemy te czynności aż do ostatniego korzenia. Po całkowitym wyczyszczeniu kopca w tablicy wynikowej będziemy mieli posortowane elementy z tablicy wejściowej.

Tablica *A* reprezentująca kopiec ma dwa atrybuty: *length[A]* – liczba elementów tablicy i *heap-size[A]* – liczba elementów kopca przechowywanych w tablicy. Oznacza to, że żaden element tablicy *A[1..length(A)]* występujący po *A[heap-size[A]],* gdzie $heap-size[A] \leq length[a]$, nie jest elementem kopca. Korzeniem drzewa jest *A[1]*. Mając dany indeks *i* węzła można łatwo obliczyć indeks ojca *Parent(i) to i/2*, lewego syna *Left(i) to 2i* oraz prawego syna *Right(i) to 2i + 1*.

Procedura *Heapsort* sortuje tablicę w miejscu.

Procedura *Build-Heap* tworzy kopiec z nieuporządkowanej tablicy wejściowej.

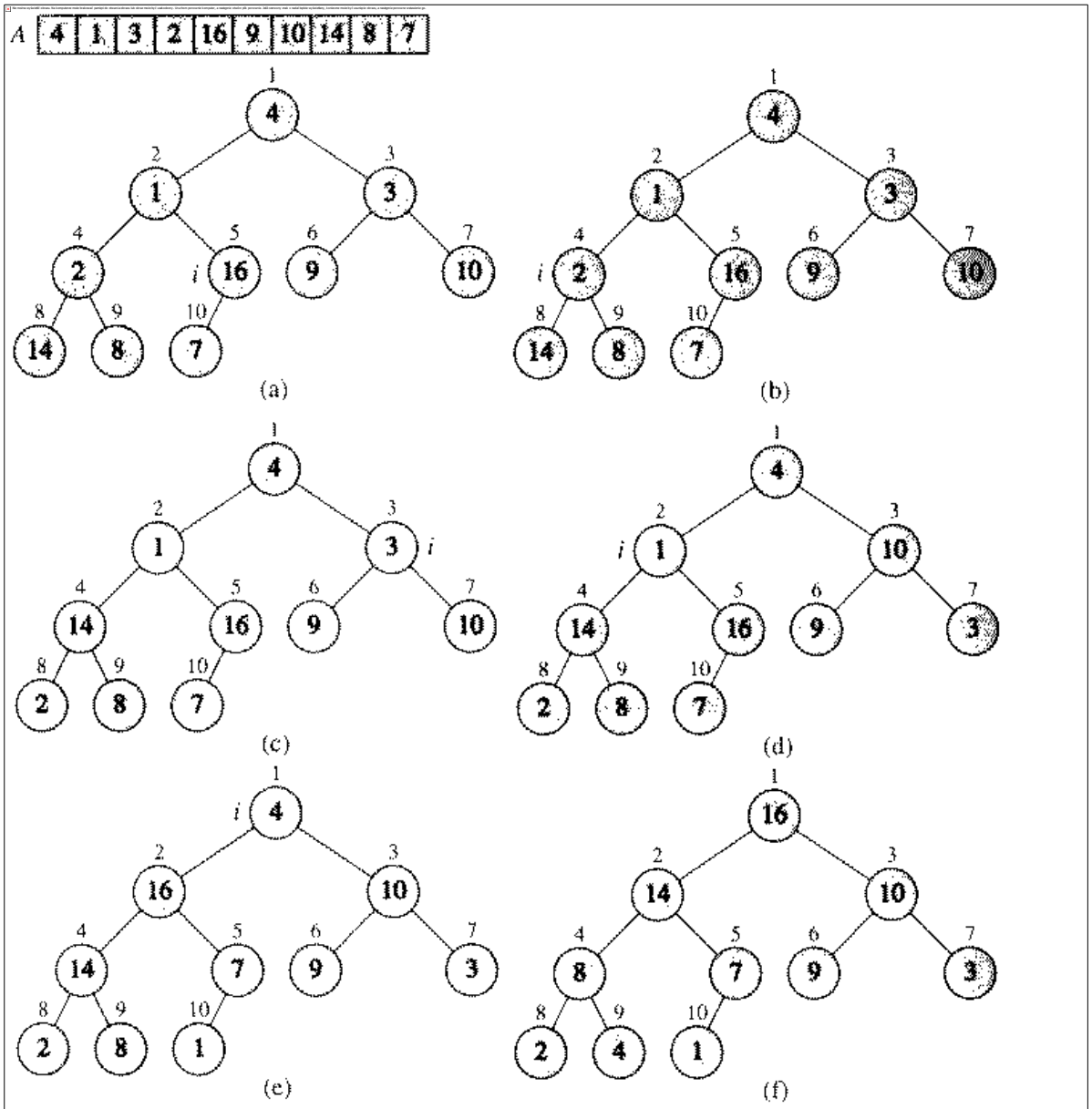
Procedura *Heapify* służy do przywracania własności kopca. Zakłada, że drzewa binarne zaczepione w *Left(i)* i *Right(i)* są kopcami oraz, że *A[i]* może być mniejszy od swoich synów i naruszać w ten sposób własność kopca. W procedurze tej wybierany jest największy element spośród synów oraz *A[i]*. Jeśli największy jest *A[i]* to poddrzewo zaczepione w *i* jest kopcem i procedura kończy działanie. Jeśli nie to następuje zamiana *A[i]* z największym z synów i wywołanie rekurencyjne procedury *Heapify* dla syna z którym nastąpiła zamiana.

```
HEAPSORT(A)
1   BUILD-HEAP(A)
2   for i ← length[A] downto 2
3       do zamień A[1] ↔ A[i]
4          heap-size[A] ← heap-size[A] − 1
5          HEAPIFY(A, 1)
```

