3. Algorytmy sortowania

3.1 Problem sortowania

1. W tym rozdziale omawiamy głównie klasyczne algorytmy sortowania ale zajmiemy się również zagadnieniami ściśle związanymi z tym tematem a mianowicie:

- Obliczaniem wartości statystyk pozycyjnych a w szczególności mediany oraz maksymalnego i minimalnego elementu *n* elementowego ciągu skończonego. Problemy tego typu noszą ogólną nazwę „zadania wyboru" a algorytmy rozwiązujące zadanie wyboru nazywamy algorytmami wyboru.

- Układami sortującymi (sieciami sortującymi) i sortowaniem równoległym

Obliczanie wartości statystyk pozycyjnych sprowadza się do wyznaczania *k*-tego co do wielkości elementu w *n* elementowym ciągu.

Sieci sortujące to bardzo szybkie cyfrowe układy sortujące o dużym stopniu zrównoleglenia realizowane z reguły w postaci tzw. sieci systolicznych (ang, systolic arrays). Sieci sortujące mogą być wykorzystane np. jako filtry medianowe lub kwantylowe w cyfrowym przetwarzaniu sygnałów.

Poniżej rozważamy również tematy ściśle związane z sortowaniem, takie jak scalanie (dwóch posortowanych ciągów w jeden posortowany ciąg), struktury danych zwane kopcami. i kolejki priorytetowe.

Algorytmy sortowania są omawiane w każdym podręczniku o algorytmach komputerowych. Najobszerniejszą monografią o algorytmach sortowania jest tom 3 monografii D.E.Knutha „Sztuka programowania" (The Art of Computer Programming) z dużą ilością zadań i problemów do samodzielnego rozwiązania. Proste wprowadzenie w tematykę algorytmów sortowania można znaleźć w podręczniku T.Cormena, C.Leisersona, R.Rivesta i C.Steina „Wprowadzenie do algorytmów" (Introduction to Algorithms) oraz w podręczniku L.Banachowskiego,K.Diksa i W.Ryttera „Algorytmy i struktury danych". Polecamy również bardzo starannie napisany podręcznik Richarda Neapolitana i Kumarssa Naimpoura „Podstawy algorytmów z przykładami w C++".
Krótki ale pozbawiony głębszych analiz przegląd algorytmów sortowania zawiera książka P.Wróblewskiego „Algorytmy – struktury danych i techniki programowania".

2. Algorytmy sortowania – lista bardziej znanych algorytmów sortowania

1. Algorytm sortowania bąbelkowego (ang. bublesort)

2. Zmodyfikowany algorytm sortowania bąbelkowego (ang. modified bublesort)

3. Algorytm sortowania przez wstawianie (ang. insertionsort lub insertsort)

4. Scalanie i algorytm sortowania przez scalanie (ang. mergesort)

5. Algorytm sortowania przez selekcję (przez wybieranie) (ang. selectionsort)
6. Algorytm sortowania kubełkowego (ang. bucketsort)

7. Algorytm sortowania turniejowego (ang. tournamentsort)

8. Algorytm sortowania pozycyjnego (ang. radixsort)

9. Algorytm sortowania przez kopcowanie (ang. heapsort)

10. Algorytm sortowania szybkiego (ang. quicksort)

11. Algorytm sortowania Shella

12. Sortowanie przez zliczanie (ang. countsort)

13. Algorytm sortowania koktajlowego

Można by zapytać po co tyle różnych algorytmów sortowania czy nie ma jednego najlepszego algorytmu.

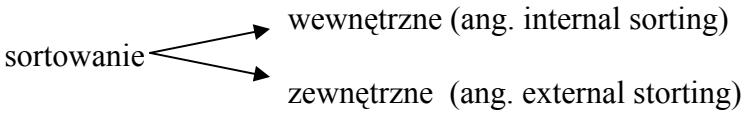
Trudno na to pytanie odpowiedzieć jednym zdaniem. W dalszym ciągu rozdziału będziemy starali się ocenić efektywność algorytmów sortowania w funkcji rozmiaru danych wejściowych *n* biorąc za punkt wyjścia ilość porównań niezbędnych do realizacji algorytmu. Kryterium takie daje dobry, przejrzysty ale nieco niepełny obraz efektywności algorytmu. Liczba operacji dodatkowych np. w przypadku rekurencji może być duża i nie zawsze można te dodatkowe operacje zaniedbać w ocenie efektywności algorytmu.

Wpływ na efektywność algorytmu oprócz rozmiaru danych wejściowych *n* ma również rozkład prawdopodobieństwa na przestrzeni danych wejściowych.

Niekiedy też algorytm asymptotycznie nieefektywny doskonale zachowuje się dla niewielkich *n*. Czasami algorytm uważany za bardzo efektywny zachowuje się znacznie gorzej niż inne algorytmy dla pewnego typu danych wejściowych.

Ponadto w ocenie algorytmu istotna jest możliwość zrównoleglenia wykonywanych operacji tak jak np. w algorytmie modified bubblesort lub mergesort. Ważna jest również możliwość układowej realizacji algorytmu sortowania.

3. Algorytmy sortowania dzieli się na wewnętrzne i zewnętrzne. Algorytmy sortowania wykorzystujące tylko pamięć operacyjną systemu komputerowego nazywane są algorytmami sortowania wewnętrznego (ang. internal storting). Algorytmy sortowania wykorzystujące pamięć zewnętrzną są nazywane algorytmami sortowania zewnętrznego lub krótko sortowaniem zewnętrznym (ang. external storting).



W praktyce podział na algorytmy sortowania wewnętrznego i zewnętrznego oznacza, że inne algorytmy stosujemy dla stosunkowo niewielkich zestawów danych (mieszczących się w pamięci operacyjnej komputera) a inne dla bardzo dużych zestawów danych, które wymagają pamięci zewnętrznych.

4. Zanim przejdziemy do precyzyjnej definicji algorytmu sortowania przypomnimy pojęcia zbioru uporządkowanego i zbioru liniowo uporządkowanego .

System relacyjny to para uporządkowana (X,R), gdzie X jest niepustym zbiorem a R dowolną ustaloną relacją w zbiorze X.

System relacyjny $(X,\leq)$ z relacją dwuargumentową $\leq$ nosi nazwę zbioru quasi uporządkowanego jeśli relacja $\leq$ jest zwrotna i przechodnia tzn. dla każdego $x \in X$ mamy $x \leq x$ (zwrotność) oraz dla każdego $x,y,z \in X$ $(x \leq y \text{ i } y \leq z) \Rightarrow z \leq z$ (przechodniość).

System relacyjny $(X,\leq)$ z relacją dwuargumentową $\leq$ nosi nazwę zbioru uporządkowanego (lub częściowo uporządkowanego) jeśli relacja $\leq$ jest zwrotna, przechodnia i dodatkowo spełnia warunek dla każdego $x,y \in X$ $(x \leq y \text{ i } y \leq x) \Rightarrow x = y$ (antysymetryczność). Relację $\leq$ nazywamy w tej sytuacji porządkiem (lub porządkiem częściowym).

System relacyjny $(X,\leq)$ z relacją dwuargumentową $\leq$ nosi nazwę zbioru liniowo uporządkowanego jeśli $(X,\leq)$ jest zbiorem uporządkowanym i ponadto dla każdego $x,y \in X$ spełniony jest tzw. warunek spójności

$$x \leq y \quad \text{lub} \quad y \leq x .$$

Występująca w definicji zbioru liniowo uporządkowanego relacja $\leq$ nosi nazwę relacji liniowego porządku. Oczywiście każdy zbiór liniowo uporządkowany jest uporządkowany.

Rozważając zbiory uporządkowane wygodnie jest operować obok relacji porządkującej $\leq$ relację mniejszości $<$. Relację mniejszości definiujemy następująco: dla każdego $x,y \in X$

$x < y$ wtedy i tylko wtedy $x \le y$ i $y \neq x$

Przykład 1. Para uporządkowana $(N, \le)$, (gdzie $\le$ jest zwykłą definiowaną w arytmetyce relacją „mniejsze, równe") jest zbiorem liniowo uporządkowanym. Podobnie para uporządkowana $(Z, \le)$ jest zbiorem liniowo uporządkowanym.

■

Przykład 2. Para uporządkowana $(R, \le)$ (gdzie $\le$ jest zwykłą definiowaną w analizie relacją „mniejsze, równe") jest zbiorem liniowo uporządkowanym.

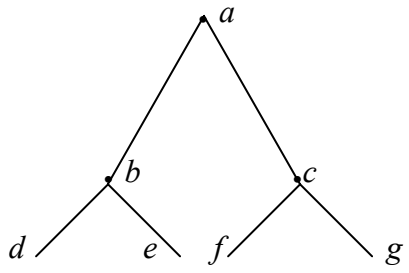
■

Przykład 3. Para uporządkowana $(2^X, \subseteq)$, (gdzie 2^X jest rodziną wszystkich podzbiorów niepustego zbioru X a $\subseteq$ relacją inkluzji) to częściowy porządek, ale nie porządek liniowy.

■

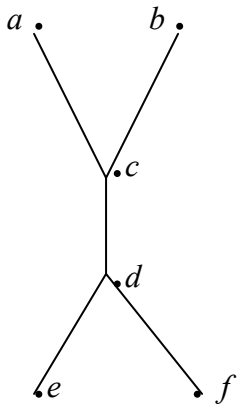
Jeśli zbiór X jest skończony to dowolną relację $\le$ porządkującą ten zbiór można opisać za pomocą grafu.

Przykład 4. Relacja wprowadzona za pomocą grafu z rys. 3.1 w zbiorze wierzchołków $\{a, b, c, d, e, f, g\}$ definiuje relację porządku.



Rys.3.1. Relacja porządku wprowadzona grafem w zbiorze wierzchołków $\{a, b, c, d, e, f, g\}$. Element położony wyżej poprzedza element położony niżej.

Przykład 5. Relacja wprowadzona za pomocą grafu z rys. 3.2 w zbiorze wierzchołków $\{a,b,c,d,e,f\}$ definiuje relację porządku.



Rys. 3.2. Relacja porządku wprowadzona grafem w zbiorze wierzchołków $\{a,b,c,d,e,f\}$. Element położony wyżej poprzedza element położony niżej.

5. Ważna w algorytmach sortowania jest tzw. relacja uporządkowania leksykograficznego lub jak mówimy krótko porządek leksykograficzny.

Niech będzie danych *n* zbiorów liniowo uporządkowanych $(A_1 \leq_1),(A_2 \leq_2),...,(A_n \leq_n)$.
W produkcie $A_1 \times A_2 \times ... \times A_n$ możemy wprowadzić relację liniowego porządku wzorem:
dla $x = (x_1, x_2, ..., x_n), y = (y_1, y_2, ..., y_n)$, gdzie $x, y \in A_1 \times A_2 \times ... \times A_n$
$(x \leq y)$ wtedy i tylko wtedy, gdy
1. $x_1 < y_1$ lub
2. $x_1 = y_1$ i $x_2 < y_2$ lub
3. $x_1 = y_1$ i $x_2 = y_2$ i $x_3 < y_3$ lub

.
.
.

n. $x_1 = y_1, x_2 = y_2, ..., x_{n-1} = y_{n-1}, x_n < y_n$

Tak zdefiniowana relacja $\leq$ w zbiorze $A_1 \times A_2 \times ... \times A_n$ jest jak łatwo sprawdzić relacją liniowego porządku i nazywa się uporządkowaniem leksykograficznym lub porządkiem leksykograficznym.

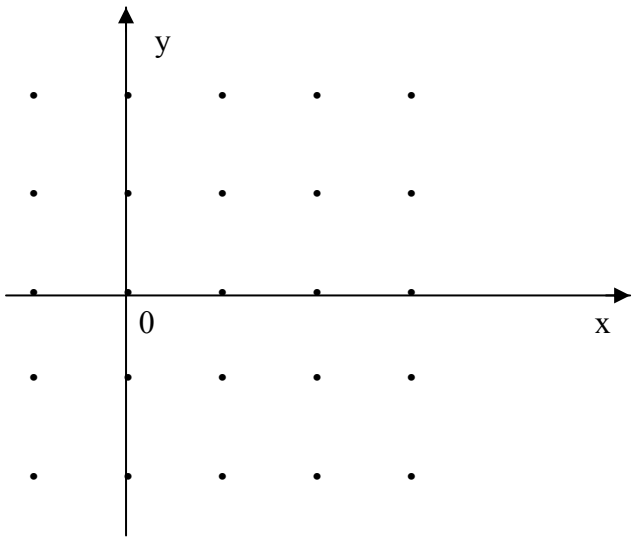
Uwaga. W analogiczny sposób wprowadzamy relację porządku leksykograficznego dla produktu $\overset{\infty}{\underset{k=1}{P}} A_k$, gdzie $(A_k, \leq_k)$ dla każdego $k \in N$ jest zbiorem liniowo uporządkowanym.

Przykład 1. W produkcie kartezjańskim N^k możemy wprowadzić porządek leksykograficzny $\leq$ przyjmując za punkt wyjścia porządek liniowy w N z przykładu 1. Podobnie w zbiorze liczb zespolonych

Przykład 2. Tak porządkujemy liczby z $N \cup \{0\}$ w zapisach *NKW* (naturalny kod wagowy) np.: $529 \leq 629$ bo $5 \leq 6$, $525 \leq 527$ bo $5 \leq 7$.

Przykład 3. Tak porządkujemy nazwiska na liście np.: $abb \leq abc$.

Przykład 4. Uporządkowanie leksykograficzne w $Z \times Z$ rys. 3.3. By porównać 2 punkty porównujemy najpierw pierwsze współrzędne a następnie drugie.



Rys. 3.3. Uporządkowanie leksykograficzne w $Z \times Z$

6. Definicja permutacji. Permutacja to przekształcenie różnowartościowe i „na” czyli bijekcja pewnego niepustego zbioru X (na ogół zakłada się, że X jest skończonym zbiorem n elementowym). Wygodnie jest ponumerować elementy zbioru X liczbami 1,2,...,n i wówczas permutację zbioru X można utożsamiać z permutacją zbioru {1,2,...,n}.

Zbiór wszystkich permutacji ustalonego niepustego zbioru stanowi grupę wraz z działaniem superpozycji odwzorowań.

Definicja sortowania. Niech zbiór A będzie liniowo uporządkowany relacją $\leq$ tzn. niech $(A,\leq)$ będzie zbiorem liniowo uporządkowanym i niech będzie dany ciąg skończony $(a_1,a_2,...a_n)$ o wartościach w zbiorze A. Niech $\pi :< 1,n >\to< 1,n >$ będzie taką permutacją zbioru $< 1,n >$, że

$$a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)} \quad (*)$$

Uzyskanie ciągu $(a_{\pi(1)},a_{\pi(2)},...,a_{\pi(n)})$ z ciągu $(a_1,a_2,...a_n)$ nazywamy **sortowaniem**. Wynikiem sortowania jest więc ciąg $(a_{\pi(1)},a_{\pi(2)},...,a_{\pi(n)})$ spełniający warunek (*).

Sortowanie ciągu skończonego $(a_1,a_2,...a_n)$ lub jak mówimy inaczej sortowanie listy $q=(a_1,a_2,...a_n)$ to inaczej porządkowanie, przestawianie wyrazów ciągu wejściowego tak, by uzyskać ciąg skończony $(a_{\pi(1)},a_{\pi(2)},...,a_{\pi(n)})$ spełniający warunek (*). Problem sortowania ciągu jest oczywiście rozwiązany jeśli znajdziemy permutację $\pi :< 1,n >\to< 1,n >$ dającą (*).

Algorytm sortowania zapisujemy więc następująco

Algorytm Sortowanie wg pewnego algorytmu

Dane wejściowe: ciąg skończony $(a_1,a_2,...a_n)$ o wartościach w zbiorze A.

Dane wyjściowe: ciąg skończony $(a_{\pi(1)},a_{\pi(2)},...,a_{\pi(n)})$ o wartościach w zbiorze A spełniający warunek $a_{\pi(1)} \leq a_{\pi(2)} \leq ... \leq a_{\pi(n)}$.

Zapis algorytmu sortowania

■

Przykład. Jeśli dane wejściowe algorytmu sortowania stanowi ciąg 8,2,4,9,3,6 to dane wyjściowe stanowi ciąg 2,3,4,6,8,9. Ogólnie rzecz biorąc elementy sortowanego ciągu nie muszą być liczbami naturalnymi ale najczęściej tak jest.
■

Uwaga 1. Algorytmy sortowania są bardzo ważne w praktyce, ponieważ często korzystamy ze zbiorów uporządkowanych. Lubimy np.: by listy studenckie z nazwiskami były drukowane w kolejności alfabetycznej. Podobnie hasła w encyklopedii czy w słowniku uporządkowane

są zgodnie z porządkiem leksykograficznym. Uporządkowanie listy na ogół znakomicie ułatwia wyszukiwanie elementów listy o zadanych cechach.

Uwaga 2. Można by się zastanawiać skąd wzięła się nazwa „sortowanie". Wydaje się, że naturalniejszą nazwą w języku polskim byłoby „porządkowanie". Z drugiej strony sortowanie to bezpośrednie tłumaczenie terminu angielskiego „sorting".

Sortowanie w szczególności możemy zdefiniować dla ciągu skończonego rekordów $(a_1, a_2, ...a_n)$. Taki zbiór nie musi mieć w naturalny sposób zdefiniowanego liniowo porządku, ale możemy postąpić tak. Porządek liniowy w zbiorze A wprowadzamy np.: za pomocą funkcji klucza $key : A \to N$. Sortowanie ciągu skończonego rekordów $(a_1, a_2, ...a_n)$ polega w tej sytuacji na znalezieniu takiej permutacji $\pi : \langle 1, n \rangle \to \langle 1, n \rangle$, by $key(a_{\pi(1)}) \le key(a_{\pi(2)}) \le ... \le key(a_{\pi(n)})$ co oczywiście można też zapisać jako: $a_{\pi(1)} \le a_{\pi(2)} \le ... \le a_{\pi(n)}$.

Możemy jeszcze inaczej spojrzeć na porządkowanie rekordów np. niech $a_i = (k_i, p_i)$ dla każdego $i \in < 1, n >$, gdzie k_i jest kluczem a p_i tzw. dowiązaniem, wskaźnikiem do rekordu. Sortując ciąg $(a_1, a_2, ...a_n)$ w gruncie rzeczy porządkujemy dowiązania unikając przestawiania być może długich rekordów.

Uwaga. Lista do posortowania może znajdować się w pamięci wewnętrznej albo zewnętrznej, stąd podział na algorytmy sortowania : wewnętrzne i zewnętrzne.

Istota rzeczy w sortowaniu wewnętrznym: elementy do posortowania pamiętamy w macierzy jednowymiarowej *A* i mamy swobodny dostęp do każdego elementu macierzy.

W sortowaniu zewnętrznym elementy ciągu $(a_1, a_2, ...a_n)$ wczytywane są z pliku , który z natury rzeczy ma organizacje sekwencyjną.

Operacja dominująca w algorytmach sortowania. Sprawdzanie czy dla danych $x, y \in A$ zachodzi relacja $x \le y$ nazywamy porównaniem. Za operację dominującą uważamy w algorytmach sortowania porównanie dwóch elementów ciągu. Złożoność czasową algorytmu, będziemy więc utożsamiać z ilością porównań wymaganą do uporządkowania listy wejściowej.

Złożoność pamięciowa (czyli przestrzenna) to będzie dodatkowa niezbędna do realizacji algorytmu liczba miejsc w pamięci na pamiętanie elementów sortowanej listy (dodatkowa w stosunku do *n* miejsc na elementy ciągu).

Będziemy zakładać, że elementy listy są liczbami całkowitymi i że znajdują się w jednowymiarowej *n* elementowej tablicy $A[1..n]$, przy czym $A[i] = a_i$ dla każdego $i \in \langle 1, n \rangle$.

Czasem przyjmujemy $A[0] = -\infty$, $A[n+1] = +\infty$, gdzie $-\infty$ oznacza w praktyce najmniejszą z możliwych liczb w danej reprezentacji, a $+\infty$ największą.

Przy analizie probabilistycznej tzn. obliczaniu średniej złożoności obliczeniowej $A(n)$ oraz standardowego odchylenia $\sigma(n)$ zakładamy na ogół, że danymi wejściowymi są permutacje zbioru $<1, n>$ oraz, że każda taka permutacja jest jednakowo prawdopodobna.

Uwaga. Czasami przy sortowaniu wymaga się zachowania tzw. warunku stabilności tzn. zachowania początkowego ustawienia względem siebie elementów równych (np.: rekordów o takich samych kluczach). Na przykład mamy alfabetyczna listę studentów: sortujemy ją względem uzyskanych ocen – z reguły wymagamy wtedy, by nazwiska studentów (w przypadku tej samej oceny) były podawane w kolejności alfabetycznej.

3.2 Sortowanie bąbelkowe (ang. bubblesort)

Sortowanie bąbelkowe nazywa się też czasem sortowaniem pęcherzykowym lub algorytmem *bubblesort.* Jest to chyba najprostszy z algorytmów sortowania jeśli chodzi o pomysł i zwartość zapisu. Równie prosta jest analiza złożoności obliczeniowej tego algorytmu.

Algorytm Sortowanie bąbelkowe (bubblesort)

Dane WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

Dane WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *bubblesort* (*A*,*n*)

begin

for $j = n$-1 **step** –1 **until** 1 **do**
for $i = 1$ **step** 1 **until** j **do**
if $A[i+1] < A[i]$ **then** zamień miejscami $A[i]$ z $A[i+1]$
end

Jest to algorytm z kwadratową złożonością obliczeniową. Niezależnie od danych algorytm do posortowania tablicy A używa jak łatwo obliczyć $\frac{n(n+1)}{2}$ porównań. Sortowanie bąbelkowe jest więc algorytmem klasy $\Theta(n^2)$.

