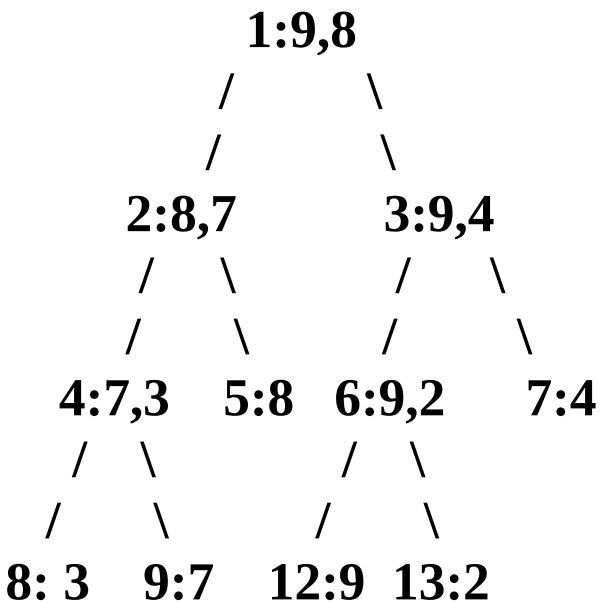
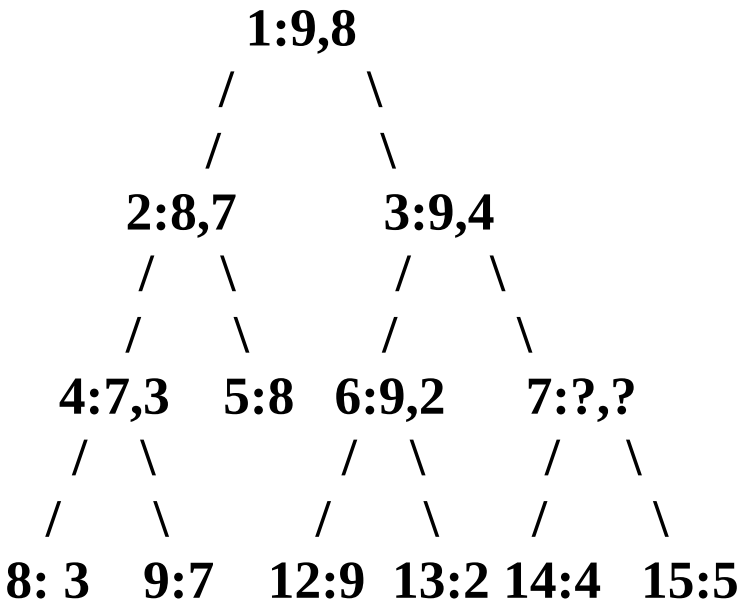
Wykład: Sortowanie III