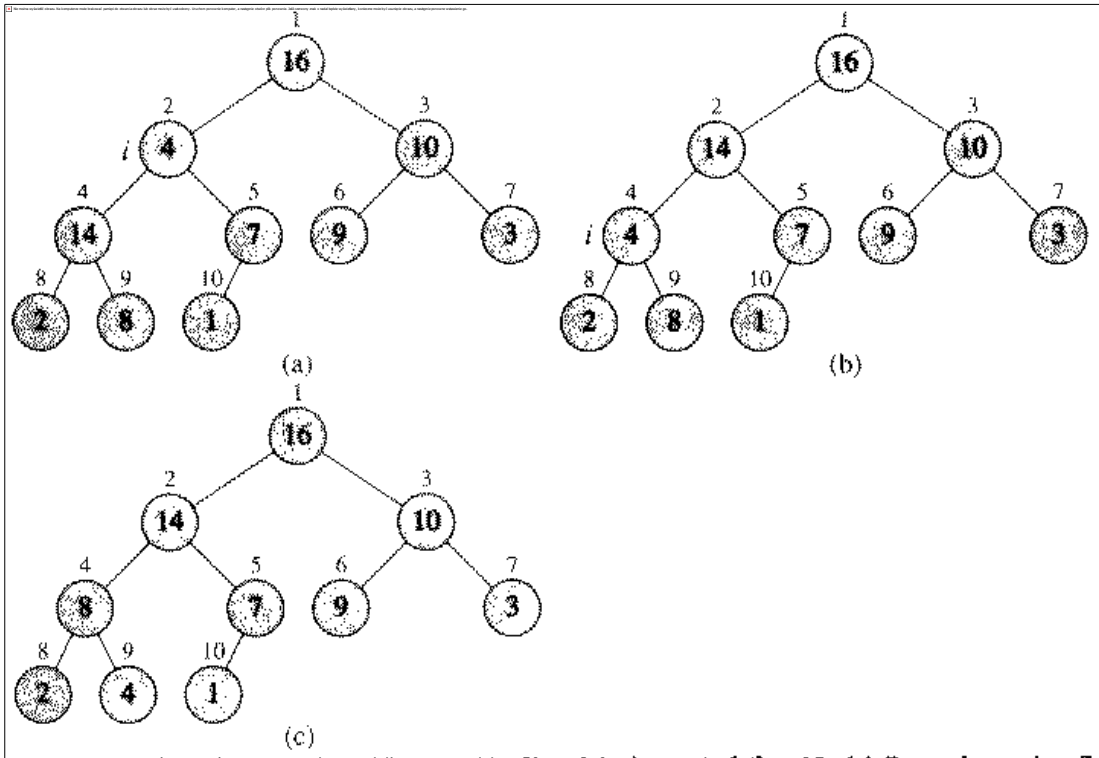
```
BUILD-HEAP(A)
1   heap-size[A] ← length[A]
2   for i ← ⌊length[A]/2⌋ downto 1
3       do HEAPIFY(A, i)
```

```
HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ⩽ heap-size[A] i A[l] > A[i]
 4      then largest ← l
 5      else largest ← i
 6  if r ⩽ heap-size[A] i A[r] > A[largest]
 7      then largest ← r
 8  if largest ≠ i
 9      then zamień A[i] ↔ A[largest]
10          HEAPIFY(A, largest)
```