Poprawność algorytmu jest dosyć oczywista. Pierwszy obieg pętli wewnętrznej dla $j = n$-1 umieszcza element największy w elemencie $A[n]$ tablicy. Drugi obieg pętli wewnętrznej dla $j = n$-2 umieszcza element drugi co do wielkości w elemencie $A[n-1]$ tablicy itd.

Przemieszczanie się elementu największego (i kolejnych) w prawo można porównać do ruchu w górę pęcherzyków powietrza w wodzie. Stąd nazwa algorytmu.

3.3 Zmodyfikowane sortowanie bąbelkowe (ang. modified bubblesort)

Wygodnie jest zmodyfikowany algorytm sortowania bąbelkowego sformułować oddzielnie dla n parzystego i n nieparzystego. W obu przypadkach mamy $T_w(n) = \Theta(n^2)$, $T_{ave}(n) = \Theta(n^2)$.

Dla n parzystego algorytm jest następujący.

Algorytm: Zmodyfikowane sortowanie bąbelkowe (modified bubblesort)

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$, n parzyste, $n \geq 2$
DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \leq A[2] = a_{\pi(2)} \leq ... \leq A[n] = a_{\pi(n)}$

---

procedure *modified bubblesort* (*A*,*n*)

begin

for $i = 1$ **until** $n - 1$ **do**

begin

if i nieparzyste **then**

for $j := 1$ **step** 2 **until** n-1 **do**

if $A[j+1] < A[j]$ **then** zamień miejscami $A[j]$ z $A[j+1]$;

if i parzyste **then**

for $j := 2$ **step** 2 **until** n-2 **do**

if $A[j+1] < A[j]$ **then** zamień miejscami $A[j]$ z $A[j+1]$

end

end

■

Dla n parzystego ilość dokonywanych w algorytmie porównań jest zawsze równa:

$$\underbrace{\frac{n}{2} + (\frac{n}{2} - 1) + \frac{n}{2} + (\frac{n}{2} - 1) + ... \frac{n}{2} + (\frac{n}{2} - 1) + \frac{n}{2}}_{n-1} = \Theta(n^2)$$

Dla *n* nieparzystego algorytm można zapisać tak:

Algorytm: Zmodyfikowane sortowanie bąbelkowe (modified bubblesort)

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., \ A[n] = a_n$, n nieparzyste, $n \geq 2$

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \leq A[2] = a_{\pi(2)} \leq ... \leq A[n] = a_{\pi(n)}$

---

procedure *modified bubblesort* (*A*,*n*)

begin

for $i = 1$ **until** $n - 1$ **do**

begin

if *i* nieparzyste **then**

for $j := 1$ **step** 2 **until** *n*-2 do

if $A[j+1] < A[j]$ **then** zamień miejscami $A[j]$ z $A[j+1]$

if *i* parzyste **then**

for $j := 2$ **step** 2 **until** *n*-1 **do**

if $A[j+1] < A[j]$ **then** zamień miejscami $A[j]$ z $A[j+1]$

end

end *modified bubblesort*;

Dla *n* nieparzystego ilość dokonywanych w algorytmie porównań jest zawsze równa

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

Ponieważ ilość porównań nie zależy od danych zatem pesymistyczna i średnia czasowa złożoność obliczeniowa są sobie równe.

Ostatecznie stwierdzamy, że czasowa pesymistyczna i średnia złożoność obliczeniowa algorytmu modified bubblesort są kwadratowe czyli rzędu $\Theta(n^2)$.

Zmodyfikowany algorytm sortowania bąbelkowego ma szybką realizację hardware'ową w postaci systolicznej sieci sortującej. Wydaje się, że dla niezbyt dużych n jest to najlepiej nadający się do implementacji sprzętowej w postaci układu scalonego algorytm.

Dowód poprawność i algorytmu zmodyfikowanego sortowania bąbelkowego jest nieco skomplikowany. Można go znaleźć np. w pracy [7].

Algorytm modified bubblesort można też zapisać jednolicie bez rozbijania zapisu na wersję z n parzystym i z n nieparzystym.

Algorytm: Zmodyfikowane sortowanie bąbelkowe (modified bubblesort)

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., \ A[n] = a_n$,

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *modified bubblesort* (*A*,*n*)

begin

if *n*=1 **then goto** koniec;

*k*1:=*n*-2; *k*2:=*n*-1;

if *n* nieparzyste **begin** *k*3:=*k*1; *k*1:=*k*2; *k*2:=*k*3 **end;**

for $i = 1$ **until** $n - 1$ **do**

begin

if i nieparzyste **then**

for $j := 1$ **step** 2 **until** $k1$ **do**

if $A[j+1] < A[j]$ **then** zamień miejscami $A[j]$ z $A[j+1]$

if i parzyste **then**

for $j := 2$ **step** 2 **until** $k1$ **do**

if $A[j+1] < A[j]$ **then** zamień miejscami $A[j]$ z $A[j+1]$

end

koniec:

end *modified bubblesort*;

3.4 Insertionsort – sortowanie przez wstawianie

Wyobraźmy sobie, że mamy *n* elementowy ciąg sortowany $A[1], A[2],..., A[n]$ podzielony na 2 podciągi, na dwa fragmenty. Pierwszy podciąg jest już uporządkowany a drugi jeszcze nie.

$\underbrace{A[1] \le A[2] \le ... \le A[i-1]}$ oraz $\underbrace{A[i],..., A[n]}$

uporządkowana część listy ta druga część

gdzie $i \in < 2, n >$.

Sortowanie przez wstawianie polega na wstawianiu pierwszego z brzegu elementu $A[i]$ w uporządkowaną część listy $A[1] \le A[2] \le ... \le A[i-1]$ (z ewentualnym „rozepchnięciem" i przenumerowaniem) tak by ciąg wynikowy $A[1], A[2],..., A[i]$ był uporządkowany tzn. by było $A[1] \le A[2] \le ... \le A[i-1] \le A[i]$.

Jeśli początkowo uporządkowana część listy składała się z jednego elementu $A[1]$ to stosując tę technikę dla każdego $i = 2,3,...,n$ dostajemy ostatecznie uporządkowany ciąg $A[1], A[2],..., A[n]$.

Zapis algorytmu w pseudokodzie jest następujący.

Algorytm: *insertionsort* – sortowanie przez wstawianie

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2,..., A[n] = a_n$
DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *insertionsort*(*A*,*n*);

$A[0] := -\infty$
var i, j, key : integer;
begin

for $i := 2$ **to** n **do** **begin** $j := i$; $key := A[i]$;
{wstawianie wyrazu $A[i]$ w posortowany już ciąg $A[1..(i-1)]$ tak by znalazł się na właściwym miejscu }

while $A[j-1] > key$ **do begin**
$A[j] := A[j-1]$;
$j := j-1$ {rozepchnięcie i cofnięcie indeksu}
end;
$A[j] := key$
end

end *insertionsort*;

Uwaga. Pętla **while** $A[j-1] > key$ **do** może być zastąpiona pętlą **while** $(j > 1$ and $A[j-1] > key)$ do zbędne jest wtedy stosowanie wartownika $A[0] := -\infty$.

Zastosowana w algorytmie *insertionsort* zasada jest naturalna i analogiczna do zasady stosowanej na ogół do porządkowania kart do gry, gdy po kolei dostajemy karty podczas rozdania. Otrzymaną kartę „wpasowujemy pomiędzy" trzymane w ręce, uporządkowane już karty tak by zachować porządek kart.

Poprawność algorytmu jest oczywista. Zewnętrzna pętla **for** dla $i = 2,3,...,n$ „wybiera" z nieuporządkowanego podciągu $a[i..n]$ kolejne elementy $a[i]$. Pętla **while** dokonuje porównań elementu $key := a[i]$ z kolejnymi elementami uporządkowanego podciągu $A[0..(i-1)]$ wstawiając porównywany element w odpowiednie miejsce. Pętla **while** „przestaje działać" jeśli właściwe położenie wartośći $a[i]$ zostanie zlokalizowane.

Przykład. Mamy 6 elementowy ciąg 8,2,4,9,3,6, . Algorytm sortowania przez wstawianie działa następująco

1 krok mamy ciąg 8, 2, 4, 9, 3, 6 przestawiamy 2 przed 8 i dostajemy 2, 8, 4, 9, 3, 6

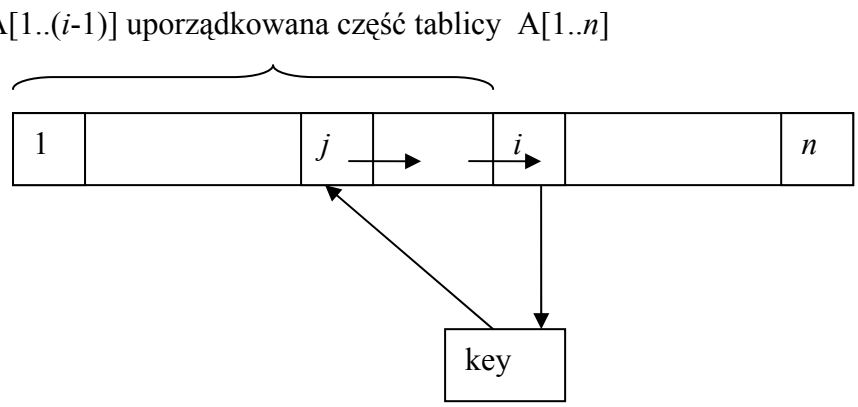
2 krok mamy ciąg 2, 8, 4, 9, 3, 6 przestawiamy 4 przed 8 i dostajemy 2, 4, 8, 9, 3, 6

3 krok mamy ciąg 2, 4, 8, 9, 3, 6 nie dokonujemy przestawienia więc mamy 2, 4, 8, 9, 3, 6

4 krok mamy ciąg 2, 4, 8, 9, 3, 6 przestawiamy 3 przed 4 i dostajemy 2, 3, 4, 8, 9, 6

5 krok mamy ciąg 2, 3, 4, 8, 9, 6 przestawiamy 6 przed 8 i dostajemy 2, 3, 4, 6, 8, 9

Ciąg 2, 3, 4, 6, 8, 9 jest już posortowany. Koniec ■



Rys. 3.4 Ilustracja działania algorytmu *insertionsort*

Pesymistyczną złożoność czasową powyższego algorytmu łatwo jest znaleźć. Mamy bowiem (n-1) obiegów pętli zewnętrznej for dla $i = 2,3,...,n$. Dla obiegu pętli o numerze i mamy w najgorszym przypadku i porównań (licząc porównanie z elementem $A[0] := -\infty$).

Najgorszym przypadkiem będzie oczywiście sytuacja, gdy wstępnie mamy

$$A[1] > A[2] > ... > A[n]$$

W tej sytuacji pesymistyczna złożoność czasowa $T_w(n)$ jest sumą postępu arytmetycznego i wynosi:

$$T_w(n) = 2 + 3 + 4 + ... + n = \frac{n(n+1)}{2} - 1 = \frac{1}{2}n^2 + \frac{1}{2}n - 1 = \Theta(n^2)$$

Liczymy tu również porównania z elementem $A[0]$. Jeśli ciąg jest posortowany mamy najlepszy przypadek. Dokonujemy wówczas tylko n-1 porównań zatem

$$\Delta(n) = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{1}{2}n^2 - O(n) = O(n^2)$$

Żeby obliczyć średnią złożoność czasową algorytmu musimy jak zawsze uczynić pewne założenia o zbiorze danych wejściowych D_n (tzn. ustalić jakie są elementy D_n) i ustalić pewien rozkład prawdopodobieństwa na tym zbiorze czyli musimy przyjąć pewien model probabilistyczny danych wejściowych. Przez $G_1, G_2, ..., G_n$ będziemy oznaczać w dalszy ciągu zmienne losowe generujące dane wejściowe. Zakładamy, że zmienne losowe $G_1, G_2, ..., G_n$ określone są na pewnej przestrzeni probabilistycznej $(\Omega, \mathfrak{M}, P)$.

Rozważmy teraz sytuację taką, że dane wejściowe algorytmu *insertsort* są n elementowymi wariacjami (bez powtórzeń) ze zbioru m elementowego $m \geq n$ oraz wystąpienie każdej takiej wariacji jest jednakowo prawdopodobne. Zachodzi następujący fakt

Fakt. Niech $m, n \in N$ i $m \geq n$. Rozważmy przestrzeń probabilistyczną $(\Omega, \mathfrak{M}, P)$, taką, że zdarzeniem elementarnym $\omega \in \Omega$ jest n elementowa wariacja ze zbioru m elementowego $<1, m>$ czyli $\omega = (a_1, a_2, ..., a_n)$ oraz $\mathfrak{M} = 2^{\Omega}$. Załóżmy, że rozkład prawdopodobieństwa P na Ω jest równomierny tzn. każda wariancja $\omega = (a_1, a_2, ..., a_n)$ jest jednakowo prawdopodobna. Zdefiniujmy zmienne losowe $G_1, G_2, ..., G_n$ na Ω wzorem $G_i(\omega) = a_i$ wówczas prawdopodobieństwo tego, że $G_i(\omega)$ zajmuje pozycję $k \in <1, n>$ po posortowaniu ciągu $G_1(\omega), G_2(\omega), ..., G_n(\omega)$ jest równe $\frac{1}{n}$.

Dowód. Zbiór Ω wszystkich zdarzeń elementarnych czyli wariacji *n* elementowych (bez powtórzeń) można rozbić na $\binom{m}{n}$ parami rozłącznych podzbiorów $\Omega_t \subset \Omega$, gdzie $t \in <1, \binom{n}{m}>$, przy czym każdy zbiór $\Omega_t \subset \Omega$ jest złożony ze wszystkich *n*! wariacji bez powtórzeń pewnego *n* elementowego podzbioru zbioru $B_t \subset <1, m>$.

Wybierzmy sobie pewien dowolny zbiór $B_t \subset <1, m>$. Któryś z elementów zbioru B_t zajmuje po posortowaniu pozycję k. Oznaczmy ten element przez k'. Dla każdego $\omega \in \Omega_t$ mamy więc

$G_i(\omega) = k'$ wtedy i tylko wtedy $G_i(\omega)$ po posortowaniu zajmuje pozycję *k*

Ilość zdarzeń elementarnych ze zbioru Ω_t takich, że $G_i(\omega) = k'$ jest równa $(n-1)!$ z kolei zbiór Ω_t liczy $n!$ elementów zatem dla $\frac{(n-1)!}{n!} = \frac{1}{n}$ wszystkich elementów $\omega \in \Omega_t$ wartość $G_i(\omega)$ przyjmuje pozycję *k*-tą po posortowaniu.

Ponieważ analogiczne rozumowanie możemy powtórzyć dla każdego $t \in <1, \binom{n}{m}>$ (dla każdego podzbioru B_t i odpowiadającego mu zbioru Ω_t) więc w sumie dla $\frac{1}{n}$ zdarzeń elementarnych $\omega \in \Omega$ wartość $G_i(\omega)$ zajmuje pozycję k po posortowaniu ciągu $G_1(\omega), G_2(\omega), ..., G_n(\omega)$.

Ponieważ z założenia rozkład *P* jest rozkładem równomiernym mamy ostatecznie

$$P(G_i = k) = \frac{1}{n}$$

■

Obliczymy teraz średnią czasową złożoność obliczeniową algorytmu *insertionsort* dla *i* tego etapu obliczeń tzn.

Jeśli mamy realizację losowego wektora *n*-wymiarowych danych wejściowych $G_1(\omega), G_2(\omega), ..., G_n(\omega)$, gdzie $n \geq 2$ to możemy zdefiniować $n-1$ zmiennych losowych dyskretnych $Y_2, Y_3, ..., Y_n$ podających dla każdego zdarzenia elementarnego ω liczbę porównań w kolejnych obiegach pętli (dla indeksu $i = 2,3,...n$). Oczywiście mamy:

$$T_{ave}(n) = E(Y_1 + Y_2 + ... + Y_n) = \sum_{j=2}^{n} E(Y_j)$$

Pozostaje więc obliczyć $E(Y_j)$ dla $j = 2,3,...n$. Korzystając z udowodnionego faktu stwierdzamy, że jeśli weźmiemy również pod uwagę porównanie z elementem $A[0]$ czyli wartownikiem to:

$$E(Y_j) = \frac{1+2+...j}{j} = \frac{j+1}{2}$$

Zatem

$$T_{ave}(n) = \sum_{j=2}^{n} E(Y_j) = \frac{1}{2}(3+4+...+n) = \frac{1}{2}(\frac{n(n+1)}{2} - 3) = \frac{n^2}{4} + \frac{n}{4} - \frac{3}{2} = \Theta(n^2)$$

Średnia czasowa złożoność obliczeniowa dla algorytmu *insertionsort* jest więc tego samego rzędu jak złożoność czasowa w najgorszym przypadku i wynosi

$$T_{ave}(n) = E(T(n)) = \Theta(n^2)$$

Uwaga. Wartość średnia $T_{ave}(n)$ zależy od modelu probabilistycznego dla danych wejściowych. Dla przypadku, gdy modelem danych wejściowych jest ciąg niezależnych zmiennych losowych o tym samym rozkładzie $G_1, G_2, ..., G_n$ i zakładamy, że każda zmienna losowa $G_1, G_2, ..., G_n$ ma rozkład równomierny na $< 1, m >$ uzyskujemy nieco inne $T_{ave}(n)$.

Wykorzystując metodę funkcji tworzących można obliczyć wariancję złożoności czasowej $D^2(T(n))$. Wariancja zmiennej losowej $T(n)$ jest równa

$$D^2(T(n)) = n(n-1)(2n+5)/72 = \Theta(n^3)$$

a średnie odchylenie standardowe

$$\sqrt{D^2(T(n))} = \sqrt{n(n-1)(2n+5)/72} = \frac{1}{6}n^{3/2} + \Theta(n) = \Theta(n^{3/2})$$

Zalety algorytmu *insertionsort*: algorytm jest prosty, łatwy w implementacji programowej i sprzętowej oraz stabilny.

Wyniki testowania algorytmu *insertionsort* wykazują, że algorytm ten jest bardzo szybki w porównaniu z innymi algorytmami sortowania dla małych rozmiarów danych wejściowych gdy $n \leq 20$.

Algorytmu *insertionsort* działa w miejscu (ang. in place) tzn. dodatkowa pamięć potrzebna do realizacji algorytmu jest stała i nie zależy od rozmiaru danych wejściowych *n*.

Algorytm sortowania przez wstawianie w porównaniu z innymi algorytmami sortowania nie jest algorytmem szybkim łatwo można jednak zbudować bardzo szybki sortujący układ systoliczny wykorzystujący ten algorytm (por. podrozdział o sieciach sortujących)

3. 5 Sortowanie przez selekcję (ang. selectionsort)

Sortowanie przez selekcję nazywa się inaczej sortowaniem przez wybieranie (ang. selectionsort). Sortowanie przez selekcję polega na tym, że:

1. W tablicy sortowanej $A[1:n]$ stanowiącej dane wejściowe wybieramy indeks *min* elementu o najmniejszej wartości i zamieniamy miejscami element $A[\min]$ z elementem na pierwszej pozycji tzn. z elementem $A[1]$.

Element $A[1]$ naszej tablicy zawiera teraz najmniejszy element sortowanego ciągu.

2. Z następnych *n*-1 elementów $A[2], A[3], \ldots, A[n]$ wybieramy ponownie najmniejszy element i zamieniamy z elementem na drugiej pozycji itd.

3. Zatrzymujemy się, gdy osiągniemy dwa ostatnie elementy.

Algorytm: Selectionsort – sortowanie przez selekcję

DANE WEJŚCIOWE: ciąg sortowany $(a_1, a_2, \ldots a_n)$ umieszczony jest w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, \ldots, \; A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ umieszczony jest w tablicy $A[1:n]$ tzn. $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le \ldots \le A[n] = a_{\pi(n)}$

```
procedure selectionsort(A,n);
var i, j, min, key : integer;
begin
for i : =1 to n –1 do

        begin min :=i;
              for j := i + 1 to n do if A[ j] < A[min] then min := j ;
              {wybierając min pracujemy na indeksach}
              key := A[i]; A[i] := A[min]; A[min] := key ;
              {zamiana miejscami elementów A[i] i A[min] }
        end

end selectionsort;
```

Analiza złożoności algorytmu sortowania przez selekcję

Ilość wszystkich porównań (obiegi dwóch pętli) daje:

$$W(n) = A(n) = (n-1) + (n-2) + ... + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = \Theta(n^2)$$

a ponieważ liczba porównań nie zależy od danych wejściowych, dostajemy $\Delta n = \sigma(n) = 0$.

Ponadto dla przestrzennej złożoności obliczeniowej mamy $S(n) = 0(1)$. Mówimy w tej sytuacji, że algorytm pracuje w miejscu (ang. in place)

Zalety algorytmu:

1.Mamy tylko co najwyżej (n-1) przestawień
2.Prosta jest implementacja algorytmu
3.Dla małych n algorytm jest szybki

Uwaga. Można łatwo pokazać, że algorytm nie jest stabilny. Na przykład sortowanie pięcioelementowego ciągu (5,5,4,5,5) za pomocą algorytmu *selectionsort* spowoduje przestawienie elementu pierwszego i trzeciego a pozostałe elementy pozostaną na swoich miejscach. Można jednak algorytm *selectionsort* tak zmodyfikować by uczynić ten algorytm stabilnym.