Drzewa Turniejowe

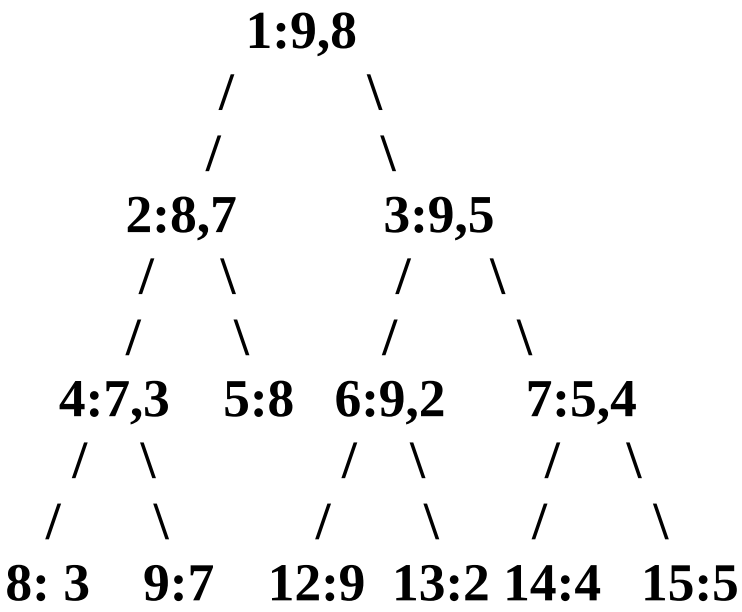


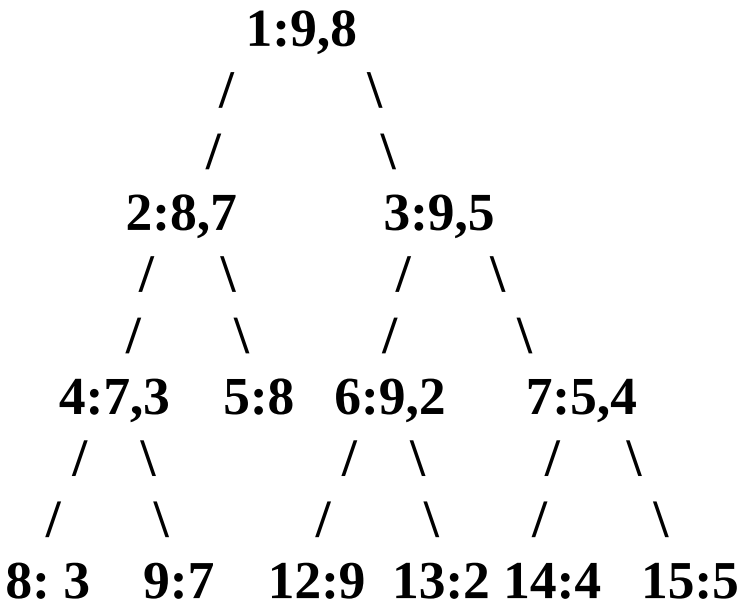
Insert(x,S)

1) tworzymy dwa nowe liście na ostatnim poziomie,
2) do jednego wstawiamy x a do drugiego wartość z liścia, np. val(i), z poziomu sąsiedniego,
3) oba liście stają się następnikami i,
4) aktualizujemy wartości na ścieżce do korzenia.


1:9,8
2:8,7
3:9,4
4:7,3
5:8
6:9,2
7:?,?
8: 3
9:7
12:9
13:2
14:4
15:5

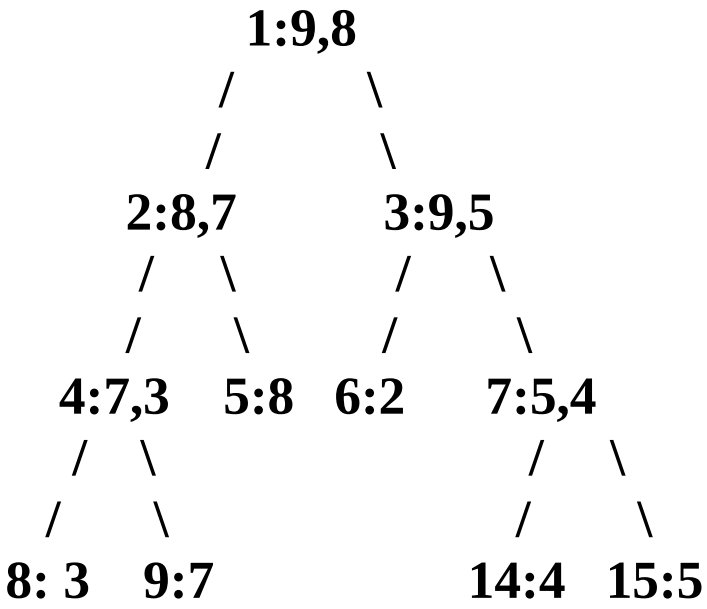
Insert(5,S)


1:9,8
2:8,7
3:9,5
4:7,3
5:8
6:9,2
7:5,4
8: 3
9:7
12:9
13:2
14:4
15:5

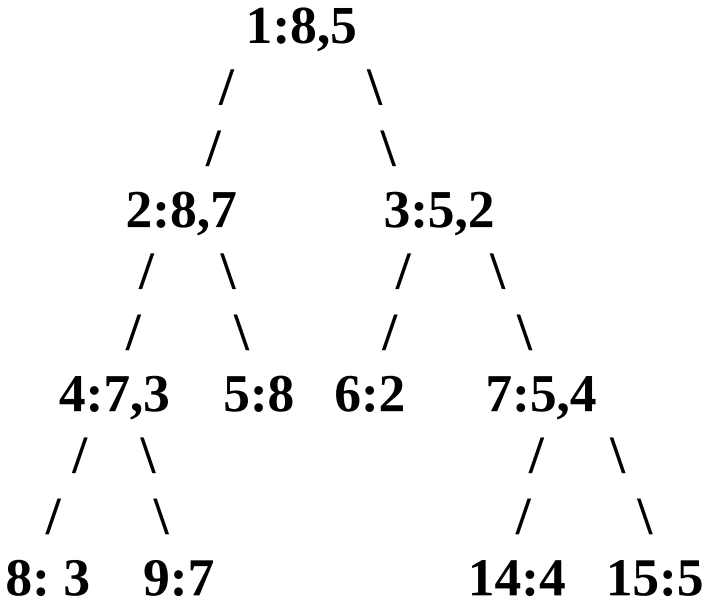


Deletmax(S):