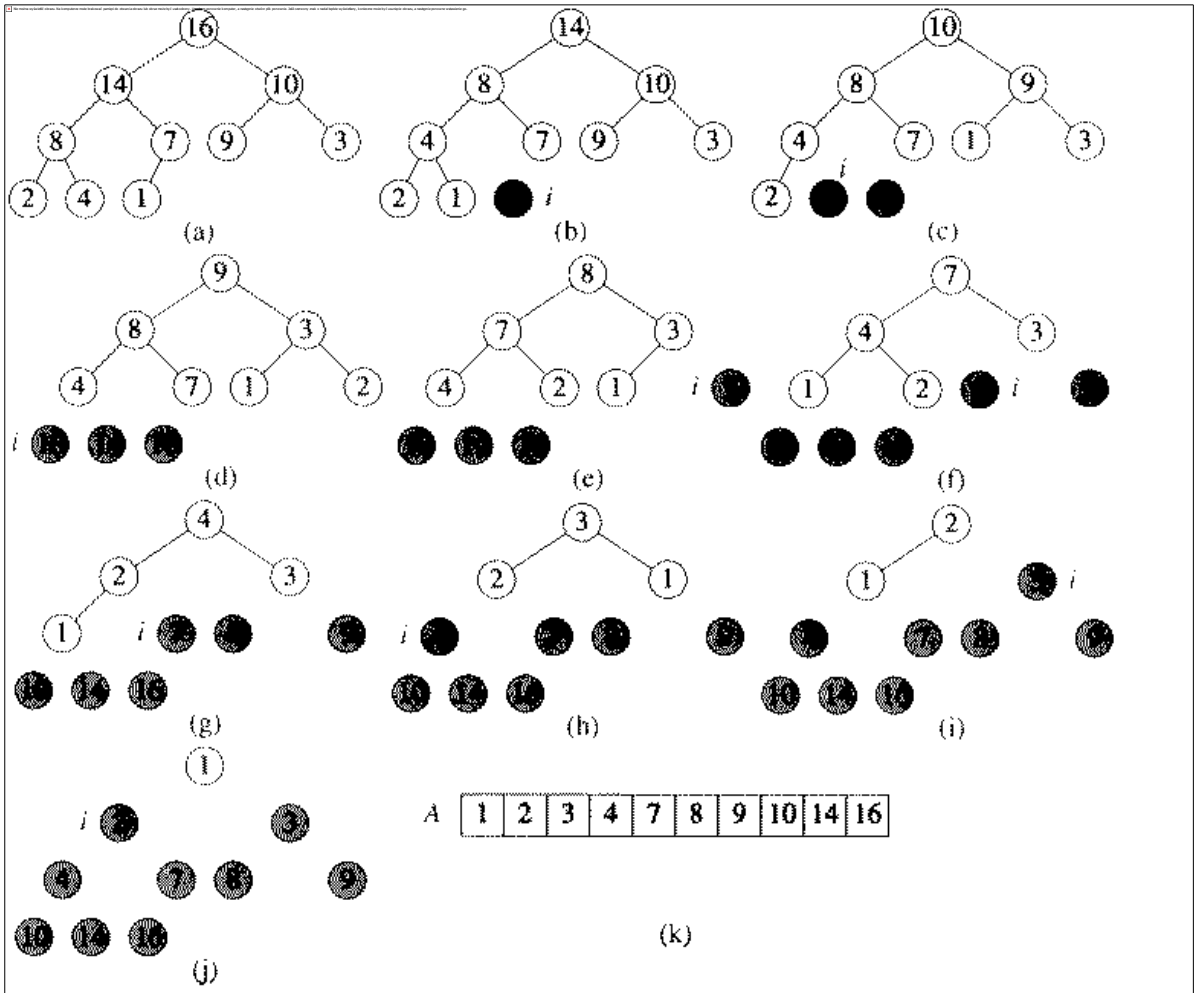
Kopiec można rozpatrywać jako drzewo binarne i jako tablicę. Liczba w kółku w każdym węźle drzewa jest wartością przechowywaną w tym węźle. Liczba obok węzła jest odpowiadającym mu indeksem tablicy.