3.6 Algorytm sortowania „mergesort” (algorytm sortowania przez scalanie)

Scalanie. Zadaniem podobnym do sortowania ale prostszym jest scalanie ciągów uporządkowanych. Problem scalania jest taki. Mamy 2 uporządkowane ciągi $a_1 \le a_2 \le ... \le a_n$ oraz $b_1 \le b_2 \le ... \le b_m$. Naszym celem jest scalenie ich w jeden ciąg uporządkowany tzn. utworzenie z wyrazów tych 2 ciągów jednego ciągu $c_1, c_2, ..., c_{m+n}$ takiego, że $c_1 \le c_2 \le ... \le c_{m+n}$. Operacja scalania ciągów znajduje jak zobaczymy zastosowanie w algorytmach sortowania wewnętrznego ale co warto w tym miejscu podkreślić jest zasadniczym chwytem stosowanym w algorytmach sortowania zewnętrznego.

Oczywiście do scalania ciągów uporządkowanych można użyć dowolnego algorytmu sortowania. Scalanie jest jednak znacznie prostszym zagadnieniem i daje się rozwiązać w czasie liniowym względem $n+m$. Dokładniej złożoność czasowa przedstawionego niżej algorytmu scalania jest równa:

$$T_w(n,m) = T_{ave}(n,m) = n + m = \Theta(n+m)$$

oraz $\Delta(n) = \delta(n) = 0$ i $S(n) = \Theta(n)$ (potrzebne jest miejsce na pomocnicze tablice $L[1...n1]$ i $R[1...n2]$).

Zasada działania algorytmu jest prosta. Najpierw porównujemy najmniejsze elementy obu uporządkowanych ciągów $a_1, a_2, ..., a_n$ (gdzie $a_1 \le a_2 \le ... \le a_n$) i $b_1, b_2, ..., b_m$ (gdzie $b_1 \le b_2 \le ... \le b_m$) tzn. porównujemy a_1 i b_1 oraz wybieramy mniejszy równy z tych elementów i podstawiamy pod c_1. Następnie usuwamy podstawiony pod c_1 element z ciągu do którego należał. Odpowiedni ciąg staje się w ten sposób krótszy. Teraz ponownie porównujemy dwa najmniejsze elementy w ciągach wejściowych, wybieramy mniejszy z nich i podstawiamy pod c_2 itd. aż do wyczerpania się elementów w jednym z ciągów. Widać więc, że scalając w ten sposób 2 ciągi $a_1, a_2, ..., a_n$ i $b_1, b_2, ..., b_m$ dokonamy w najgorszym przypadku $n+m-1$ porównań a w najlepszym $\min(n,m)$ porównań. Jeśli jednak uwzględnimy (występujące w poniżej przedstawionym algorytmie) porównania z wartownikami list $a_1, a_2, ..., a_n$ oraz $b_1, b_2, ..., b_m$ to ilość porównań jest stała i wynosi dokładnie $n+m$.

Poniżej przedstawiony jest algorytm scalania w wersji, która posłuży nam do opisu algorytmu sortowania przez scalanie.

Algorytm: Scalanie posortowanych ciągów

DANE WEJŚCIOWE: Tablica $A[1..d]$ oraz $p,q,r \in <1,d>$ indeksy tablicy $A[1..d]$ takie, że $p \le q < r \le d$. Zakładamy, że podtablice $A[p..q]$ i $A[q+1..r]$ tablicy $A[1..d]$ są już posortowane.

DANE WYJŚCIOWE: Posortowana podtablica $A[p..r]$ (zawierająca scalone podtablice $A[p..q]$ i $A[q+1...r]$)

---

procedure *merge*(A,*p*,*q*,*r*);

begin
$n1 := q - p + 1$; {obliczenie rozmiaru tablicy $A[p..q]$}
$n2 := r - q$; {obliczenie rozmiaru tablicy $A[p+1..r]$}

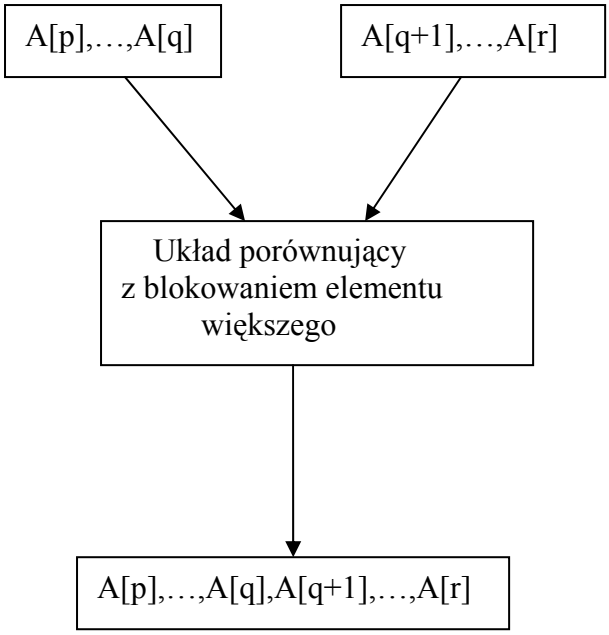
Tworzymy dwie tablice $L[1...n1+1]$ $R[1...n2+1]$

for $i := 1$ **to** $n1$ **do** $L[i] := A[p+i-1]$; {przepisanie podtablicy $A[p..q]$ do tablicy $L[1...n1]$}
for $j := 1$ **to** $n2$ **do** $R[i] := A[q+j]$;{przepisanie podtablicy $A[q+1..r]$ do tablicy $R[1...n2]$}
$L[i] := +\infty$; {ustawienie wartownika}
$R[i] := +\infty$; {ustawienie wartownika}
$i := 1$;
$j := 1$;

for $k := p$ **to** r **do if** $L[i] \le R[j]$ **then begin** $A[k]; = L[i]$; $i := i+1$ **end**
else begin $A[k]; = R[i]$; $j := j+1$ **end;**

end.

Powyższa procedura *merge* działa jak już powiedzieliśmy w czasie liniowym tzn. w oznaczeniach z opisu procedury w czasie $\Theta(n)$, gdzie $n = r - p + 1$ jest liczbą scalanych elementów.



Rys. 3.5 Procedura *merge* scala dwie uporządkowane podtablice $A[p..q]$ i $A[p+1..r]$ tworząc jedną podtablicę $A[p..r]$

Przykład. Scalanie dwóch posortowanych tablic $A[1]=4, A[2]=7, A[3]=13, A[4]=20$ oraz $B[1]=1, B[2]=9, B[3]=11, A[4]=12$

20	12
13	11
7	9
2	1

20	
13	12
7	11
2	9

20	12
13	11
7	9

	12
20	11
13	9

20	12
13	11

1 **1, 2** **1,2,7** **1,2,7,9** **1,2,7,9,11**

20	
13	12

20	
13	

20	

1,2,7,9,11,12 **1,2,7,9,11,12,13** **1,2,7,9,11,12,13,20**

Sortowanie przez scalanie. Scalanie można w oczywisty sposób wykorzystać do stworzenia algorytmu sortowania.

Algorytm sortowania przez scalanie jest typowym algorytmem zbudowanym według schematu „dziel i zwyciężaj".

Algorytm: mergesort – sortowanie przez scalanie

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *mergesort*(A,*n*); $\{ A[1..n] \}$

begin

if $n > 1$ **then**
begin
1. podziel tablicę na 2 części $A[1..\lfloor n/2 \rfloor]$ i $A[1..\lceil n/2 \rceil]$ i posortuj rekurencyjnie obie tablice
2. scal dwie posortowane tablice używając algorytmu scalania *merge*
end
end *mergesort*;

Ilość porównań w algorytmie *mergesort* a jednocześnie pesymistyczną i średnią złożoność czasową opisuje jak łatwo zauważyć równanie rekurencyjne:

$T(1) = 0$

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n$ dla $n>1$

Rozwiązaniem tego równania jest $T(n) = \Theta(n \log n)$. Jednocześnie $\Delta(n) = \delta(n) = 0$ i złożoność przestrzenna $S(n) = \Theta(n)$.

Wnioski. Algorytm *mergesort* działa w średnim czasie $\Theta(n \log n)$. Widać więc, że asymptotycznie *mergesort* jest szybszy od algorytmów *insertionsort* czy *bubblesort*, które działają w średnim czasie $\Theta(n^2)$.

W praktyce algorytm *mergesort* jest na ogół szybszy od algorytmu *insertionsort* dla $n > 30$.

3. 7 Algorytmy sortowania w czasie liniowym

Ogólnie rzecz biorąc algorytmy sortowania można podzielić na 2 grupy:
- algorytmy sortowania przez porównanie elementów ciągu sortowanego
- algorytmy sortowania czyniące użytek z samej wartości elementów ciągu sortowanego

Algorytmy sortowania omawiane dotychczas były algorytmami sortowania przez porównanie. tzn. w algorytmach sortowania omawianych do tej pory porównywaliśmy wyrazy ciągu $a_1, a_2, ..., a_n$ stanowiącego dane wejściowe i na tej podstawie dokonywaliśmy odpowiednich przestawień..

Można pokazać, metodą drzew decyzyjnych, że dolne ograniczenie na średnią złożoność czasową algorytmu sortowania przez porównanie jest rzędu $\Omega(n \lg n)$.

Teraz skorzystamy z faktu, że w komputerze liczby całkowite zapisujemy za pomocą słów binarnych. Do umieszczenia liczby na właściwym miejscu w ciągu posortowanym wykorzystamy jej wartość. Cały szereg algorytmów opartych na tej zasadzie daje liniowy czas sortowania. Po kolei zostaną omówione następujące algorytmy z czasem liniowym

1. Sortowanie przez zliczanie – countsort

2. Sortowanie pozycyjne – algorytm radixsort

3. Sortowanie kubełkowe - algorytm *bucketsort*

3.8 Sortowanie przez zliczanie – countsort

Zaczniemy od „metody liczników częstości" lub inaczej „metody sortowania przez zliczanie" . Algorytm sortowania przez zliczanie" nosi również nazwę algorytmu *countsort*.

Istota rzeczy. Niech $m = 2^b$, dla każdego $j = 0,1,...,,m-1$ zliczamy ile razy j pojawiło się w ciągu wejściowym liczb całkowitych $a_1, a_2, ..., a_n$; gdzie $a_i \in \leq 0, 2^b - 1 >$ czyli $0 \leq a_i \leq 2^b - 1 = m - 1$.

W ten sposób tworzymy dla każdego $j \in \leq 0, 2^b - 1 >$ (w pewnej tablicy *count*) licznik wystąpień liczby j w ciągu $a_1, a_2, ..., a_n$ i na podstawie znalezionych liczników wystąpień umieszczamy każdy element a_i we właściwym miejscu w ciągu wyjściowym.

Dokładniej, wyznaczamy dla każdego elementu a_i ciągu $a_1, a_2, ..., a_n$ liczbę elementów mniejszych od a_i. Mając tę liczbę potrafimy łatwo obliczyć, na której pozycji umieścić a_i.

Uwaga 1 Oczywiście, jeśli $0 \leq a_i \leq m-1$ dla pewnego $m \in N$, to nie musimy przyjmować $m = 2^b$, choć taka sytuacja jest zupełnie naturalna.

Uwaga 2 Pomysł sortowania przez zliczanie jest intuicyjnie jasny, tak właśnie sortujemy w wielu sytuacjach praktycznych.

Algorytm: ***countsort*** **(sortowanie przez zliczanie)**

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \leq A[2] = a_{\pi(2)} \leq ... \leq A[n] = a_{\pi(n)}$

---

procedure *countsort*(A,*n*); {Używamy tablic $A[1..n], t[1..n]$, count [0..*m*-1]}

var *i, j, p* : integer
begin
for *j* :=0 **to** *m*-1 **do** *count*[*j*] :=0;{zerowanie tablicy wystąpień}

for *i*:=1 **to** *n* **do** *count* [*A*[*i*]]:= *count* [*A*[*i*]]+1;
{licznik *count*[*i*] liczy liczbę wystąpień liczby *i* w ciągu wejściowym $a_1, a_2, ..., a_n$}
for *j* :=1 **to** *m*-1 **do** *count* [*j*] := *count* [*j*-1]+ *count* [*j*];
{*count* [*j*] to liczba wystąpień elementów $\leq j$, czyli jakby „dystrybuanta"}

for *i* := *n* **downto** 1 **do**
begin
p := *A*[*i*];
t [*count* [*p*]] := *p*;
count [*p*] : = count [*p*]-1
end;
{umieszczamy kolejne wartości $A[n], A[n-1], ..., A[1]$ na właściwych pozycjach w $t[1..n]$ }

for *i* :=1 **to** *n* **do** $A[i] := t[i]$ {przepisanie tablicy $t[1..n]$ do tablicy $A[1..n]$}
end *countsort*;

Analiza złożoności obliczeniowej algorytmu countsort

1.Złożoność czasową pesymistyczną liczymy bezpośrednio z liczby obiegów pętli. Otrzymujemy wówczas.

$$T_w(n,m) = \Theta(n+m)$$

Jeśli $m = O(n)$ to algorytm *countsort* działa w czasie $\Theta(n)$.

2.Ponieważ liczba obiegów dla każdych danych wejściowych o rozmiarze *n* jest taka sama więc:

$$T_{ave}(n,m) = T_w(n,m) = \Theta(n+m)$$

3.Zatem miara wrażliwości pesymistycznej $\Delta(n,m)=0$ i miara wrażliwości średniej, czyli oczekiwanej $\sigma(n,m)=0$

4.Złożoność przestrzenna jest równa $s(n,m)=n+m+O(1)$

Zalety algorytmu countsort:

Duża szybkość działania, gdy $m=O(n)$.
Algorytm jest stabilny (dzięki wypisywaniu elementów ciągu począwszy od $A[n]$ do $A[1]$).

Wady algorytmu countsort:

Dla dużych m algorytm ma spore wymagania pamięciowe (tablica *count* o rozmiarze m). Pamiętamy, że może być $m=2^b$

3. 9 Sortowanie pozycyjne – algorytm radixsort

Nazwą sortowania pozycyjnego obejmujemy kilka zbliżonych metod sortowania. Zasadniczym pomysłem jest w sortowaniu pozycyjnym wykorzystanie uporządkowania leksykograficznego elementów sortowanych.

W dalszym ciągu opiszemy dwie odmiany sortowania pozycyjnego będzie to sortowanie *radixsort* w połączeniu z algorytmem *countsort* i sortowanie *radixsort* w połączeniu z algorytmem *insertionsort.*

Sortowanie pozycyjne– algorytm *radixsort* wykorzystujący algorytm *countsort*

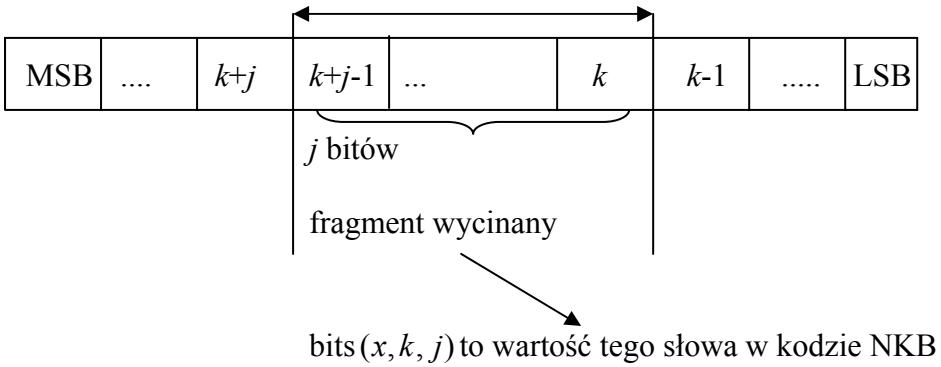
Wykorzystamy w dalszym ciągu następującą funkcję „wycinania” bitów w słowie binarnym:

function *bits* $(x,k,j:\text{integer}):\text{integer};$
begin
bits := (x div 2^k) mod 2^j
end *bits*;

gdzie x jest liczbą z $N\cup\{0\}$ reprezentowaną w zapisie NKB przez słowo b bitowe.

k – pierwszy bit z prawej strony „niewycięty”
j – długość słowa wycinanego
$k+j-1$ - ostatni bit z lewej strony „niewycięty”

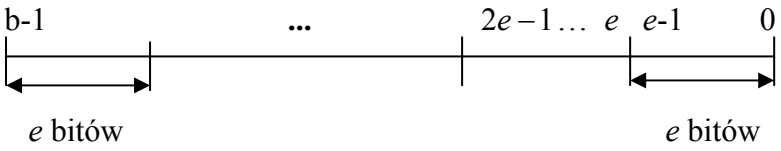
Przykład. *bits*(*x*,0,1) to LSB słowa binarnego reprezentującego liczbę *x* w kodzie NKB. *bits*(*x*,*b*-1,1) to MSB słowa binarnego reprezentującego liczbę *x* w kodzie NKB.



Rys. 3.6 Działanie funkcji bits.

Widać, że w algorytmie *countsort m* może być bardzo duże np.: typowe słowo maszynowe ma długość 32 bitów, zatem może być $m = 2^{32}$ i tablica *count* nie mieści się w typowej pamięci operacyjnej (choć warto w tym miejscu zauważyć, że przestrzeń adresowa Pentium 4 jest równa 64 GB).

Rozwiązaniem staje się w tej sytuacji zmniejszenie *m*, a ściślej dzielimy zbiór wartości $\langle 0, m-1 \rangle$ na części i stosujemy algorytm *countsort* do tych części.



Rys. 3.7 Słowo *b* bitowe dzielimy na $\frac{b}{e}$ grup *e* bitowych.

Opis pomysłu: Dokładniej postępujemy tak:

1.Najpierw sortujemy ciąg liczb $a_1, a_2, ..., a_n$ względem ostatniej, prawej (najmniej znaczącej) grupy bitów.

2.Następnie sortujemy ciąg liczb $a_1, a_2, ..., a_n$ względem przedostatniej grupy bitów itd. aż dojdziemy do najbardziej znaczącej grupy bitów.

Uwaga. Dla poprawności algorytmu *radixsort* jest istotne to, że algorytm *countsort* jest stabilny.

W algorytmie *radixsort* opisanym niżej wykorzystujemy uporządkowanie leksykograficzne jakie wprowadza naturalny zapis binarny NKB przy podziale słowa binarnego (reprezentującego sortowane elementy) na grupy e-bitów. O pozycji elementu decyduje najbardziej znacząca grupa *e*-bitów. Jeśli są one takie same, to znaczenie ma druga od lewej grupa bitów itd. aż do najmniej znaczącej grupy bitów. Jako pomocniczego algorytmu sortowania używamy w poniższym algorytmie algorytmu *countsort.*

Algorytm: ***radixsort*** **(sortowanie pozycyjne)**

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

```
procedure radixsort(A,n)   ;{m = 2^e, A[1..n], t[1..n], count [0..m-1],  e|b }

var i, j, pass, nofpasses, key : integer;

begin

nofpasses := b div e -1;

for pass := 0 to nofpasses do

        begin
                for j := 0 to m-1 do count [j] := 0;
                {zerowanie liczników, zerowanie tablicy zliczającej}

                for i := 1 to n do
                 begin
               key := bits (A[i]), pass * e,e) ;{"wycięcie" współrzędnej o numerze pass}

                count [key] := count [key]+1;  {zliczanie}

                end;
                 for j : = 1 to m-1 do count[j] : = count[j-1]+ count[j];

                                {tworzenie "dystrybuanty"}

                 for i := n downto 1 do
                                begin
                                        key : = bits(A[i], pass * e,e) ;
                                        t [count[key]] : = A[i];
                                        count[key] : = count [key]-1;
```

```
                    end
            {wpisywanie wszystkich A[i] na właściwe miejsce}

            for i :=1 to n do A[i] := t[i]  {wpisywanie do A}
        end

end radixsort;
```

Analiza złożoności:

$$T_w(n,m) = T_{ave}(n,m) = O(\frac{b}{e}(n+m))$$

$$\Delta(n,m) = \sigma(n,m) = 0$$

$$S(n,m) = n + m + O(1)$$

Uwagi końcowe. Nie używamy w algorytmie *radixsort* porównań a złożoność czasowa algorytmu jest liniowa. Złożoność przestrzenna może być jednak duża ponieważ w naturalnych sytuacjach *m* jest wysoką potęgą 2.

W praktyce okazuje się, że dla małych *n* algorytm jest stosunkowo wolny.

Gdy bity liczb sortowanych można uważać za realizację ciągu *b* doświadczeń Bernoulliego z $p = \frac{1}{2}$ to można algorytm *radixsort* znacznie przyspieszyć tak:

1. używamy *radixsort* dla najbardziej znaczących $\frac{b}{2}$ bitów (wstępne sortowanie)
2. wykonujemy insertionsort.

Uzasadnienie: Prawdopodobieństwo, w powyższej sytuacji, że dwie liczby będą miały takie same bity na połówkach wynosi $2^{-b/2}$ (jest to jednocześnie prawdopodobieństwo tego, że para jest nieposortowana). Zatem algorytm *insertionsort* jest stosowany do ciągu prawie uporządkowanego, a więc działa szybko!

W najprostszym ujęciu algorytm *radixsort* można zapisać dla dowolnego współpracującego z *radixsort* stabilnego algorytmu sortowania następująco:

Algorytm: *radixsort* (sortowanie pozycyjne)

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *radixsort*(*A*,*n*,*d*) ;

{*d* to liczba cyfr występujących w naturalnym zapisie pozycyjnym lub liczba współrzędnych w uporządkowaniu leksykograficznym}
begin

for *i* : =1 **to** *d* **do** "posortuj stabilnie tablicę *A*[1..n] według pozycji *i* używając dowolnego stabilnego algorytmu sortowania np. *insertionsort* lub *countsort*"
end *radixsort*;

3.10 Sortowanie kubełkowe - algorytm *bucketsort*

Algorytm sortowania kubełkowego nazywamy również algorytmem sortowania rozrzutowego lub algorytmem *bucketsort*.