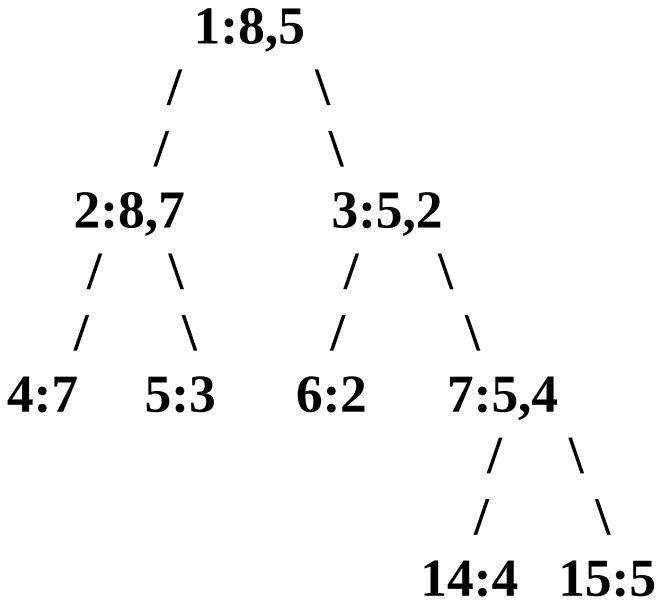
1) usuwamy liść zawierający największy element (mx(1)),
2) jeśli mx(1) nie należy do ostatniego poziomu, to w jego miejsce wstawiamy dowolny element z ostatniego poziomu mniejszy od sąsiada,
3) aktualizujemy wartości na ścieżce od mx(1) do korzenia.



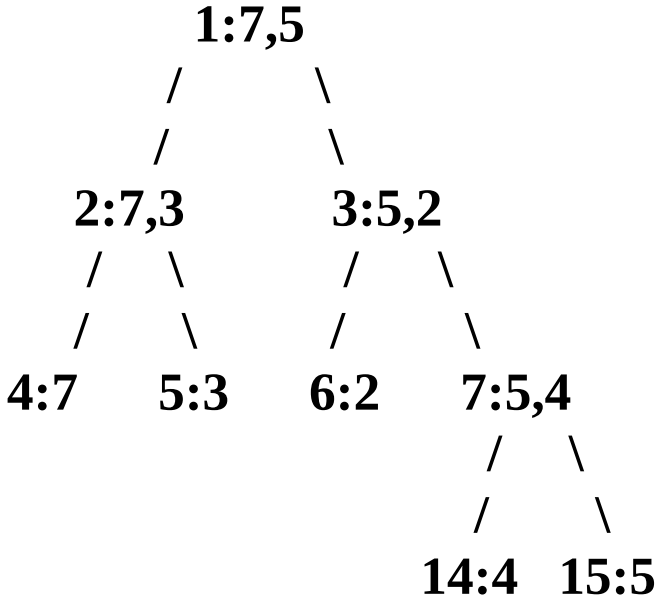
Po usunięciu największego elementu.



Po aktualizacji.



Po usunięciu największego elementu.



Po aktualizacji.

T_deletmax = |log n| - 1 = O(log n)

Sortowanie pozycyjne

Binarna reprezentacja danych w komputerze nie jest brana pod uwagę w ogólnych algorytmach sortowania.

Również w językach programowania wysokiego poziomu nie ma operacji na liczbach binarnych.

```
function bits(x, k, j : integer): integer,
begin
    bits := (x div 2^k) mod 2^i
end bits;
```

Funkcja bits wycina j bitów, poczynając od bitu o numerze k z prawej strony.

Sortowanie pozycyjnego - metoda liczników częstości.

Załóżmy, że $b = 2^n$.

Dla każdego j = 0, l, ... ,b - l liczymy, ile razy j pojawia się w ciągu wejściowym a[1], ... ,a[n].