Rys. 7.3. Działanie procedury BUILD-HEAP, pokazujące strukturę danych przed wywołaniem HEAPIFY w wierszu 3 BUILD-HEAP. **(a)** 10-elementowa tablica wejściowa i reprezentowane przez nią drzewo binarne. Na rysunku widać indeks *i* pętli, który wskazuje na węzeł 5 przed wywołaniem HEAPIFY(*A*, *i*). **(b)** Struktura danych, która jest rezultatem wywołania. Indeks pętli do następnego wywołania wskazuje na węzeł 4. **(c)-(e)** Następne iteracje pętli **for** w BUILD-HEAP. Zauważ, że kiedy HEAPIFY jest wywoływana w węźle, oba poddrzewa tego węzła są kopcami. **(f)** Kopiec po zakończeniu działania BUILD-HEAP



Rys. 7.2. Działanie procedury HEAPIFY(A, 2), gdzie *heap-size*[*A*] = 10. **(a)** Początkowa konfiguracja kopca, w której wartość *A*[2] w węźle $i = 2$ narusza własność kopca, gdy nie jest on większy od obu swoich synów. Własność kopca jest przywracana węzłowi 2 w **(b)** przez zamianę *A*[2] z *A*[4], co narusza własność kopca w węźle 4. Rekurencyjne wywołanie HEAPIFY(*A*, 4) ustawia $i = 4$. Po zamianie *A*[4] z *A*[9], co jest pokazane w **(c)**, węzeł 4 jest poprawiony, a rekurencyjne wywołanie HEAPIFY(*A*, 9) nie zmienia więcej struktury danych



Rys. 7.4. Działanie procedury HEAPSORT. **(a)** Struktura kopca zaraz po jego zbudowaniu przez BUILD-HEAP. **(b)-(j)** Kopiec po każdym wywołaniu HEAPIFY w wierszu 5. Wartość i w tym momencie jest pokazana na rysunku. Tylko jasnoszare węzły pozostaną w kopcu. **(k)** Posortowana tablica A

8. Szybkie (Quicksort) – QS

Algorytm sortowania szybkiego opiera się na strategii "dziel i zwyciężaj" (ang. *divide and conquer*), którą możemy krótko scharakteryzować w trzech punktach:

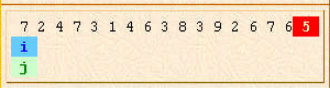
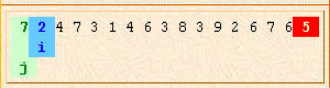
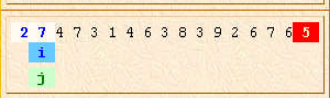
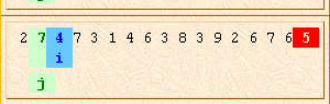
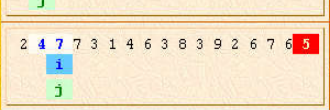
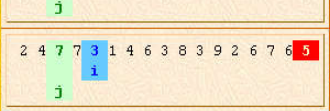
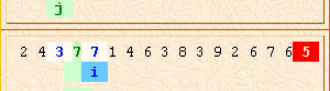
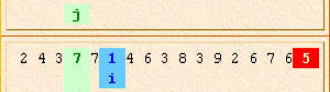
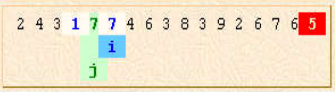
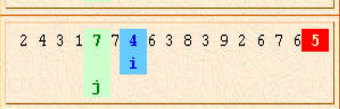
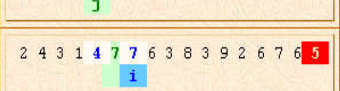
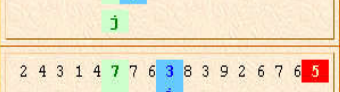
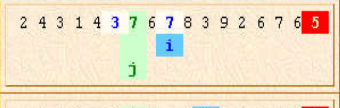
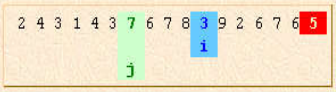
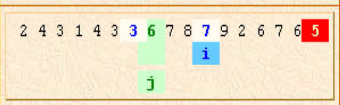
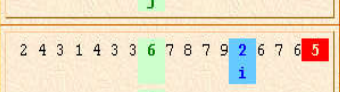
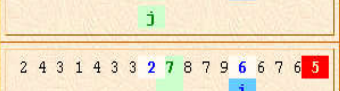
- DZIEL - problem główny zostaje podzielony na podproblemy
- ZWYCIĘŻAJ - znajdujemy rozwiązanie podproblemów
- POŁĄCZ - rozwiązania podproblemów zostają połączone w rozwiązanie problemu głównego

Idea sortowania szybkiego jest następująca:

- **DZIEL:** najpierw sortowany zbiór dzielimy na dwie części w taki sposób, aby wszystkie elementy leżące w pierwszej części (zwanej lewą partycją) były mniejsze lub równe od wszystkich elementów drugiej części zbioru (zwanej prawą partycją).
- **ZWYCIĘŻAJ:** każdą z partycji sortujemy rekurencyjnie tym samym algorytmem.
- **POŁĄCZ:** połączenie tych dwóch partycji w jeden zbiór daje w wyniku zbiór posortowany.

Zbiór danych zostaje podzielony na dwa podzbiory i każdy z nich jest sortowany niezależnie od drugiego. Do utworzenia podzbioru musimy ze zbioru wybrać jeden z elementów, który nazwiemy piwotem. W lewym podzbiorze znajdą się wszystkie elementy nie większe od piwotu, a w prawym podzbiorze znajdą się wszystkie elementy niemniejsze od piwotu. Położenie elementów równych nie wpływa na proces sortowania, zatem mogą one występować w obu podzbiorach. Również porządek elementów w każdym z podzbiorów nie jest ustalony. Jako piwot można wybierać element pierwszy, środkowy, ostatni, medianę lub losowy. Dla naszych potrzeb wybierzemy element środkowy.

Dzielenie na partycje polega na umieszczeniu dwóch wskaźników na początku zbioru - i oraz j. Wskaźnik i przebiega przez zbiór poszukując wartości mniejszych od piwotu. Po znalezieniu takiej wartości jest ona wymieniana z elementem na pozycji j. Po tej operacji wskaźnik j jest przesuwany na następną pozycję. Wskaźnik j zapamiętuje pozycję, na którą trafi następny element oraz na końcu wskazuje miejsce, gdzie znajdzie się piwot. W trakcie podziału piwot jest bezpiecznie przechowywany na ostatniej pozycji w zbiorze.