Istota rzeczy. Zakładamy, że ciąg porządkowany $a_1, a_2, ..., a_n$ jest ciągiem o wartościach rzeczywistych a dokładniej o wartościach w przedziale $[0,1)$ tzn. dla każdego $i = 1,2,...,n$ mamy $a_i \in [0,1)$.

Dzielimy przedział $[0,1)$ na *n* podprzedziałów o jednakowej długości tzw. „kubełków".

$$I_0 = [0, \frac{1}{n}), \quad I_1 = [\frac{1}{n}, \frac{2}{n}), \quad I_2 = [\frac{2}{n}, \frac{3}{n}), \ldots, I_1 = [\frac{n-1}{n}, 1)$$

Załóżmy, że liczby $a_1, a_2, ..., a_n$ są realizacją wektora losowego $(X_1, X_2, ..., X_n)$, określonego na pewnej przestrzeni probabilistycznej $(\Omega, \mathfrak{M}, P)$ a zmienne losowe $X_1, X_2, ..., X_n$ są parami niezależne i mają rozkład jednostajny na odcinku $[0,1)$. W tej sytuacji prawdopodobieństwo $P(X_j \in I_i) = \frac{1}{n}$.

Z każdym przedziałem I_i wiążemy listę, którą oznaczamy symbolem $B[i]$. mamy więc tablicę list $B[0..(n-1)]$. W algorytmie kubełkowym przeglądamy po kolei elementy tablicy $A[1..n]$. Jeśli $A[i] \in I_j$ to $A[i]$ dołączamy do listy $B[j]$. W następnym kroku algorytmu sortujemy listy $B[0]$, $B[1]$, …, $B[n-1]$ np. algorytmem *insertionsort* W ostatnim kroku konkatenujemy listy $B[0]$, $B[1]$, …, $B[n-1]$ otrzymując posortowany ciąg wejściowy.

Zamiast algorytmu *insertionsort* można użyć dowolnego innego algorytmu sortowania. W analizie złożoności czasowej algorytmu będziemy jednak zakładać, że stosujemy do sortowania list $B[0], B[1], ..., B[n-1]$ algorytm *insertionsort*.

Algorytm: *bucketsort* (sortowanie kubełkowe lub sortowanie rozrzutowe)

DANE WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$
DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *bucketsort*(*A*,*n*);

begin $n := length(A)$;

for $i := 1$ **to** *n* **do** „wstaw A[i] na listę $B[\lfloor n \cdot A[i] \rfloor]$

for $i := 0$ **to** *n* **do** „posortuj listę $B[i]$ stosując insertionsort"

„scal listy $B[0], B[1], \ldots, B[n-1]$ z zachowaniem kolejności elementów wewnątrz list"
end *bucketsort*;

Przeanalizujmy teraz złożoność czasową naszego algorytmu. Jeśli n_i oznacza liczbę elementów umieszczonych w kubełku $B[i]$ w czasie realizacji algorytmu to czas działania algorytmu sortowania kubełkowego wynosi

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Tak jest ponieważ złożoność czasowa algorytmu *insertionsort* jest kwadratowa Pesymistyczna złożoność czasowa jest więc równa

$$T_w(n) = \Theta(n) + O(n^2) = O(n^2)$$

Oznaczmy przez N_i zmienną losową zdefiniowaną wzorem $N_i = \left|\{j \in <1, n>; X_j \in I_i\}\right|$. Oczywiście $N_i(\omega) = n_i$. Średnia złożoność czasowa algorytmu *insertionsort* jest przy takich oznaczeniach równa:

$$T_{ave}(n) = E(T(n)) = E(\Theta(n) + \sum_{i=0}^{n-1} O(N_i^2)) = \Theta(n) + \sum_{i=0}^{n-1} E(O(N_i^2))$$

Niech $\chi_{I_i} : R \to R$ dla $i \in <0, n-1>$ oznacza funkcję charakterystyczną przedziału I_i mamy wówczas

$$N_i = \sum_{j=1}^{n} \chi_{I_i}(X_j)$$

Zatem

$$E(N_i^2) = E(\sum_{j=1}^{n} \chi_{I_i}(X_j))^2 = \sum_{k=1}^{n} \sum_{r=1}^{n} E(\chi_{I_i}(X_k) \cdot \chi_{I_i}(X_r)) \quad (*)$$

Jeśli $k \neq r$ to X_r i X_k są niezależnymi zmiennymi losowymi więc

$$E(\chi_{I_i}(X_k) \cdot \chi_{I_i}(X_r)) = E(\chi_{I_i}(X_k)) \cdot E(\chi_{I_i}(X_r)) = P(X_k \in I_i) \cdot P(X_r \in I_i) = \frac{1}{n^2} \quad (**)$$

Jeśli $k = r$ to dostajemy

$$E(\chi_{I_i}(X_k) \cdot \chi_{I_i}(X_k)) = E(\chi_{I_i}^2(X_k)) = E(\chi_{I_i}(X_k)) = P(X_k \in I_i) = \frac{1}{n} \quad (***)$$

Wstawiając (***) i (**) do (*) dostajemy po przesumowaniu

$$E(N_i^2) = \sum_{k=1}^{n} \sum_{r=1}^{n} E(\chi_{I_i}(X_k) \cdot \chi_{I_i}(X_r)) = 2 - \frac{1}{n}$$

Ostatecznie więc

$$T_{ave}(n) = \Theta(n) + \sum_{i=0}^{n-1} E(O(N_i^2)) = \Theta(n) + O(n \cdot (2 - \frac{1}{n})) = \Theta(n) + O(n) = \Theta(n)$$

3.11 Sortowanie przez kopcowanie (ang. heapsort)

Kopiec (ang. heap) to bardzo użyteczna w praktyce struktura danych. W dalszym ciągu podrozdziału zdefiniujemy kopiec i opiszemy algorytm sortowania wykorzystujący kopiec. Będzie to tzw. algorytm sortowania przez kopcowanie (ang. heapsort). Algorytm ten został wynaleziony przez Williamsa. Williams opisał również realizację kolejki priorytetowej za pomocą kopca.

Algorytm *heapsort* demonstruje pewien ogólny pomysłowy sposób konstrukcji algorytmów:

I. wprowadzamy zręcznie pewną strukturę danych tu zwaną kopcem,

II.. realizujemy algorytm wykorzystując wprowadzoną strukturę danych i typowe działania na tej strukturze – tu wykorzystujemy dwie procedury *build-heap* oraz *max-heapify.*

Do zdefiniowania kopca będą nam potrzebne pewne elementarne pojęcia z teorii grafów. Przypomnimy je krótko.

Drzewo to graf niekierowany spójny acykliczny. Acykliczność grafu niekierowanego oznacza, że graf nie zawiera cykli prostych utworzonych z krawędzi grafu. Spójność grafu nieskierowanego oznacza, że każde 2 wierzchołki grafu połączone są ścieżką utworzoną z krawędzi grafu. Jeśli graf niekierowany nie zawiera cykli prostych ale nie jest spójny to nazywamy go *lasem*.

Drzewo z korzeniem (lub drzewo ukorzenione) to drzewo, w którym wyróżniony jest jeden wierzchołek nazywany *korzeniem.*
W drzewie z korzeniem wprowadzamy intuicyjnie jasne pojęcie następnika i poprzednika wierzchołka. Na przykład wierzchołek *c* z przykładu pokazanego na rys. 3.8 ma 3 następniki *f, g* i *h* oraz jeden poprzednik *a*. Zbiór następników wierzchołka *x* ustalonego drzewa *T* oznaczamy symbolem $children_T(x)$. Poprzednik wierzchołka *x* (każdy wierzchołek oprócz korzenia ma poprzednik) oznaczamy symbolem $parent_T(x)$. *Potomkami* wierzchołka *x* drzewa są wszystkie wierzchołki znajdujące się na jakiejkolwiek ścieżce od wierzchołka *x* w dół drzewa (a więc również wierzchołek *x* i następnik *x* są potomkami *x*.

Każdy wierzchołek drzewa (z korzeniem), który ma następnik nazywa się wierzchołkiem wewnętrznym w przeciwnym razie nazywa się ten wierzchołek *liściem*.
Przodkiem wierzchołka *x* drzewa nazywamy każdy wierzchołek leżący na ścieżce od *x* do korzenia drzewa (a więc również *x*).

Jeśli z kontekstu wynika, że rozpatrywane drzewo jest drzewem z korzeniem to często drzewo z korzeniem dla uproszczenia nazywamy drzewem

Głębokość lub poziom wierzchołka w drzewie to jego odległość od korzenia. Wysokość wierzchołka to maksymalna długość drogi w dół od tego wierzchołka do liścia drzewa.

Drzewo z korzeniem jest jednocześnie często traktowane jako struktura danych i reprezentowane albo za pomocą struktury dowiązaniowej (używamy wówczas rekordów i dowiązań) albo za pomocą tablic.

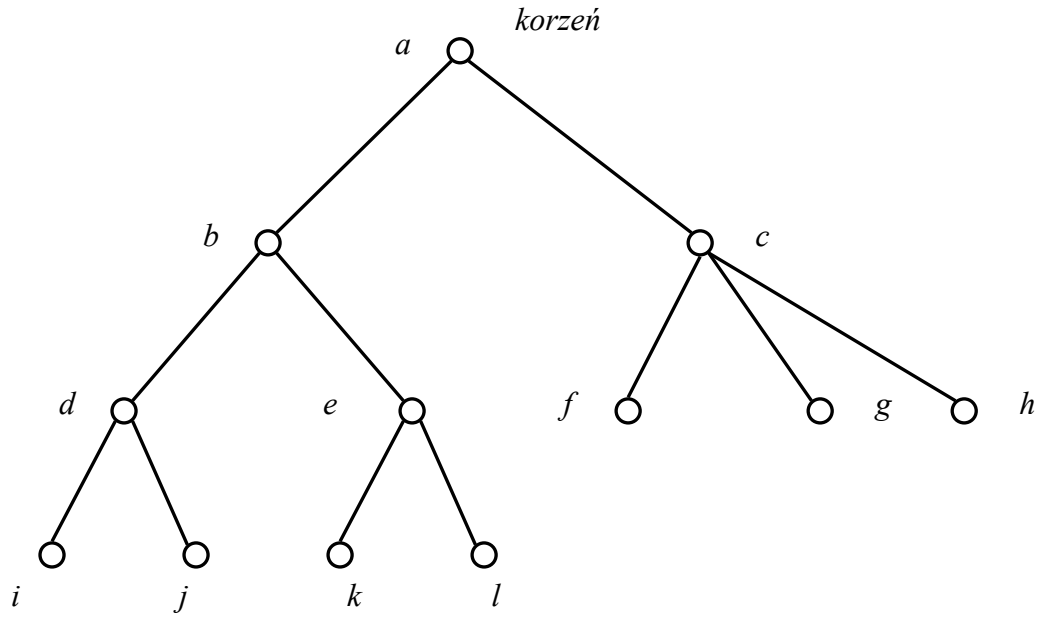
Drzewo binarne. Szczególnym przypadkiem drzewa z korzeniem jest *drzewo binarne*. Drzewo binarne charakteryzuje się tym, że dla każdego wierzchołka x drzewa mamy spełniony warunek $|children(x)| \leq 2$. W drzewie binarnym jednak każdy następnik wierzchołka wewnętrznego x jest albo lewy albo prawy mówimy „lewy syn" i „prawy syn". Tylko jeden z następników może być lewy i tylko jeden prawy.

Wprowadzamy dla drzew binarnych (dla zbioru wierzchołków drzewa binarnego) następujące dwie funkcje

$$left(x) = \begin{cases} \text{lewy syn (lewy nastepnik) jesli istnieje} \\ \text{NIL} \qquad \text{w przeciwnym przypadku} \end{cases}$$

$$right(x) = \begin{cases} \text{prawy syn (prawy nastepnik) jesli istnieje} \\ \text{NIL} \qquad \text{w przeciwnym przypadku} \end{cases}$$

NIL jest tu symbolem specjalnym oznaczającym brak wartości będącej wierzchołkiem. Przykład drzewa binarnego pokazany jest na rys. 3.16.



Rys. 3.8 Przykład drzewa z korzeniem. Korzeniem jest tu wyróżniony wierzchołek *a*.

Kopiec. Kopiec (a dokładniej kopiec binarny typu max) to drzewo binarne takie, że każdemu węzłowi drzewa przypisany jest jeden element np.: pewnego ciągu $a_1, a_2, ..., a_n$ ($a_i \in N$). Mówimy wtedy , że kopiec jest *n*-elementowy.

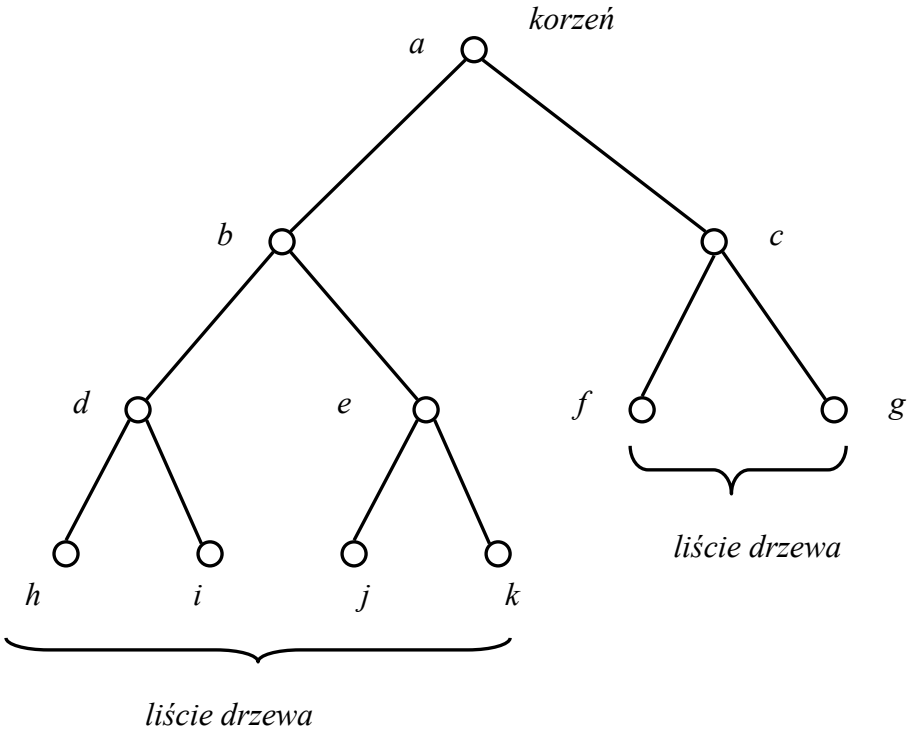
Drzewo binarne kopca (tzn. służącego do zdefiniowania kopca) jest pełne na wszystkich poziomach z wyjątkiem być może najniższego, który jest wypełniony od strony lewej do pewnego miejsca.

Ponadto spełniony jest warunek kopca, zasadnicza własność kopca tzn. w grafie rodzic jest zawsze większy niż synowie tzn. elementy bezpośrednio pod nim.

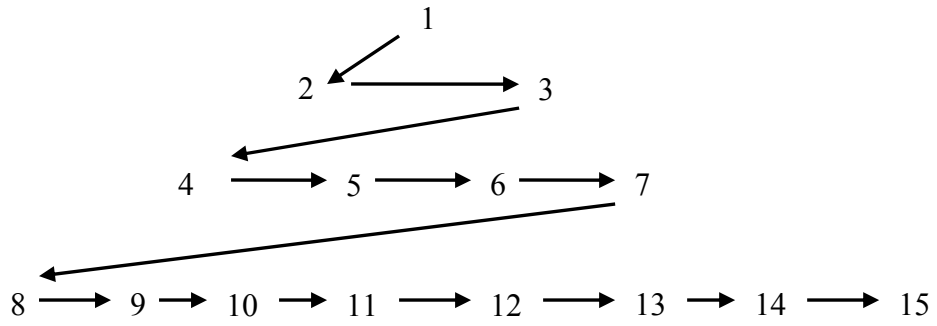
Kopiec jest implementowany w postaci jednowymiarowej tablicy *A*, której przypisujemy 2 atrybuty: *length*[*A*] – liczba elementów tablicy oraz *heap-size* [*A*] – liczba elementów kopca przechowywanych w tablicy *A*[1..*lenth*[*A*]]. Oczywiście musi być spełniona nierówność *heap-size*[*A*] ≤ *length*[*A*].

Żaden element tablicy *A*[1..*length*[*A*]] występujący po *A*[*heap-size*[*A*]] nie jest elementem kopca.

Korzeniem drzewa jest *A*[1]. Ułożenie w tablicy *A* elementów kopca pokazane jest na rys.3.9. Przykład kopca podany jest na rys. 3.10.



Rys. 3.9 Przykład drzewa binarnego służącego do zdefiniowania kopca



Rys. 3.10. Ułożenie w tablicy $A[1..lenth[A]]$ elementów 15 elementowego kopca

Elementy kopca są tak ułożone w tablicy $A[1..lenth[A]]$, że dla danego węzła i (indeksu i tablicy) mamy 3 funkcje: „indeks rodzica": $parent(i)$, „indeks lewego syna": $left(i)$ i „indeks prawego syna": $right(i)$ zdefiniowane wzorami:

$parent(i)$;
begin parent : = $\lfloor i/2 \rfloor$ **end** ;

$left(i)$;
begin $left(i)$: = $2i$ **end** ;

$right(i)$;
begin $right(i)$:= $2i$+1 **end** ;

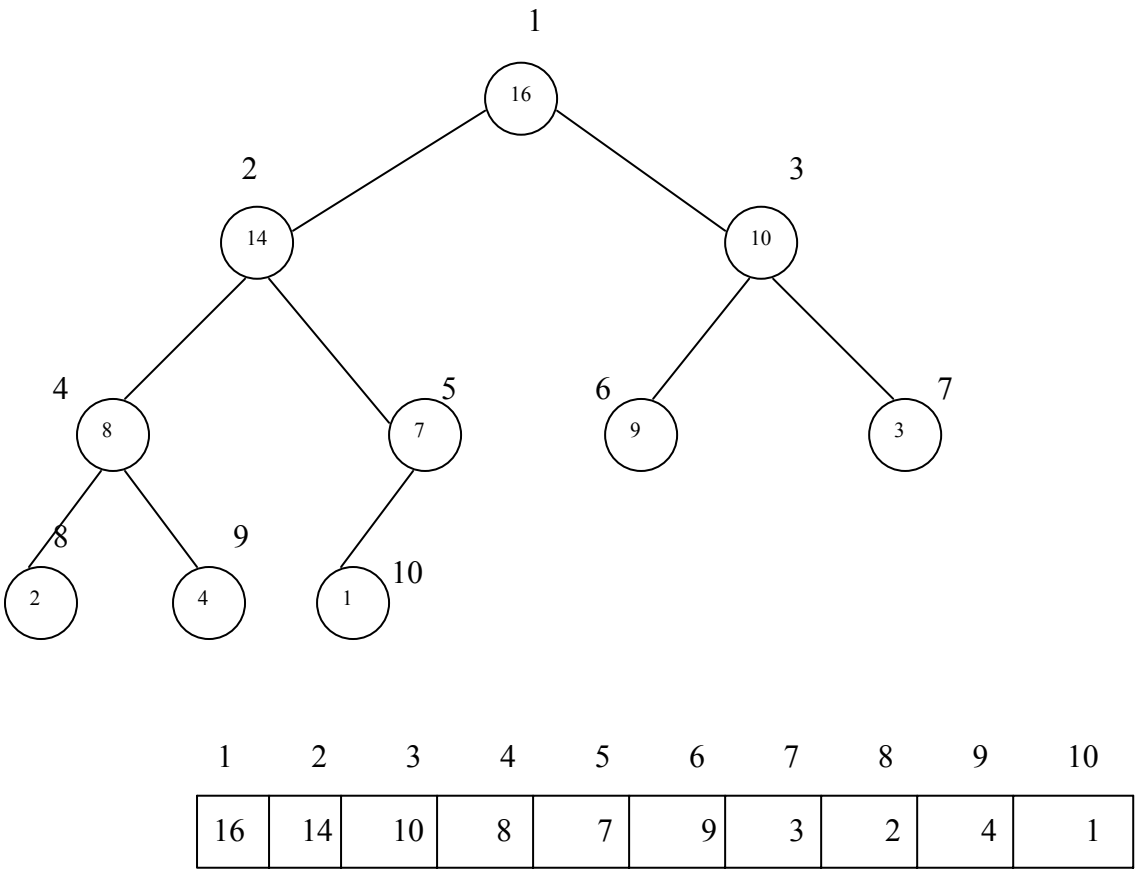
Własność kopca zapisujemy teraz jako: $A[parent(i)] \geq A[i]$ dla każdego $i = 2,...,heap-size(A)$ (czyli dla każdego węzła i, który nie jest korzeniem).

Kopiec mający n elementów ma wysokość $\Theta(\log n)$, a dokładniej $\lceil \log_2 n \rceil$.

Czasami wprowadza się pojęcie **kopca zupełnego**. Przez kopiec zupełny rozumiemy kopiec, którego drzewo binarne jest zupełnym drzewem binarnym tzn. takim, w którym wszystkie poziomy wypełnione są całkowicie z wyjątkiem być może ostatniego, który spójnie wypełniony jest od strony lewej. W niniejszym podrozdziale powyższy warunek został zawarty w definicji kopca.

Ogólnie rzecz biorąc kopce mogą być 2 rodzajów: pierwszy rodzaj to kopce typu max a drugi kopce typu min. W dalszym ciągu używając określenia kopiec będziemy mieli na myśli kopiec typu max. Taki właśnie kopiec zdefiniowaliśmy powyżej.

Kopiec typu min różni się tym od kopca typu max, że własność kopca w przypadku kopca typu min jest taka: każdy rodzic w grafie kopca jest mniejszy od swoich synów co zapisujemy jako: $A[parent(i)] \leq A[i]$ dla każdego $i = 2,...,heap-size(A)$.



Rys.3.11 Przykład 10 elementowego kopca i jego realizacja w postaci macierzy $A[1..10]$

Podstawowe procedury zdefiniowane dla kopca:

1.Procedura *heapify* – działa w czasie $O(\log n)$ i służy do przywracania własności kopca $A[\text{Parent}(i)] \geq A(i)$.

2. Procedura *build-heap* – działa w czasie liniowym tzn. w czasie $O(n)$ i służy do tworzenia kopca z nieuporządkowanej tablicy danych wejściowych.

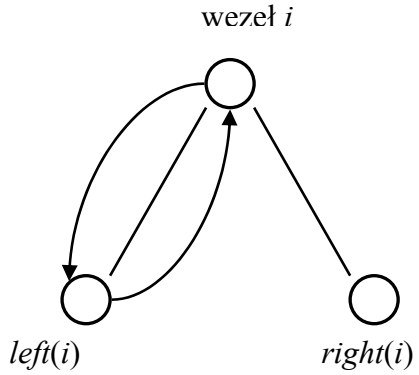
3. Procedury *extract-max* i *insert,* które działają w czasie $O(\log n)$ i pozwalają na użycie kopca jako tzw. kolejki priorytetowej.

Procedura przywracania własności kopca: *max-heapify:*

Mamy jako dane wejściowe procedury tablicę A i indeks i tej tablicy, w którym być może został naruszony warunek kopca. Zakładamy, że drzewa zaczepione w węźle *left*(i) oraz węźle *right*(i) są kopcami.

Zadaniem *max-heapify* jest przywrócenie być może naruszonej węźle i własności kopca. Jeśli własność kopca naruszona jest w węźle i to musimy spowodować, żeby wartość $A[i]$ spłynęła w dół kopca tak, by ostateczne poddrzewo zaczepione w węźle i stało się kopcem.

Istota rzeczy. Wybieramy największy element z trójki $A[left(i)]$, $A[right(i)]$, $A[i]$. Jeśli największym elementem jest $A[i]$ to nic nie musimy robić bo własność kopca jest zachowana. Jeśli któryś z elementów $A[left(i)]$, $A[right(i)]$ jest większy od $A[i]$ to $A[i]$ „spływa" na miejsce większego z nich a ten z kolei zajmuje miejsce $A[i]$ czyli wymieniamy wartości w odpowiednich węzłach. Do węzła syna, w którym nastąpiła wymiana wartości rekurencyjnie stosujemy procedurę *max-heapify*.



Rys. 3.12. Mechanizm działania procedury *max-heapify*(A,i) dla węzła i jeśli spełniona jest nierówność $A[left(i)] \geq \max(A[i], right(i))$.

procedure *max-heapify* (A, i);

begin

l : = *left* (i); r : = *rigth* (i); {obliczanie indeksów synów, lewego i prawego}

if ($l \leq$ *heap-size*[A] and $A[l] > A[i]$) **then** *largest* :=l **else** *largest* : = i;

if ($r \leq$ *heap-size*[A] and $A[r] > A$[largest]) **then** *largest* :=r;

{*largest* jest indeksem określającym położenie największego elementu z trójki: ojciec, prawysyn , lewy syn}

if *largest* $\neq i$ **then** "zamień $A[i]$ z A[*largest*]"; {wymiana}

max-heapify(A, *largest*) {rekurencyjnie stosujemy procedurę *max-heapify*}

end;

Analiza złożoności czasowej procedury *max-heapify* (*A*, *i*) jest prosta. Jeśli wysokość węzła *i* wynosi *h* to wywołujemy procedurę *max-heapify* rekurencyjnie *h* razy dokonując za każdym razem 5 porównań. Zatem złożoność czasowa *max-heapify* jest równa $O(h)$. Ponieważ jednak wysokość drzewa *n*-elementowego binarnego jest równa $\lfloor \log_2 n \rfloor$ więc ogólnie rzecz biorąc dla *n*-elementowego kopca procedura *max-heapify* działa w czasie $O(\log_2 n)$ (lub równoważnie w czasie $O(\lg n)$).

Procedura budowania kopca (procedura build-max-heap)

Procedury *max-heapify* można użyć w sposób wstępujący (ang. bottom-up) do przekształcenia tablicy $A[1..n]$, gdzie n = length $[A]$, w kopiec.

Elementami podtablicy $A[(\lfloor n/2 \rfloor + 1)..n]$ są wszystkie liście drzewa. Pierwszym elementem, który nie jest liściem licząc od prawej strony (czyli strony większych indeksów) jest $A[\lfloor n/2 \rfloor]$. Procedura *build-max-heap* przechodzi począwszy od węzła $\lfloor n/2 \rfloor$ przez pozostałe węzły *i* drzewa i wywołuje w każdym z nich procedurę *max-heapify*(*A,i*). Kolejność przechodzenia przez węzły jest taka, że poddrzewa zaczepione w węźle *i* są kopcami kiedy *max-heapify*(*A*,*i*) zostaje wywołana w tym węźle. Dokładniej procedura *build-max-heap*(*A*) jest następująca.

procedure *build-max-heap*(*A)*;

begin

heap-size[*A*] : = *length*[*A*];

for *i* : = $\lfloor$*length*[*A*]/2$\rfloor$ **downto** 1 **do** *max-heapify*(*A*, *i*);

end ;

Procedura *build-max-heap* została zaproponowana przez Floyda.

Oszacujemy teraz złożoność czasową procedury *build-max-heap*. Wstępnie zauważmy, że ze wzoru na sumę postępu geometrycznego mamy dla $|x| < 1$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Różniczkując powyższy szereg wyraz po wyrazie wewnątrz promienia zbieżności dostajemy

$$\left(\frac{1}{1-x}\right)' = \frac{1}{(1-x)^2} = \sum_{k=1}^{\infty} kx^{k-1}$$

Mnożąc obie strony powyższej równości przez $x \in (-1,1)$ dostajemy

$$\frac{x}{(1-x)^2} = \sum_{k=1}^{\infty} kx^k$$

Co dla $x = \frac{1}{2}$ daje

$$\sum_{k=1}^{\infty} \frac{k}{2^k} = 2\,. \qquad (*)$$

Biorąc pod uwagę fakt, że złożoność czasowa procedury *max-heapify*(A,i) jest rzędu $O(h)$ (a dokładniej ilość porównań jest równa $5h$), gdzie h jest wysokością wierzchołka i kopca dostajemy, że złożoność czasowa procedury *build-max-heap* jest równa

$$\sum_{h=1}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot 5h \qquad (**)$$

Przechodzimy bowiem w procedurze *build-max-heap* przez wszystkie węzły i kopca , które nie są liśćmi. Sumę (**) można oszacować wykorzystując równość (*) następująco

$$\sum_{h=1}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot 5h \le \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{2^{\lfloor \log_2 n \rfloor + 1}}{2^{h+1}} \cdot 5h \le 5 \cdot 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \le 5 \cdot 2^{\log_2 n} \cdot \sum_{h=1}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \le$$

$$\le 5 \cdot n \cdot \sum_{h=1}^{+\infty} \frac{h}{2^h} \le\le 5 \cdot n \cdot 2 = 10n = O(n)$$

Zatem złożoność czasowa procedury *build-max-heap* jest jest rzędu $O(n)$

Algorytm sortowania przez kopcowanie (heapsort)

Algorytm sortowania przez kopcowanie, to inaczej algorytm *heapsort.* Angielski termin „heap” tłumaczymy tu jako kopiec drugie znaczenie słowa „heap” to znana z programowania tzw. sterta oznaczająca w programowaniu ilość wolnej pamięci dostępną programowi (lub

programom) podczas wykonywania programu. Algorytm sortowania przez kopcowanie jest następujący.

Algorytm: ***heapsort*** **(sortowanie przez kopcowanie)**

DANE WEJŚCIOWE: ciąg sortowany $a_1, a_2, ..., a_n$ umieszczony jest w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, ...,\ A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *heapsort*(A,*n*);

var *i* : = integer

begin *build-max-heap* (*A*); {wywołanie procedury *build-max-heap* tworzącej kopiec}

for *i* : = *length* [*A*] **downto** 2 **do** "zamień *A*[1] z *A*[*i*]";

heap-size[*A*] : = *heap-size*[*A*]-1;

max-heapify(*A*,1);

end *heapsort*;

Istota rzeczy-poprawność algorytmu. Budujemy z tablicy wejściowej *A*[1..*n*], gdzie *n* = *length*[*A*] kopiec procedurą *build-heap*(*A*). Największy element znajdzie się po wykonaniu tej procedury w *A*[1].

Zamieniamy wartości zmiennych *A*[1] i *A*[*n*]. (wartość *A*[1] znajdzie się wówczas na pewno na dobrym miejscu tzn. skrajnej prawej pozycji). Odrzucamy węzeł *n* z kopca a więc „skracamy kopiec" i stosujemy procedurę *max-heapify* do krótszej tablicy *A*[1..(*n*-1)] i węzła 1 przekształcając ją znowu w kopiec itd. aż do uzyskania kopca o rozmiarze 2.

Złożoność czasowa. Złożoność czasowa pesymistyczna algorytmu *heapsort* wynosi $O(n \lg n)$ ponieważ wywołanie procedury *build-max-heap* (*A*) zajmuje czas $O(n)$ a każde z *n*-1 wywołań procedury *max-heapify*(*A*, *i*) zajmuje czas $O(\lg n)$. Ponieważ liczba porównań w algorytmie nie zależy od danych dostajemy więc:

$$T_w(n) = T_{ave}(n) = O(n \lg n)$$

Złożoność przestrzenna. Algorytm *heapsort* sortuje w miejscu.

Kolejka priorytetowa-realizacja kolejki priorytetowej za pomocą kopca

Struktury danych jak już powiedzieliśmy w rozdziale 1 to obiekty, na których operuje algorytm. Istotne są przy tym związane ze strukturą danych typowe realizowane na niej operacje. Można więc na strukturę danych patrzeć jak na parę uporządkowaną:

(sama struktura, typowe operacje na tej strukturze)

Bardzo ważny jest też sposób implementacji w komputerze samej struktury danych oraz implementacja operacji na strukturze danych. Zależy nam przy tym na tym by wszystkie operacje realizowane na strukturze były możliwie szybkie. Sama definiowana struktura realizowana jest z reguły za pomocą innych znacznie prostszych struktur danych.

Podobnie jest w przypadku kolejki priorytetowej.

Kolejka priorytetowa to jedna z ważniejszych struktur danych. Mówiąc najprościej kolejka priorytetowa z abstrakcyjnego punktu widzenia to multizbiór S wyposażony w pewne specjalne operacje (których listę za chwilę podamy). Multizbiór to zbiór, w którym elementy mogą się powtarzać. Zakładamy przy tym, że zbiór multizbioru S jest zbiorem liniowo uporządkowanym.

Można też patrzeć na kolejkę priorytetową jak na zbiór (lub multizbiór) elementów S wyposażony w funkcję klucza $key : S \to A$, gdzie A jest zbiorem liniowo uporządkowanym. Funkcja klucza wprowadza wówczas uporządkowanie liniowe w S. Można powiedzieć, że kolejka priorytetowa to struktura danych reprezentująca zbiór S elementów z tym, że z każdym elementem zbioru związana jest wartość zwana kluczem**.**

Kolejka priorytetowa tym różni się od listy, że w kolejce priorytetowej (tak jak w przypadku zbioru) elementy nie są ustawione w ciąg.

Kolejki priorytetowe podobnie jak kopce mogą być typu max i typu min. Zajmiemy się tylko kolejkami typu max. Operacje definiowane dla kolejki priorytetowej (typu max) to:

1. $construct(q, S)$ - utworzenie multizbioru $S = \{a_1, a_2, ..., a_n\}$ z danej listy $q = [a_1, a_2, ..., a_n]$

2. $insert(S, x)$ - dodawanie, wstawianie elementu x do multizbioru $S = \{a_1, a_2, ..., a_n\}$ czyli $S := S \cup \{x\}$

3. *maximum*(*S*) - daje w wyniku element zbioru *S* z największym kluczem

4. *extract-max*(S) - daje w wyniku element zbioru *S* z największym kluczem i jednocześnie usuwa ten element

5. $delete\_max(S)$ - usunięcie z multizbioru $S = \{a_1, a_2, ..., a_n\}$ największego elementu

Dla kolejki priorytetowej typu min zamiast operacji *maximum*(*S*) . *exstract-max*(*S*) oraz $delete\_max(S)$ mamy odpowiednio operacje *minimum*(*S*), *extract-min*(S), oraz $delete\_min(S)$,

np. $delete\_min(S)$ - jest operacją usunięcie z multizbioru $S = \{a_1, a_2, ..., a_n\}$ najmniejszego elementu

Kolejkę priorytetową można zaimplementować za pomocą:

- listy nieuporządkowanej

- listy uporządkowanej (czyli listy). Z kolei listę można reprezentować np. za pomocą metody dowiązaniowej lub tablicowej.

- kopca (ang heap)

Kopiec jest często wykorzystywany do implementacji kolejki priorytetowej gdyż implementacja kolejki priorytetowej za pomocą kopca jest bardzo wygodna.

Ważnym zastosowaniem kolejki priorytetowej jest tzw. szeregowanie zadań do wykonania na komputerze w systemie operacyjnym wielozadaniowym.

Jeśli tablica $A[1:n]$, gdzie $heap-size(A) \leq n$ implementuje kopiec reprezentujący multizbiór *S* to kolejkę priorytetową (typu max) otrzymamy z kopca implementując wszystkie potrzebne do utworzenia kolejki priorytetowej operacje. Na przykład operację *maximum*(*S*) realizuje procedura:

procedure *heap-maximum*(*A*);

begin *heap-maximum* : = *A*[1] **end**;

Procedura *heap-maximum*(*A*) oczywiście nie zmienia struktury kopca Złożoność obliczeniowa procedury *heap-maximum*(*A*) jest rzędu $\Theta(1)$.

Operację *extract-max*(*S*) realizuje następująca procedura *heap-extract-max*(*A*):

procedure *heap-extract-max*(*A*);

begin

if *heap-size*(*A*) < 1 **then** "error- kopiec pusty" ;

max : = *A*[1];

A[1] : =*A*[*heap-size*(A)-1];

heap-size(*A*) : = *heap-size*(A)-1;

max-heapify(*A*,1); {główny problem to przywrócenie własności kopca tablicy *A*}

heap-extract-max : =*max*;

end;

Złożoność czasowa procedury *heap-extract-max*(*A*) jest rzędu $O(\lg n)$. Podobnie wykorzystując kopiec można zrealizować pozostałe operacje na kolejce priorytetowej.

3. 12 Sortowanie szybkie - quicksort

Algorytm sortowania szybkiego to inaczej algorytm *quicksort* Jest to najszybszy ze znanych algorytmów sortowania opartych na porównaniach elementów ciągu sortowanego. Algorytm *quicksort* sortuje ciąg wejściowy w średnim czasie czasie $T_{ave}(n) = \Theta(n \log n)$. Algorytm *quicksort* został podany przez Hoara.

Konstrukcja algorytmu *quicksort* oparta jest na zasadzie dziel i zwyciężaj oraz rekurencji. Zasadnicza dla algorytmu *quicksort* jest procedura *partition*. Procedura *partition* ma kilka wersji o takiej samej złożoności czasowej. Poniżej zostaną opisane 2 wersje procedury *partition.* Wersja oryginalna Hoara i wersja zaproponowana przez N.Lomuto.

Algorytm: quicksort – sortowanie szybkie

DANE WEJŚCIOWE: ciąg sortowany $(a_1, a_2, ...a_n)$ umieszczony jest w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, ..., \quad A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ umieszczony w tablicy $A[1:n]$ tzn. $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *quicksort* (*A, l, r*) : integer);

{parametry *l, r* są typu integer przy czym $l < r$ i definiują podciąg a dokładniej podtablicę *A*[*l*,*r*] do posortowania, w wyniku wykonania instrukcji **if** *n*>1 **then** *quicksort*(*A*,1,*n*) zostaje posortowana cała lista wejściowa}

var *j* : integer;

begin *j* : = *partition* (*A*, *l, r*); {wywołanie procedury *partition*}
if $j - 1 > 1$ **then** *quicksort* $(l, j-1)$;
if $r > j + 1$ **then** *quicksort* $(j+1, r,)$;
{rekurencyjne wywołanie procedury *quicksort*}
end *quicksort*;

Kluczowe znaczenie ma tu funkcja *partition,* która przekształca podtablicę $A[l..r]$ tak, że :

1.Element $v = A[j]$ znajduje się na właściwym ostatecznym miejscu w tablicy oraz

2. $A[l], A[l+1], ..., A[j-1] \le v$ i $v \le A[j+1], A[j+2], ..., A[r]$ tzn. wszystkie elementy na lewo od $A[j]$ są mniejsze równe od $A[j]$ a wszystkie elementy na prawo od $A[j]$ są większe równe od $A[j]$.

3.Jako wartość funkcja *partition* podaje indeks *j* elementu rozdzielającego $v = A[j]$

Procedura *partition* działa następująco.

1.Jako tzw. element rozdzielający ciąg $A[l...r]$ wybieramy wstępnie $v = A[l]$

2.Przeglądamy ciąg $A[l+1], A[l+2],..., A[r]$ od lewej strony dopóki nie znajdziemy elementu większego równego od $A[l]$

3.Potem przeglądamy $A[l+1],..., A[r]$ od strony prawej dopóki nie znajdziemy elementu mniejszego równego od $A[l]$.

4.Dwa elementy, na których się zatrzymujemy wymieniamy miejscami i wznawiamy procedurę od lewej, a później od prawej strony

5.Kiedy wskaźniki pozycji zatrzymania się po lewej i prawej stronie spotkają się, proces dzielenia jest zakończony

Jak łatwo widać złożoność czasowa procedury *partition* wynosi $\Theta(n)$. Oryginalna wersja procedury *partition* zaproponowana przez Hoara jest następująca:

procedure *partition*(*A, l, r*);

begin {zakładamy, że *l*<*r*, oraz $A[l], A[l+1],..., A[r] \le A[r+1]$}

$v := A[l]$; $i := l$; $j := r+1$

repeat **repeat** $i := i+1$ **until** $A[j] \ge v$;
repeat $j := j-1$ **until** $A[j] \le v$;
if *i*<*j* **then** „ $A[i]$ zamień miejscem z $A[j]$”
until $a[j] \le i$;

$A[l] := A[j]$; $A[j] := v$; {ustawienie $a[l]$ na właściwym miejscu}

partition : = *j*

end *partition*;

Druga wersja procedury *partition* podana przez N.Lomuto jest następująca:

procedure *partition*(*A, l, r*);

begin *x* : = *A*[*r*]; *i* : = *p*-1;

for *j* : =*l* **to** *r*-1 **do** **begin**
if $A[j] \le x$ **then** *i*:=*i*+1;
“zamień *A*[*i*] z *A*[*j*]”
end ;

“zamień *A*[*i*+1] z *A*[*r*]”

partition : = *i*+1;

end *partition*;

Pełny opis algorytmu *quicksort* jest następujący:

Algorytm: quicksort – sortowanie szybkie

DANE WEJŚCIOWE: ciąg sortowany $(a_1, a_2, ...a_n)$ umieszczony jest w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, ..., \ A[n] = a_n$

DANE WYJŚCIOWE: ciąg posortowany $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ umieszczony jest w tablicy $A[1:n]$ tzn. $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

procedure *quicksort* (*l,r* : integer);

{parametry *l,r* określają podciąg do posortowania, w wyniku wykonania instrukcji **if** *n*>1 **then** *quicksort*(1,*n*) zostaje posortowany cały ciąg, zakładamy, że $A[n+1] = +\infty$ }

var *v,i,j* : integer;

begin {początek funkcji *partition*}

{zakładamy, że *l*<*r*, oraz $A[l], A[l+1], ..., A[r] \le A[r+1]$ }

$v := A[l]$; $i := l$; $j := r+1$

repeat **repeat** $i := i+1$ **until** $A[j] \ge v$;
repeat $j := j-1$ **until** $A[j] \le v$;
if *i*<*j* **then** „ $A[i]$ zamień miejscem z $A[j]$ ”
until $a[j] \le i$;

$A[l] := A[j]$; $A[j] := v$; {ustawienie $a[l]$ na właściwym miejscu}
{koniec funkcji partiton}

if $j-1 > l$ **then** $quicksort(l, j-1)$;

if $r > j+1$ **then** $quicksort(j+1, r)$
end *quicksort*;

Pesymistyczna złożoność obliczeniowa algorytmu *quicksort* .
Jak już stwierdziliśmy złożoność czasowa procedury *partition* pracującej na *n* elementowej podtablicy jest $\Theta(n)$. W algorytmie *quicksort* wywołujemy rekurencyjnie procedurę *partition*. Najgorszy przypadek jeśli chodzi o złożonoś czasową algorytmu *quicksort* mamy wtedy, gdy procedura *partition* rozbija wejściową tablicę złożoną z *n* elementów na 2 podtablice złożone z *n*-1 oraz 1 elementu i podobnie się dzieje przy każdym odwołaniu rekurencyjnym. Równanie opisujące ilość porównań w tym przypadku to

$$T_w(n) = T_w(n-1) + T_w(0) + c \cdot n$$

dla pewnego $c > 0$. $T_w(n)$ jest więc sumą postępu arytmetycznego a więc $T_w(n)$. Można pokazać, że taki przypadek jest dla każdego *n* możliwy np. tak właśnie jest w przypadku całkowicie posortowanego ciągu. Warto przypomnieć, że dla takiego przypadku algorytm sortowania *insertionsort* działa w czasie liniowym tzn. w czasie $O(n)$.