Lp.	Operacja	Opis
1.		Wyznaczamy na piwot element środkowy.
2.		Piwot wymieniamy z ostatnim elementem zbioru
3.		Na początku zbioru ustawiamy dwa wskaźniki. Wskaźnik *i* będzie przeglądał zbiór do przedostatniej pozycji. Wskaźnik *j* zapamiętuje miejsce wstawiania elementów mniejszych od piwotu
4.		Wskaźnikiem *i* szukamy elementu mniejszego od piwotu
5.		Znaleziony element wymieniamy z elementem na pozycji *j*-tej. Po wymianie wskaźnik *j* przesuwamy o 1 pozycję.
6.		Szukamy
7.		Wymieniamy i przesuwamy *j*.
8.		Szukamy
9.		Wymieniamy i przesuwamy *j*.
10.		Szukamy
11.		Wymieniamy i przesuwamy *j*.
12.		Szukamy
13.		Wymieniamy i przesuwamy *j*.
14.		Szukamy
15.		Wymieniamy i przesuwamy *j*.
16.		Szukamy
17.		Wymieniamy i przesuwamy *j*.
18.		Szukamy
19.		Wymieniamy i przesuwamy *j*.
20.		Brak dalszych elementów do wymiany. *Piwot* wymieniamy z elementem na pozycji *j*-tej. Podział na partycje zakończony.

PSEUDOKOD:

QUICKSORT(*A, p, r*)

```
1   if p < r
2       then q → PARTITION(A, p, r)
3            QUICKSORT(A, p, q)
4            QUICKSORT(A, q+1, r)
```

Najpierw wybierany jest element *x=A[p]* z *A[p..r]* jako element rozdzielający, względem którego będzie dokonywał się podział *A[p..r]*. Następnie na początku i końcu podtablicy *A[p..r]* tworzone są odpowiednio obszary *A[p..i]* i *A[j..r]* takie, że każdy element *A[p..i]* jest mniejszy lub równy *x* i każdy element z *A[j..r]* jest większy lub równy *x*. Na początku obszary są puste *i=p-1* i *j=r+1*. Wewnątrz pętli *while* indeks *j* jest zmniejszany, a indeks *i* zwiększany, aż zajdzie $A[i] \geq x \geq A[j]$. Pętla *while* powtarzana jest tak długo, aż $i \geq j$ (indeksy się miną) a wtedy cała tablica *A[p..r]* została podzielona na dwie takie podtablice *A[p..q]* i *A[q+1..r]*, gdzie żaden element z *A[p..q]* nie jest większy od jakiegokolwiek elementu z *A[q+1..r]*. Wartość *q=j* jest zwracana na końcu procedury.

PARTITION (*A, p, r*)

```
1   x ← A[p]
2   i ← p – 1
3   j ← r + 1
4   while TRUE
5       do repeat j ← j – 1
6          until A[j] ≤ x
7          repeat i ← i + 1
8          until A[i] ≥ x
9          if i < j
10         then zamień A[i] ↔ A[j]
11         else return j
```

W celu posortowania tablicy wywołujemy QUICKSORT(A, 1, length[A]). Główna procedura algorytmu PARTITION przestawia elementy podtablicy A[p..r] w miejscu.

Złożoność czasowa optymistyczna: $O(n \log_2 n)$

Złożoność czasowa pesymistyczna: $O(n^2)$

Literatura:

Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest
Wprowadzenie do algorytmów
Wydanie czwarte, Wydawnictwo Naukowo-Techniczne, Warszawa, 2001

N. Wirth
Algorytmy + struktury danych = programy
Wydawnictwo Naukowo-Techniczne, Warszawa, 2004

http://edu.pjwstk.edu.pl/wyklady/asd/scb/index00.html

http://edu.i-lo.tarnow.pl/inf/alg/003_sort/index.php

http://pl.wikipedia.org/wiki/Sortowanie