Pesymistyczna złożoność czasowa opisanego wyżej algorytmu *quicksort* obliczona dokładnie jest równa

$$T_w(n) = \frac{1}{2}n^2 + \frac{3}{2}n - 2 \qquad \text{dla } n > 0$$

Optymistyczna złożoność obliczeniowa algorytmu *quicksort* .Najlepszy czas wykonania algorytmu *qicksort* dostajemy w przypadku podziału ciągu wejściowego (przy każdym wywołaniu procedury *partition*) na 2 równe co do długości (lub różniące się co do długości o 1) podtablice. Czas wykonania procedury *partition* jest proporcjonalny do długości podtablicy. Równanie rekurencyjne na złożonoć czasową w przypadku np. $n = 2^k$ można zapisać następująco

$$T(n) = 2T(\frac{1}{2}n) + \Theta(n)$$

Rozwiązaniem tego równania jest $T(n) = O(n \lg n)$. Intuicyjnie rzez biorąc uzyskany wynik jest zrozumiały. Czas obliczeń jest proporcjonalny do wysokości drzewa podziału listy wejściowej na podlisty.

Średnia złożoność czasowa algorytmu *quicksort*. Przyjmijmy model probabilistyczny danych wejściowych taki jak w przypadku analizy algorytmu *insertionsort* Przy takim założeniu procedura *partition* działająca na tablicy o długości *n* może zwrócić dowolną z *n* wartości (pozycję elementu rozdzielającego) z jednakowym prawdopodobieństwem $\frac{1}{n}$.

Równanie rekurencyjne na średnią złożoność czasową algorytmu *quicksort* jest więc następujące

$$T_{ave}(n) = \underbrace{(n-1)}_{\substack{\text{czas realizacji} \\ \text{procedury partition}}} + \sum_{j=1}^{n} \frac{1}{n}(T_{ave}(j-1) + T_{ave}(n-j))$$

W powyższym wzorze $\frac{1}{n}$ jest prawdopodobieństwem tego, że zwracana przez procedurę *partition* wartość jest równa j. Suma $T_{ave}(j-1) + T_{ave}(n-j)$ jest średnim czasem sortowania podtablic jeśli procedura *partition* zwraca wartość j. Rozwiązując powyższe równanie rekurencyjne (np. za pomocą przekształcenia Z) dostajemy, że średnia złożoność czasowa algorytmu *quicksort* wynosi

$$T_{ave}(n) = \frac{2}{\log e} n \log n + O(n)$$

gdzie log oznacza logarytm dziesiętny.

Pesymistyczna wrażliwość czasowa algorytmu *quicksort* jest równa $\Delta(n) = O(n^2)$ a oczekiwana wrażliwość czasowa jest równa

$$\sigma(n) = \underbrace{\sqrt{7 - \frac{2}{3}\pi^2}}_{\approx 0,08} \cdot n + O(\log n)$$

3.13 Szybkie algorytmy wyznaczania *k*-tego elementu co do wartości w ciągu.

Określenie *k*-tego elementu co do wartości w ciągu w ciągu liczbowym $a_1, a_2, ..., a_n$ budzi pewne wątpliwości. Otóż umawiamy się, że jest to *k*-ty co do wartości element licząc od prawej strony po posortowaniu ciągu tzn. dla $k = 1$ wybieramy element maksymalny a dla $k = n$ minimalny.

Wyznaczanie *k*-tego elementu co do wartości w ciągu liczbowym $a_1, a_2, ..., a_n$ służy np. do obliczania mediany ($k = \lceil n/2 \rceil$ lub $k = \lfloor n/2 \rfloor$), elementu największego w ciągu i elementu najmniejszego w ciągu oraz statystyk pozycyzyjnych. Pośrednio zaś do realizacji filtrów kwantylowych i medianowych w cyfrowym przetwarzaniu sygnałów i obrazów.

Oczywiście możemy najpierw posortować ciąg $a_1, a_2, ..., a_n$ jakimkolwiek algorytmem sortowania a potem wybrać element $A[k]$, ale istnieją algorytmy znacznie szybsze działające w czasie liniowym np. opisany niżej algorytm Hoara.

Najpierw omówimy bardzo proste algorytmy wyboru elementu największego i najmniejszego.

Algorytm: Algorytm Hoare'a:

DANE WEJŚCIOWE: ciąg $a_1, a_2, ...a_n$ umieszczony w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

DANE WYJŚCIOWE: element *k*-ty co do wartości ciągu $a_1, a_2, ...a_n$

---

function (1, *n*, *k*)
Dane wejsciowe: $a_1, a_2, ..., a_n$ oraz $k \in \langle 1, n \rangle$
Dane wyjściowe: takie *i*, że a_i ma *k*-tą największą wartość (dla $k = 1$ wybieramy element maksymalny, dla $k = n$ minimalny)

function *select*(*l, r, k* : integer) : integer;

{$l \le k \le r - l + 1$, funkcja wyznacza *k*-ty największy co do wartości element w tablicy $A[l..r]$}

var *j*: integer

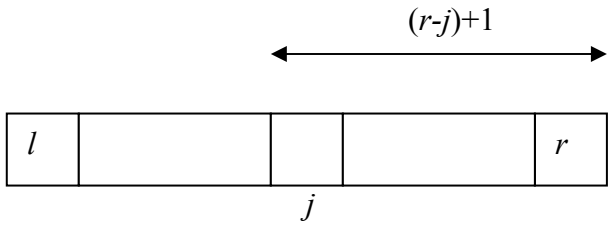
begin

if $l < r$ **then**

begin

```
        j := partition (l ,r)
    if      k − 1 = r − j   then   select : = j   else
    if      k − 1 < r − j   then   select : = select ( j + 1, r, k)  else
    select : = select  (l, j − 1, k − r − j + 1))
    end     else     select : = 1

end select;
```



$$A[l], A[l+1],..., A[j-1] \le A[j] \le A[j+1], A[j+2],..., A[r]$$

j jest właściwym miejscem dla $A[l]$

Rys. 3.13 Ilustracja działania procedury *partition* w algorytmie Hoara

Algorytm Hoara wykorzystuje rekurencyjnie procedurę *partition* i ma pesymistyczną złożoność czasową $T_w(n) = O(n^2)$. Istotnie jeśli ciąg $a_1, a_2, ...a_n$ jest już posortowany to algorytm Hoara odwołuje się $n-1$ razy do procedury *partition* , która ma złożoność czasową proporcjonalną do długości podtablicy na której pracuje co daje w rezultacie

$$(n-1) + (n-2) + ... + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

porównań wyrazów ciągu plus 2 porównania indeksów przy każdym wywołaniu funkcji select co daje w sumie $O(n^2)$ porównań.

Średnia złożoność czasowa algorytmu Hoara jest liniowa tzn. $T_{ave}(n) = O(n)$ a dokładniej $T_{ave}(n) \le 4n$. Zakładamy przy tym , że danymi wejściowymi są permutacje *n* elementowego zbioru np. $< 1, n >$ i każda permutacja jest jednakowo prawdopodobna. W dowodzie faktu, że $T_{ave}(n) \le 4n$ korzystamy z analogicznego jak w przypadku algorytmu *quicksort* równania rekurencyjne na średnią złożoność czasową. Rozwiązując to równanie dostajemy oszacowanie $T_{ave}(n) \le 4n$.

Algorytm Hoara działa w miejscu tzn. złożoność przestrzenna jest równa $S(n) = O(1)$.

Inny znany algorytm wyznaczania *k*-tego elementu co do wartości w ciągu to algorytm Bluma-Pratta-Rivesta-Trajana. Jest to modyfikacja algorytmu Hoara działająca w czasie liniowym w najgorszym przypadku tzn. mamy $T_w(n) = O(n)$.

Algorytm: Algorytm wyboru elementu maksymalnego

DANE WEJŚCIOWE: ciąg $a_1, a_2, ...a_n$ umieszczony w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, ..., \ A[n] = a_n$

DANE WYJŚCIOWE: element *k*-ty co do wartości ciągu $a_1, a_2, ...a_n$

---

function *maximum*(*A,n*) ;{funkcja wyznacza największy element w tablicy $A[1..n]$}

var *i*: integer;
max: real;

begin *max***:=***A*[1];

for $i := 2$ **to** n **do if** $max < A[i]$ **then** $max := A[i]$;

$maximum: = A[i]$;

end *maximum*;

Dokonujemy w powyższym algorytmie *n*-1 porównań i jest to minimalna liczba porównań niezbędna do wyznaczenia elementu maksymalnego ciągu $a_1, a_2, ...a_n$. Jest to więc algorytm optymalny pod względem liczby dokonywanych porównań. Analogicznie można wyznaczyć wartość minimalną ciągu.

Czasami potrzebny jest jednoczesny wybór elementu maksymalnego i minimalnego. Oczywiście można w tym celu wykorzystać powyższy algorytm w wersji dla maksimum i dla minimum łącząc je w jeden algorytm ale uzyskujemy wówczas algorytm wymagający $2(n-1)$ porównań.

Okazuje się, że wystarczy do jednoczesnego wyboru elementu maksymalnego i minimalnego tylko $3\left\lfloor \frac{n}{2} \right\rfloor$ porównań. Zasadniczy pomysł polega na tym, że porównujemy kolejne pary wyrazów ciągu ze sobą a następnie mniejszy z nich porównujemy z dotychczasowym minimum a większy z dotychczasowym maksimum. Algorytm oparty na tym pomyśle podany jest poniżej.

Algorytm: Algorytm jednoczesnego wyboru elementu maksymalnego i minimalnego

DANE WEJŚCIOWE: ciąg $a_1, a_2, ...a_n$ umieszczony w tablicy $A[1:n]$ tzn. $A[1] = a_1, A[2] = a_2, ...,\ A[n] = a_n$, zakładamy,że $n \geq 2$

DANE WYJŚCIOWE: element maksymalny *max* i minimalny *min* ciągu $a_1, a_2, ...a_n$

---

procedure *max_min*(*A,n*) ;

{procedura *max_min* wyznacza największy element *max* i najmniejszy *min* w tablicy $A[1..n]$}

var *i*: integer; *max*, *min*, *swap* : real;

begin **if** „*n* parzyste” **then**

begin
$i := 3$ **;**
if $A[1] \geq A[2]$ **then begin** $max := A[1]$; $min := A[2]$ **end else**
begin $max := A[2]$; $min := A[1]$ **end** ;
end ;

else

begin
$i := 2$ **;**
if $A[1] \geq A[2]$ **then begin** $max := A[1]$; $min := A[1]$ **end else**
begin $max := A[2]$; $min := A[1]$ **end** ;
end ;

if $n = 2$ **then goto** *end_of_procedure*

{zasadniczy pomysł realizowany za pomocą poniższej pętli while to porównywanie w jednym przebiegu pętli 4 elementów $A[i], A[i+1]$, *min* i *max* i aktualizacja *min* i *max* }
while $i \leq n-1$ **do begin**

if $A[i] > A[i+1]$ **then begin** $swap := A[i]$; $A[i] := A[i+1]$; $A[i+1] := swap$ **end**
if $min > A[i]$ **then** $min := A[i]$;
if $max < A[i+1]$ **then** $max := A[i+1]$;
$i := i + 2$;
end ;

end_of_procedure:
end *max_min*;

Jeśli $n \geq 2$ jest parzyste to w procedurze *max_min*(*A*,*n*) dokonujemy $3 \cdot \frac{n}{2}$ porównań . Jeśli $n \geq 2$ jest nieparzyste to dokonujemy $3 \cdot \frac{n-1}{2}$ porównań. Zatem dla dowolnego $n \geq 2$ dokonujemy w procedurze *max_min*(*A*,*n*) $3 \cdot \left\lfloor \frac{n}{2} \right\rfloor$ porównań.

3.14 Sortowanie zewnętrzne

Cel sortowania wewnętrznego i zewnętrznego jest taki sam. Algorytmy sortowania zewnętrznego są stosowane w sytuacji, gdy ciąg sortowany $a_1, a_2, ..., a_n$ (lub jak mówimy czasem lista sortowana $q = (a_1, a_2, ..., a_n)$) nie mieści się ze względu na zbyt duży rozmiar (czyli zbyt duże *n*) w pamięci wewnętrznej systemu komputerowego. Ciąg sortowany umieszczany jest w tej sytuacji w pliku zapisanym na twardym dysku lub taśmie magnetycznej. Rzutuje to oczywiście na stosowane algorytmy.

Żeby ocenić kiedy stają się przydatne algorytmy sortowania zewnętrznego trzeba zdawać sobie sprawę z różnic w pojemnościach pamięci wewnętrznej i zewnętrznej systemu komputerowego. Na przykład dla mikroprocesora Pentium 4 pamięć wewnętrzna ma typową pojemność 0,5 GB - 64 GB. Typowy pojedynczy napęd dyskowy ma natomiast pojemność 100 GB - 400 GB a macierze dyskowe RAID mają pojemności przekraczające 10 TB.

Podstawowym stosowanym przy sortowaniu zewnętrznym chwytem jest stosowanie algorytmu scalania *merge*. Dokładniej: rozbijamy długi ciąg $a_1, a_2, ..., a_n$ na relatywnie krótkie podciągi dające się posortować w pamięci wewnętrznej, sortujemy je a następnie scalamy posortowane już podciągi (umieszczone w pliku np. na twardym dysku) algorytmem *merge*.

Operacją dominującą w algorytmach sortowania zewnętrznego są przesłania pomiędzy pamięcią zewnętrzną a pamięcią wewnętrzną. Oceniając złożoność czasową algorytmu sortowania zewnętrznego liczymy więc nie tylko liczbę porównań (parametr nie tak bardzo istotny w przypadku sortowania zewnętrznego) ale przede wszystkim liczbę przesłań pomiędzy pamięcią wewnętrzną i zewnętrzną.

W opisie i analizie algorytmów sortowania zewnętrznego używa się pojęcia bloku posortowanego (inaczej sekwensu). Przez blok posortowany rozumiemy dowolną posortowaną część listy $q = (a_1, a_2, ..., a_n)$. Mówiąc blok posortowany lub sekwens myślimy o bloku posortowanym niemalejąco czyli sekwensie niemalejącym.

W niniejszym podrozdziale umawiamy się, że blok o długości *m* mieści się w pamięci wewnętrznej i może być posortowany algorytmem sortowania wewnętrznego a ponadto zakładamy, że $s = n/m$ jest liczbą naturalną tzn. cały ciąg $a_1, a_2, ..., a_n$ daje się podzielić na *s* podciągów o długości *m*.

W algorytmach sortowania zewnętrznego przydaje się algorytm scalania wielowejściowego (z $p \geq 2$ wejściami) stanowiący uogólnienie algorytmu *merge* . Załóżmy, że mamy *p* niemalejących sekwensów stanowiących dane wejściowe. Łączymy je algorytmem scalania w jeden sekwens o długości odpowiednio większej. Algorytm działa tak, że w każdej chwili pobieramy *p* elementów do porównania, wybieramy najmniejszy z nich i wyprowadzamy do sekwensu wyjściowego. Jeśli elementy porównywane są równe to wybieramy dowolny z nich. Jeśli *p* nie jest zbyt duże to najmniejszy element z *p* elementów znajdujemy dokonując *p*-1 porównań.

Algorytm: Sortowanie zewnętrzne – zrównoważone scalanie wielofazowe z 4-ma plikami

DANE WEJŚCIOWE: ciąg sortowany $a_1, a_2, ...a_n$ umieszczony w pliku wejściowym P_{we}
DANE WYJŚCIOWE: ciąg posortowany $a_{\pi(1)}, a_{\pi(2)}, ..., a_{\pi(n)}$ umieszczony w pliku P_{wy}

---

procedure *scalanie wielofazowe z 4-ma plikamit*

begin

1.Tworzymy z pliku wejściowego P_{we} bloki posortowane (sekwensy) sortując po kolei podciągi ciągu $a_1, a_2, ...a_n$ (o długości *m*) algorytmem sortowania wewnętrznego a następnie rozrzucając je w sposób zrównoważony do plików pomocniczych P_1 i P_2.

2.Czytamy kolejne elementy plików P_1 i P_2 scalając kolejno bloki posortowane o długości *m* i tworząc bloki posortowane o długości 2*m*. Rozrzucamy w sposób zrównoważony bloki posortowane do plików pomocniczych P_3 i P_4.

3.Jeśli $2m = n$ kończymy algorytm, podstawiając $P_{wy} := P_1$.
Jeśli $2m \neq n$ zmieniamy nazwę P_3 na P_1 i nazwę P_4 na P_2 , podstawiamy $m := 2m$ i przechodzimy do punktu 2.

end.

Zauważmy, że jeśli mamy w pliku P_{we} już $s = n/m$ sekwensów oraz $2^{k-1} < s \leq 2^k$ to wykonamy w naszym algorytmie $k = \lceil \log_2 s \rceil$ przebiegów scalania. Zatem ilość wszystkich porównań (nie licząc punktu 1 algorytmu) jest równa

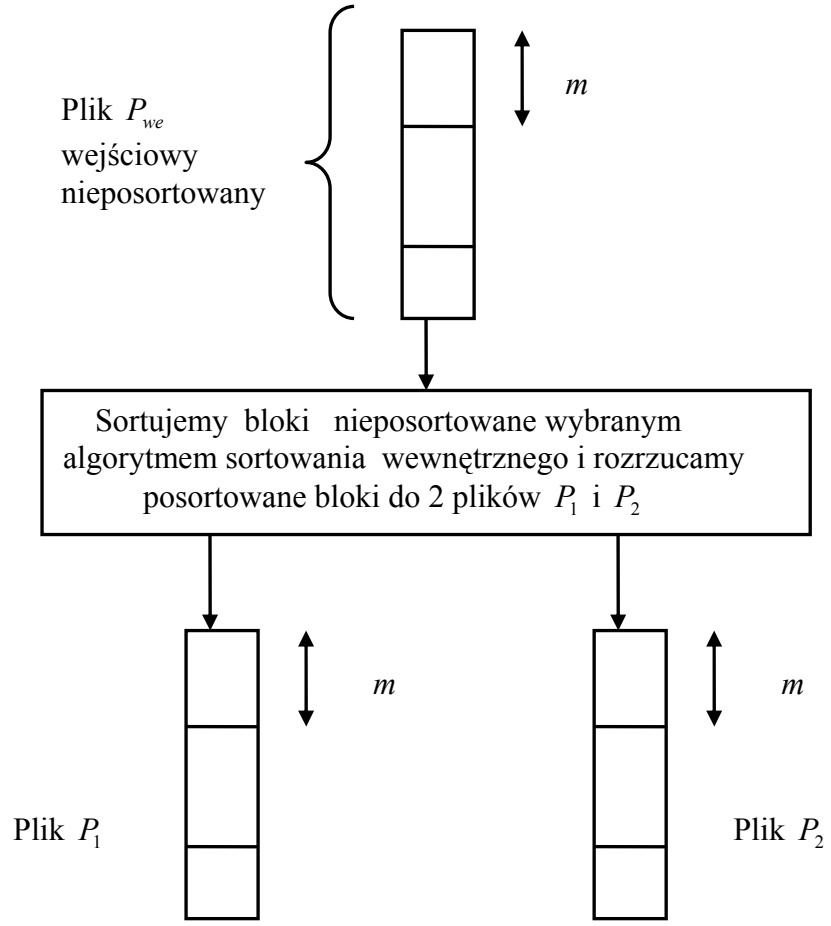
$$n\lceil \log_2 s \rceil = n\left\lceil \log_2 \frac{n}{m} \right\rceil$$

a przesłań (nie licząc punktu 1 algorytmu)

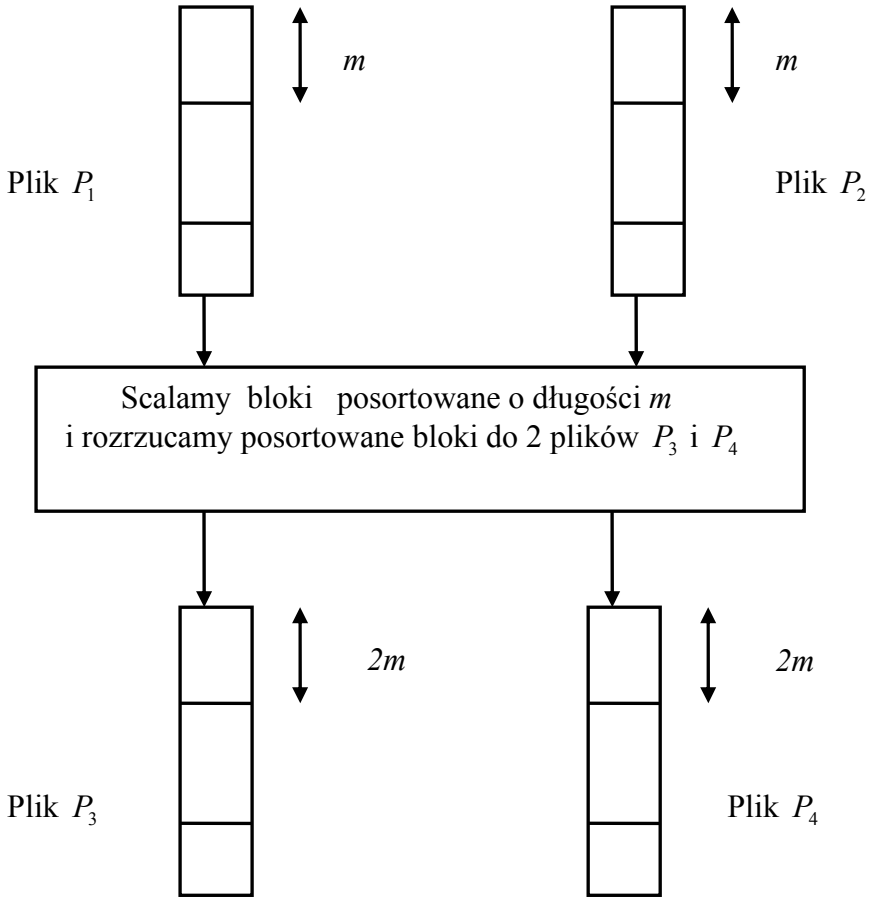
$$2n\lceil \log_2 s \rceil = 2n\left\lceil \log_2 \frac{n}{m} \right\rceil$$

Uwaga Powyższy algorytm można w naturalny sposób zmodyfikować stosując $2p$ ($p \geq 2$) plików pomocniczych P_1, P_2, …, P_p,…, P_{2p} i scalanie wielowejściowe z p wejściami. Wówczas liczba porównań (nie licząc punktu 1 algorytmu) będzie równa

równa $O(n\lceil \log_p s \rceil) = O(n\left\lceil \log_p \frac{n}{m} \right\rceil)$ a liczba przesłań (nie licząc punktu 1 algorytmu) $2n\lceil \log_p s \rceil = 2n\left\lceil \log_p \frac{n}{m} \right\rceil$.



Rys. 3.14 Pierwszy etap sortowania zewnętrznego ze scalaniem wielofazowym z 4-ma plikami



Rys. 3.15 Drugi etap sortowania zewnętrznego ze scalaniem wielofazowym z 4-ma plikami

3.15 Sieci sortujące

Pojęciami podstawowymi w tym podrozdziale będzie sieć porównująca i jej szczególny przypadek sieć sortująca. Naszym celem głównym jest opisanie kilku rozwiązań sieci sortujących.

Definicja sieci porównującej. Sieć porównująca to układ cyfrowy zbudowany z odpowiednio połączonych komparatorów i rejestrów taktowanych zegarem.

W tym podrozdziale będziemy rozważać tylko sieci porównujące zbudowane jedynie z 2 rodzajów komórek podstawowych: komparatorów i rejestrów. Układy podstawowe sieci porównujących pokazane są na rys. 1. Układ podstawowy porównujący czyli komparator realizuje operację

$$c = \min(a,b)$$
$$d = \max(a,b)$$

W sieci porównującej wszystkie porównania wykonywane są jednocześnie a przepływ danych synchronizowany jest zegarem. W sieci porównującej dane wejściowe w postaci ciągu liczb $(a_1, a_2, ...a_n)$ podawane są równolegle a dane wyjściowe $(b_1, b_2, ..., b_n)$ wyprowadzane są również równolegle. Uzyskujemy w ten sposób układ pracuje równolegle i potokowo (parallel processing i pipelining). Jeśli sieć porównująca sortuje dane wejściowe to nazywamy ją **siecią sortującą lub układem sortującym**.

Prosta sieć porównująca będąca jednocześnie siecią sortującą pokazana jest na rys. 2.

Będzie nas interesował czas sortowania (czyli ilość warstw układu sortującego). Czas sortowania to opóźnienie wprowadzane przez układ sortujący wyrażane w umownych jednostkach. Oczywiście szczególnie interesują nas układy wprowadzające najmniejsze opóźnienie.

Warto zauważyć, że właściwie przepustowość układu (ang. throughput) układów sortujących jest zawsze taka sama: jeden posortowany ciąg na takt zegara ale wprowadzane opóźnienia (ilość warstw sieci sortującej) mogą być różne.

Sieci sortujące omawiane w tym rozdziale w zasadzie należą do kategorii układów systolicznych (ang. systolic array). Układy systoliczne charakteryzują się 4 cechami

1. przetwarzaniem równoległym (paralel processing) i potokowym (pipelining)
2. modularnością
3. regularnością struktury
4. lokalnością połączeń

Dla niektórych opisywanych rozwiązań sieci sortujących warunek 4 będzie jednak naruszony.

Okazuje się, że można zbudować sieć sortującą n-liczb za pomocą sieci porównującej w czasie mniejszym niż liniowy (w czasie $O(\ln n)^2$).

Systoliczne sieci sortujące są ważnym z praktycznego punktu widzenia układami służą np. do realizacji filtrów medianowych i kwantylowych w cyfrowym przetwarzaniu sygnałów.

Warto również podkreślić, że systoliczne sieci sortujące mogą być realizowane jako program o dużym stopniu zrównoleglenia. Mówimy więc często omawiając sieci sortujące o *sortowaniu równoległym*.

Algorytmów sortowania jest dużo. Nie wszystkie jednak dobrze nadają się do przerobienia na sieci sortujące i jak mówimy w żargonie „systolizacji".

Jako sieci systoliczne dają się łatwo zrealizować następujące cztery algorytmy sortowania:

1. algorytm *bubblesort* - sortowanie bąbelkowe

2. algorytm *modified bubblesort* – zmodyfikowane sortowanie bąbelkowe

3. algorytm *insertion sort* – sortowanie przez wstawianie

4. algorytm *merge sort* – sortowanie przez scalanie

W przypadku algorytmu *merge sort* udaje się zrealizować układ systoliczny ale z naruszeniem zasady lokalności połączeń.

Systoliczna sieć sortująca oparta na algorytmie bubblesort

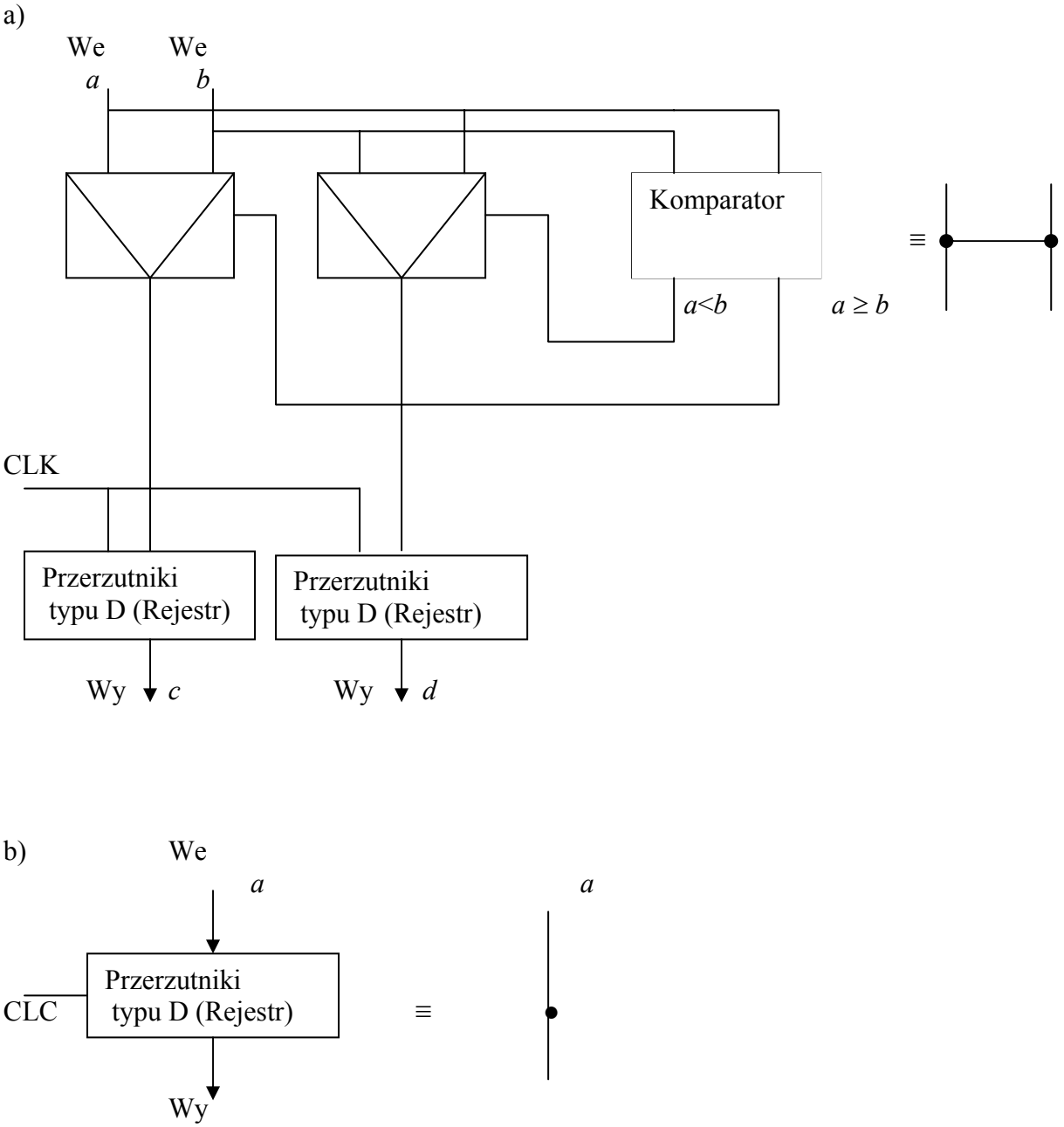
Łatwo można skonstruować sortującą sieć systoliczną realizującą algorytm bubblesort. Na rys. 3 pokazana jest systoliczna sieć sortująca ciąg 8 wyrazowy oparta na wykorzystaniu algorytmu bublesort. Sieć ma 2n-3 warstwy a więc sortuje ciąg wejściowy w czasie liniowym tzn. w czasie $\Theta(n)$. Poprawność działania sieci wynika bezpośrednio z poprawności działania algorytmu bubblesort.

Systoliczna sieć sortująca oparta na algorytmie modified bubblesort

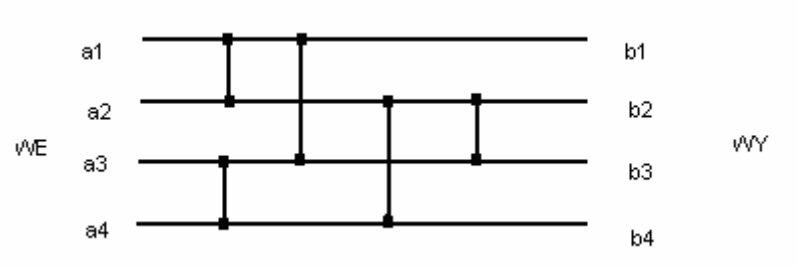
Na rys. 3.18 pokazana jest systoliczna sieć sortująca wykorzystująca algorytm modified bubblesort. Sieć sortuje ośmioelementowy ciąg skończony A[1],A [2],...,A[8] liczb w kodzie NKB. Na wejście układu podawane są w takt zegara słowa binarne o długości *w*. Ogólnie rzecz biorąc sieć ma *n*-1 warstw dla *n* parzystego i *n* dla *n* nieparzystego, gdzie *n* jest długością skończonego ciągu sortowanego. Sieć sortuje więc ciąg wejściowy w czasie liniowym $\Theta(n)$. Przedstawiona na rys. 6 sieć sortująca nosi również nazwę „sieci sortującej typu odd-even" lub „sieci sortującej odd-even Batchera".

Dowód poprawności działania sieci wykorzystująca algorytm modified bubblesort jest nieco złożony (por. [7]). Korzystamy w nim z tzw. zasady 0-1.

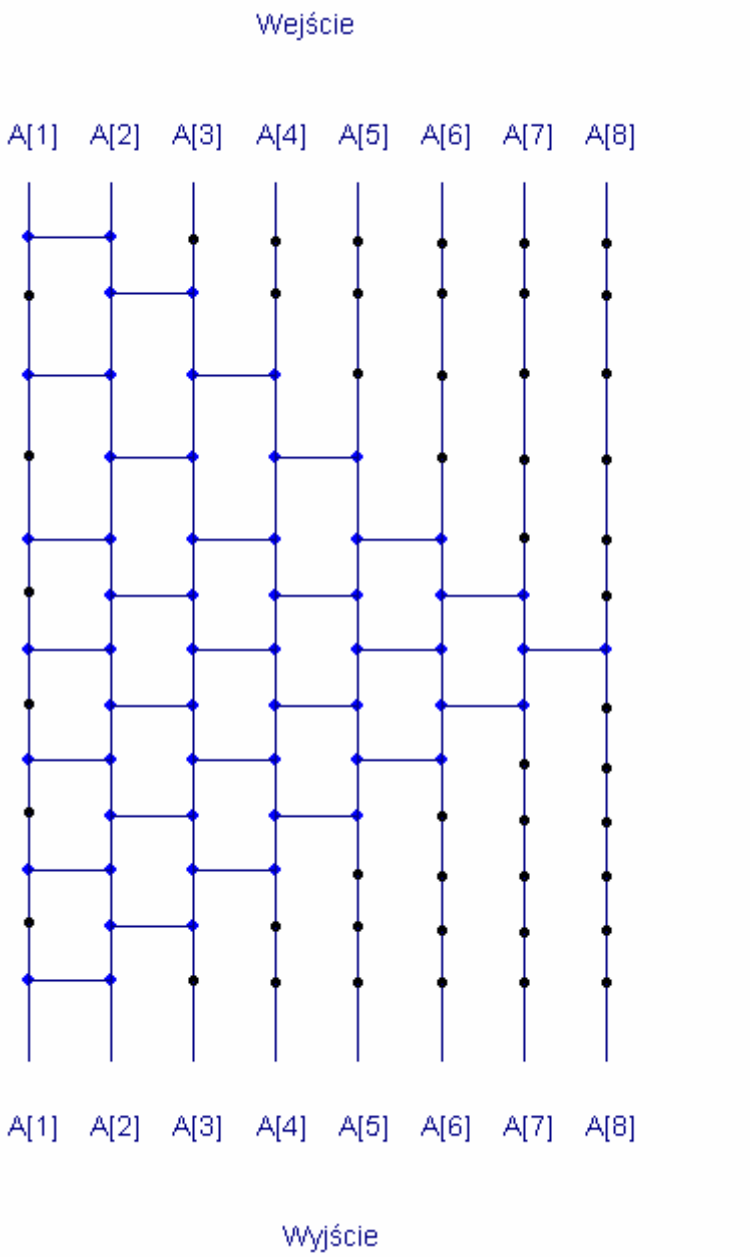
Warto zwrócić uwagę na lokalność i planarność połączeń w układzie. Lokalność i planarność połączeń ułatwia konstrukcję odpowiedniego układu scalonego realizującego algorytm.



Rys.3.16 Układy komórek podstawowych PC (ang. processing cell) w sieci sortującej . Po prawej stronie rysunku pokazany jest symbol uproszczony komórki podstawowej. Komórka z rys. a) jest komparatorem porównuje i ewentualnie zmienia porządek danych wejściowych wprowadzając jednocześnie opóźnienie o jeden takt zegara CLC. Komórka z rys. b) to rejestr, który tylko opóźnia dane wejściowe o jeden takt zegara.



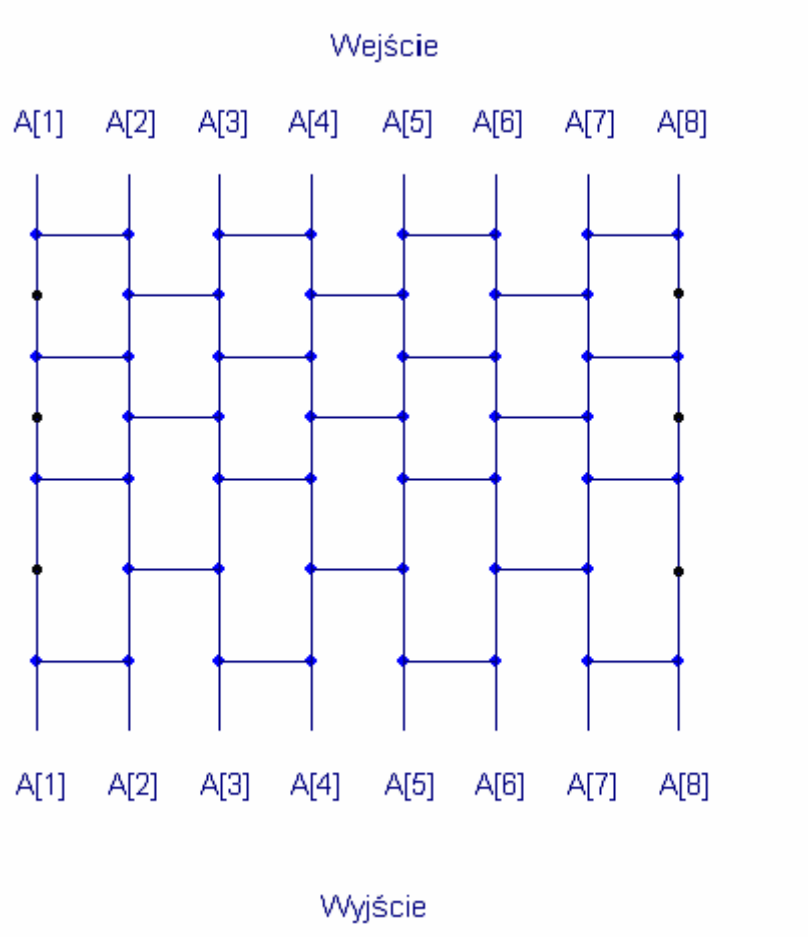
Rys. 3.17 Przykładowa 4 wejściowa sieć porównująca, która jest jednocześnie siecią sortującą



Rys. 3.18 Implementacja algorytmu sortowania bąbelkowego (bubblesort) w postaci sortującej sieci systolicznej złożonej z $m = 2 \cdot n - 3$ warstw. Na rysunku pokazana jest sieć sortująca ciąg A[1],A[2],...,A[8] (n=8 a liczba warst $m = 2 \cdot 8 - 3 = 13$).

Zasada zerojedynkowa Bardzo użyteczna w dowodzeniu poprawności działania układów sortujących jest tzw. zasada zerojedynkowa.

Zasada 0-1 -kowa mówi, że jeśli sieć sortująca działa poprawnie dla wszystkich ciągów składających się tylko z 0 i1 to poprawnie sortuje wszystkie wartości.



Rys. 3.19 Sieć systoliczna sortująca (ośmioelementowy ciąg skończony A[1],A[2],...,A[8] liczb w kodzie NKB) oparta na algorytmie "modified bubblesort". Na wejście podawane są słowa binarne o długości *w*. Ogólnie rzecz biorąc sieć ma *n*-1 warstw dla *n* parzystego i dla *n* nieparzystego, gdzie *n* jest długością skończonego ciągu sortowanego.

Sieć sortująca przez scalanie

Sieć sortująca przedstawiona poniżej jest równoległą wersją algorytmu sortowania przez scalanie. Skonstruujemy ją w 4 krokach

1. krok – konstrukcja tzw. sieci półczyszczącej (układ half-cleaner(n))
2. krok – konstrukcja bitonicznej sieci sortującej (układ bitonic-sorter(n))
3. krok – konstrukcja sieci scalającej (układ merger(n))
4. krok – z sieci scalających budujemy sieć sortującą n wartości w czasie $(\ln n)^2$ (układ sorter(n))

Definicja ciągu bitonicznego. Ciągi bitoniczne to takie ciągi skończone $f :< 1, n >\to N$, które najpierw są nierosnące a potem niemalejące albo najpierw niemalejące a potem nierosnące. Ciąg niemalejący lub nierosnący jest także bioniczny.

Przykład 1. Ciągi (1,4,6,8,3,2), (6,9,4,2,3,5), (9,8,3,2,4,3) są ciągami bitonicznymi. Oczywiście również każdy ciąg skończony jednoelementowy, dwuelementowy lub 3 elementowy jest bitoniczny.

Przykład 2. Zerojedynkowe ciągi bitoniczne mają bardzo prostą strukturę: $0^i1^j0^k$ albo $1^i0^j1^k$ gdzie $i, j, k \geq 0$.

Pokażemy, jak zbudować sieć sortującą ciągi bitoniczne za pomocą szeregowo połączonych tzw. sieci półczyszczących

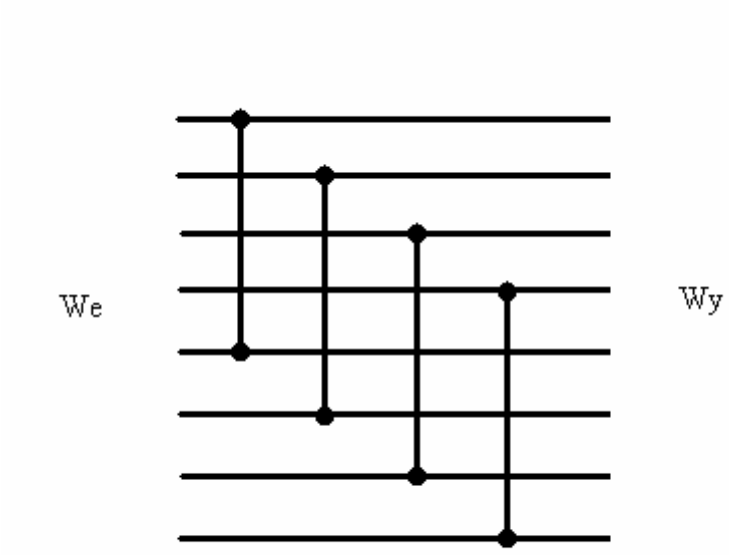
Definicja sieci półczyszczącej (ang. half-cleaner)

Sieć półczyszcząca to sieć porównująca o n wejściach i n wyjściach w której i-te wejście połączone jest komparatorem z $i+\frac{n}{2}$ wejściem (zakładamy tu, że liczba wejść $n \in N$ jest liczbą parzystą.

Sieci półczyszczące z n argumentami wejściowymi oznaczamy symbolem half-cleaner(n). Na rys. 3.20 pokazana jest sieć półczyszcząca half-cleaner(8).

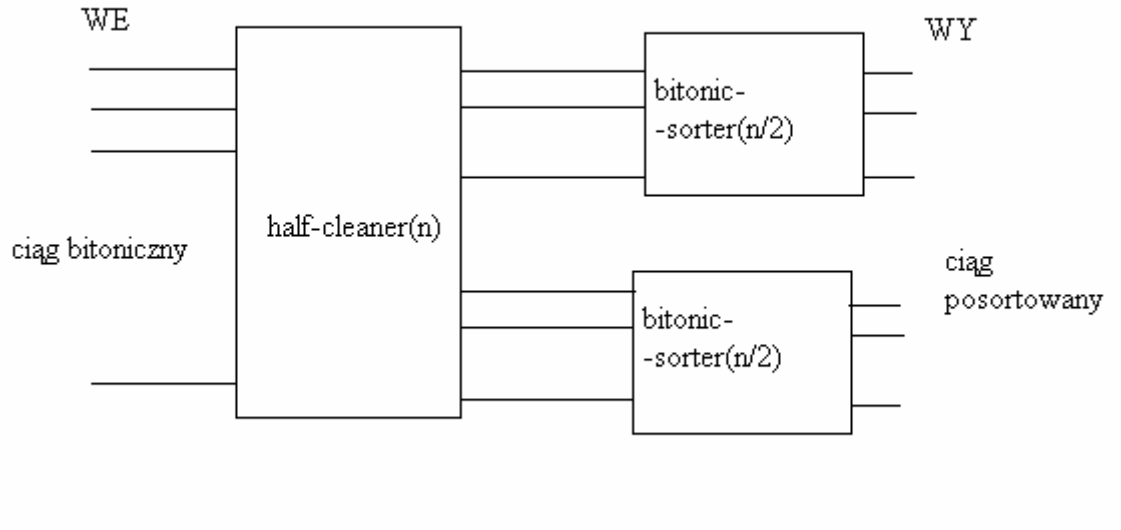
Fakt. Jeśli na wejście sieci półczyszczącej jest podany bioniczny ciąg zer i jedynek to ciąg wyjściowy ma następujące własności.

- obie połowy górna i dolna są bitoniczne
- każdy element w górnej połowie jest nie większy niż dowolny element w dolnej połowie
- co najmniej jedna połowa jest oczyszczona (oczyszczona tzn. z definicji składa się z samych zer lub samych jedynek).



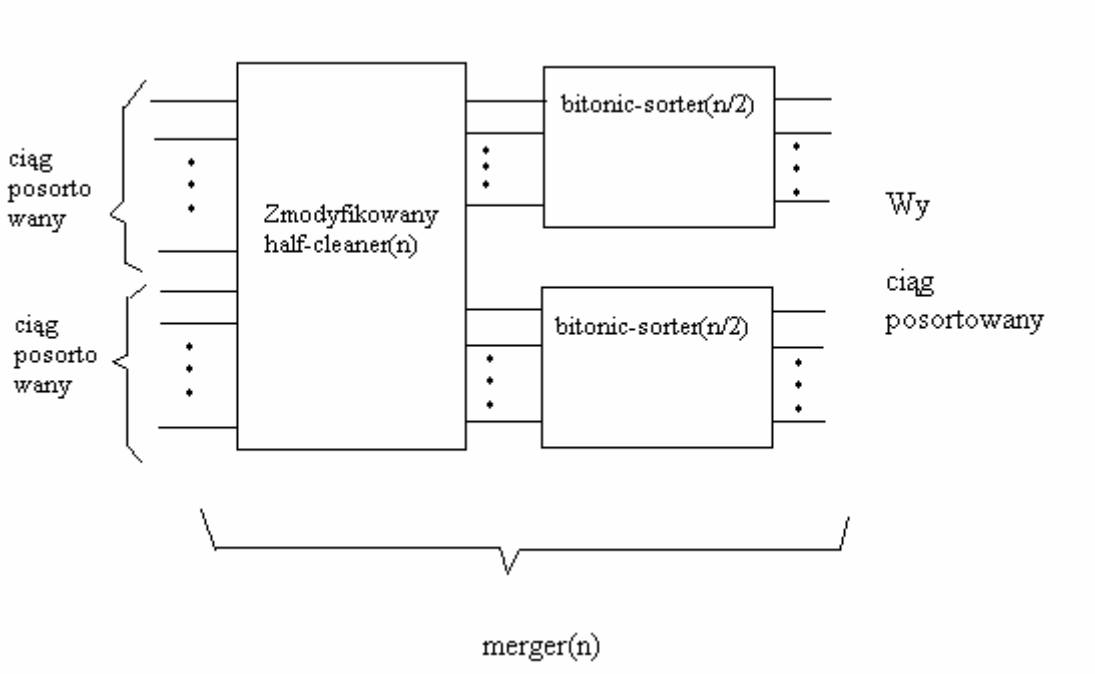
Rys.3.20 Sieć porównująca z 8 wejściami typu half-cleaner(8).

Bitoniczna sieć sortująca. Załóżmy, jak poprzednio, że liczba wejść sieci jest równa $n = 2^k$. Bitoniczna sieć sortująca bitonic-sorter(n) składa się z sieci półczyszczącej i 2 sieci bitonic-sorter(n/2) por. rys. 3.21. Stosując ten pomysł na bitoniczną sieć sortującą do układów bitonic-sorter(n/2) rekurencyjnie uzyskujemy bitoniczną sieć sortującą składającą się z $\log_2 n$ warstw. Bitoniczna sieć sortująca bitonic-sorter(n) pokazana jest na rys.3.21.

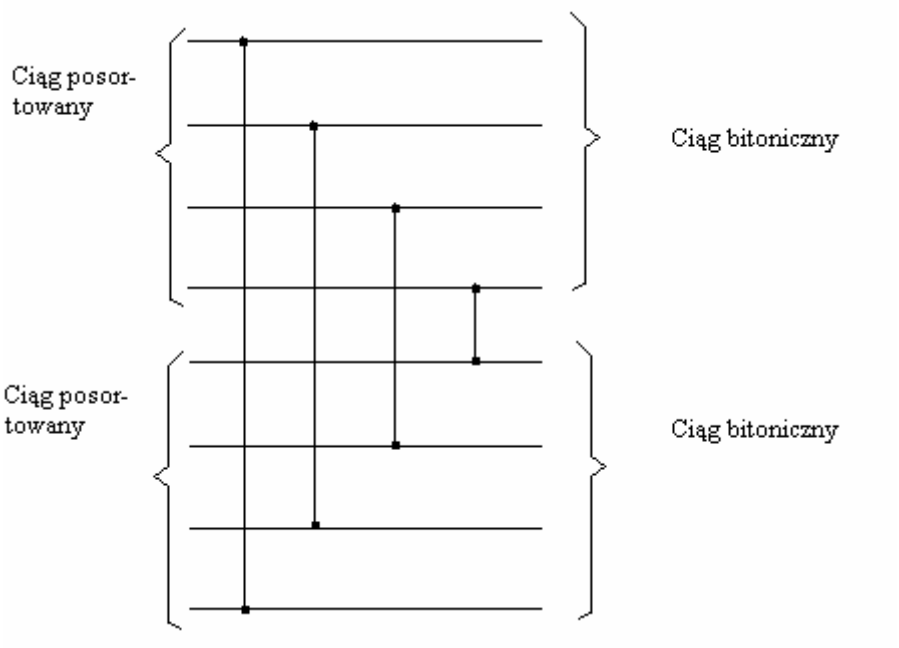


Rys. 3.21. Metoda konstrukcji układu bitonic-sorter(n) z jednego układu half-cleaner(n) i 2 układów bitonic-sorter(n/2). Powtarzając konstrukcję rekurencyjnie pozbywamy się układów bitonic-sorter(n/2).

Sieci scalające. Sieć scalająca merger(*n*) służy do scalania dwóch posortowanych ciągów wejściowych $a_1, a_2, ..., a_{n/2}$ i $b_1, b_2, ..., b_{n/2}$ w jeden posortowany ciąg wyjściowy $c_1, c_2, ..., c_n$. Sieć merger(*n*) uzyskamy jako modyfikację układu bitonic-sorter(*n*). Istota rzeczy polega na małej modyfikacji układu half-cleaner(*n*). Sieć merger(*n*) pokazana jest na rys. 3.22.

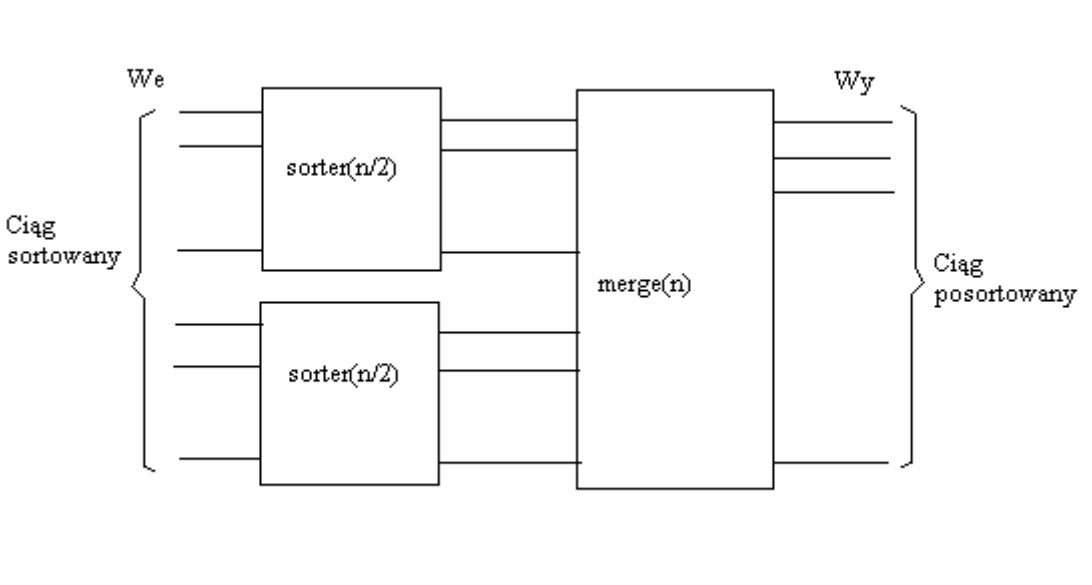


Rys.3.22. Sieć merger(n) scalająca 2 posortowane ciągi o długości n/2

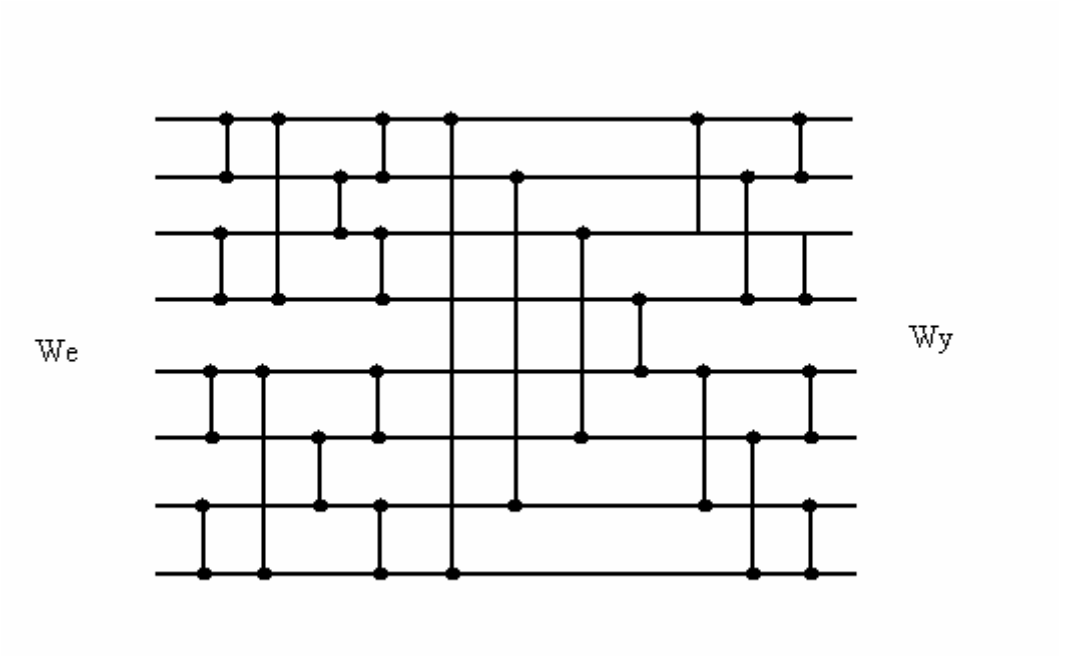


Rys.3.23. Zmodyfikowana sieć half-cleaner(n). Pierwsza warstwa sieci scalającej merger(n).

Sieć sortująca Sieć sortująca sort(n) oparta na algorytmie mergesort pokazana jest na rys.3.24 . Zakładamy tak jak poprzednio, że n jest potęgą 2. Sieć ta sortuje n-elementowy ciąg wejściowy w czasie $O(\lg^2 n)$. Stosując rekurencyjnie pokazaną na rys.3.24 koncepcję zastępujemy każdy układ sorter(n/2) dwoma układami sorter(n/4) i jednym układem scalającym merge(n/2). W ten sposób rekurencyjnie tworzymy sieć sortującą z układów typu merge(n) wykorzystując pomysł zaczerpnięty z klasycznego algorytmu mergesort. Widać, że do konstrukcji układu potrzebnych jest $\lg n$ warstw układów scalających typu merge(n). Znając liczbę warstw układu merge(n) możemy teraz ocenić liczbę warstw układu sorter(n). Rozwiązując odpowiednie równanie rekurencyjne stwierdzamy, ż układ sorter(n) sortuje ciąg wejściowy w czasie $O(\lg^2 n)$. Sieć sortująca sort(8) pokazana jest na rys.3.25.



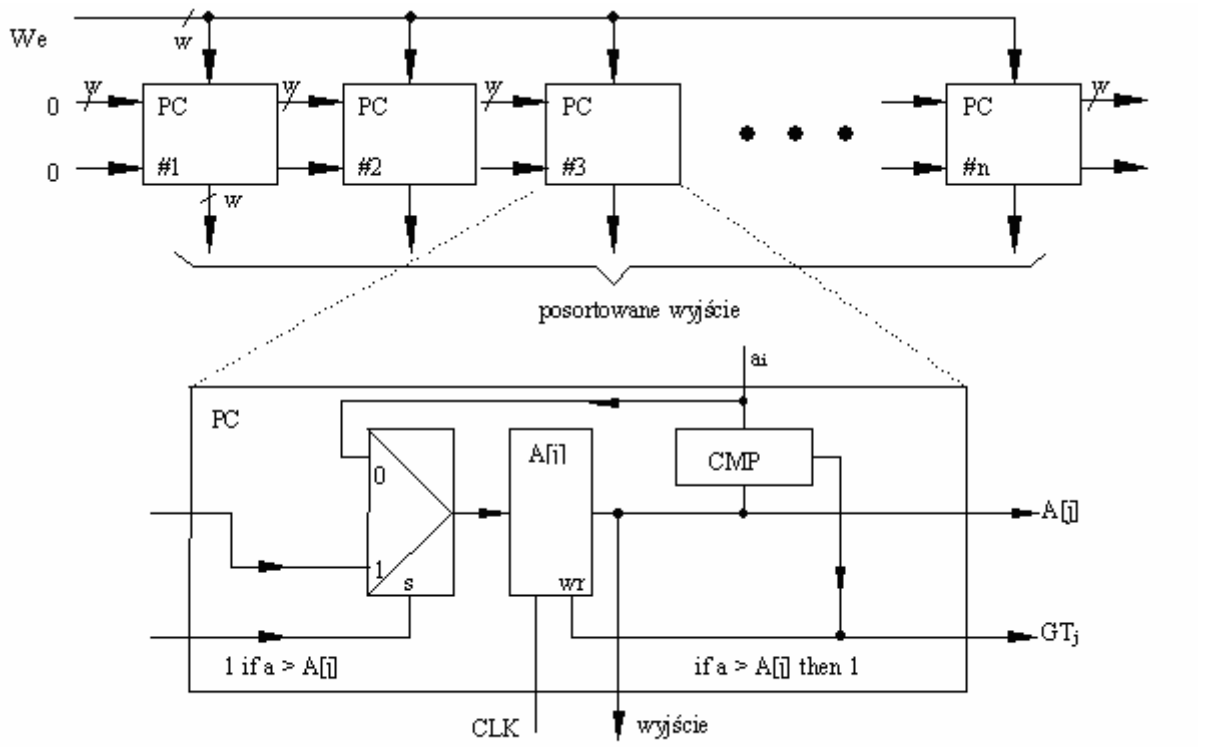
Rys. 3.24. Układ sorter(n). Widać, że do konstrukcji układu potrzebnych jest $\lg n$ warstw układów scalających typu merge(n).



Rys.3.25 Sieć sortująca sorter(n) zbudowana z 3 warst sieci scalających i 6 warstw komparatorów co daje późnienie 6 taktów zegara

Sieć systoliczna sortująca oparta na algorytmie insertionsort

Na rys. 3.26 pokazana jest systoliczna sieć sortująca oparta na algorytmie insertionsort. Jest to sieć z szeregowym wejściem i równoległym wyjściem zbudowana z szeregowo połączonych komórek elementarnych PC. Na wejście podawane są równolegle (w takt zegara CLC) słowa binarne o długości *w* reprezentujące kolejne wyrazy ciągu sortowanego. Liczby zapisywane są w kodzie NKB.



Rys. 3.26 Sieć systoliczna sortująca (ciąg skończony o długości *n*) oparta na algorytmie "insertion sort". Na wejście podawane są równolegle słowa binarne o długości *w*. Liczby zapisywane są w kodzie NKB.

Literatura

[1] D.E.Knuth; Sztuka programowania - tom 3; WNT, Warszawa 2003

[2] T.H. Cormen, C.E.Leiserson, R.L.Rivest, C.Stein; Wprowadzenie do algorytmów; WNT, Warszawa 2004.

[3] L.Banachowski, K.Diks, W.Rytter; Algorytmy i struktury danych; WNT, Warszawa 1996.

[4] A.Aho, J.E.Hopcroft, J.D.Ullman; Projektowanie i analiza algorytmów; Helion, 2003.

[5] R. Neapolitan i K.Naimpour; Podstawy algorytmów z przykładami w C++; Hellion 2004.

[6] H.Schildt; C; Wydawnictwo RM; Warszawa 2002.

[7] P.Wróblewski; Algorytmy, struktury danych i techniki programowania; Helion 2003.

[8] T.Adamski; Niestałość chwilowa w układach i systemach elektronicznych; Instytut Systemów Elektronicznych, Wydział Elektroniki P.W. , warszawa 1992.

[9] A.Drozdek, D.L.Simon; Struktury danych w języku C; WNT, 1996.

[10] E.Riengold, J.Nievergelt, N.Deo; Algorytmy kombinatoryczne; PWN, 1985.

Zadania

Zadanie 1
Pokazać, że algorytm *countsort* jest stabilny.

Zadanie 2
Jaka jest największa liczba elementów w kopcu o wysokości $h \in N$. Jaka jest najmniejsza liczba elementów w kopcu o wysokości h.

Zadanie 3
Wykazać, że n elementowy kopiec ma wysokość $\lfloor \log_2 n \rfloor$.

Zadanie 4
Sprawdzić czy tablica $A[1..11]$ zadana następująco $A = (15,13,9,5,12,8,7,4,0,6,2,1)$ reprezentuje kopiec.

Zadanie 5
Mamy dany kopiec $A = (15,13,9,5,12,8,7,4,0,6,2,1)$. Pokazać kolejne stany tablicy przy realizacji procedury *heap-extract-max*(A).

Zadanie 6
Zilustrować działanie procedury *partition* algorytmu *quicksort* dla tablicy $A = (13,19,9,5,12,8,7,4,11,2,6,21)$

Zadanie 7
Mamy 4 algorytmy sortowania
- sortowanie przez wybór
- sortowanie przez wstawianie
- sortowanie bąbelkowe
- zmodyfikowane sorowanie bąbelkowe

Który z nich wykona się szybciej jeśli dane wejściowe są już uporządkowane.

Zadanie 8
Opisz pomysł wykorzystywany w algorytmie sortowania przez zliczanie (*countsort*). Jaka jest pesymistyczna złożoność czasowa tego algorytmu. Opisz rozwinięcie tego algorytmu do algorytmu *radixsort*.

Zadanie 9
Co to jest sortowanie zewnętrzne. Jakie algorytmy stosujemy do sortowania zewnętrznego.