Na podstawie obliczonych liczników częstości umieszczamy każdy element tablicy a, we właściwym miejscu w ciągu wynikowym.

```
procedure countsort,
{a[l. . n]. t[l. . n]. count[0 .. m -l]}
var i, j, p: integer;
begin
    for j := 0 to m-l do count[j] := 0; {inicjowanie}
    for i := 1 to n do count[a[i]] := count[a[i]] + 1;
   {count[j] to liczba wystąpień liczby j}
    for j := 1 to m-l do count[j] := count[j-1] + count[j];
   {count[j] to liczba wystąpień elementów <= j}
for i := n downto l do
   begin
       p:= a[i];
       t[count[p]] := p;
       count[p] := count[p] - 1
   end;
for i := 1 to n do a[i] := t[i]
end countsort;
```

Przykład:

q = [1,3,1,2,2,1,3]

1sza iteracja
2ga iteracja count[1] = 3, count[2] = 2, count[3] = 2,
3cia iteracja count[1] = 3, count[2] =5, count[3] = 7,

1szy element zajmie pozycje a[1...count[1]]
2gi element zajmie pozycje a[count[1]+1...count[2]]
3ci element zajmie pozycje a[count[2]+1...count[3]]

Wypisywanie elementów do tablicy

- - - - - - 3
- - 1- - - 3
- - 1- 2- 3
- - 122 -3
- 1122 -3
- 112233
1112233

Analiza -algorytmu countsort jest bezpośrednia:

$T(n, m) = A(n, m) = O(n + m)$

$S(n, m) = n + m + 0(1)$

Algorytm jest szybki, gdy $m = O(n)$

Algorytm jest stabilny.

Countsort można stosować tylko dla niedużych wartości m (tablica count ma rozmiar m !).

Dla większych wartości m można stosować podobną metodę, ale z dzieleniem liczb do posortowania na części, na których algorytm countsort może działać.

Dzielimy słowa o b bitach na b/e grup e bitowych.

1) sortujemy ciąg liczb, stosując algorytm countsort względem ostatniej (najmniej znaczącej) grupy bitów.

2) sortujemy ciąg liczb względem przedostatniej grupy bitów, itd. …

b/e) sortujemy ciąg liczb względem pierwszej grupy bitów.

Istotna jest stabilność algorytmu countsort.

O pozycji elementu decyduje pierwszych e bitów;
jeśli są one takie same, to decydujące znaczenie mają dopiero następne grupy bitów, które zostały też posortowane.

```
procedure radixsort;
 {m=2^e, a, t[l .. n],count[0 .. m-l]}
var i, j, pass, nofpasses, key: integer;
begin
    nofpasses := b div e;
    if nofpasses * e := b then nofpasses := nofpasses - 1;
    for pass: = 0 to nofpasses do
    begin
        for j :=0 to m-1 do count[j] :=0;
        for i : = l to n do
        begin
            key := bits(a[i], pass*e, e);
            count[key] := count[key] + 1
       end;
       for j:=1 to m-l do count[j]:= count[j-1] + count[j];
       for i := n downto 1 do
       begin
           key:= bits(a[i], pass * e, e);
           t[count[key]] := a[i];
          count[key] := count[key]-l
      end;
      for i := 1 to n do a[i] := t[i]
   end
end radixsort;
```

T(n,m) = A(n,m) = O(n+m), gdy b/e = O(1)
S(n,m) = n+m+O(1)

Przykład:

b = 8, e = 2, n = 4

a[l] = 011100 10
a[2] = 110111 01
a[3] = 100000 00
a[4] = 011010 01

pass = 0

a[l] = 1000 00 00
a[2] = 1101 11 01
a[3] = 0110 10 01
a[4] = 0111 00 10

pass = 1

a[l] = 1000 00 00
a[2] = 0111 00 10
a[3] = 0110 10 01
a[4] = 1101 11 01

pass = 2

a[l] = 10 00 00 00
a[2] = 11 01 11 01
a[3] = 01 10 10 01
a[4] = 01 11 00 10

pass = 3

a[l] = 01 10 10 01
a[2] = 01 11 00 10
a[3] = 10 00 00 00
a[4] = 11 01 11 01

Zalety Radixsort:

Liniowa złożoność czasowa przy odpowiednich założeniach dotyczących n, tzn. m = O(n).

Dla średnich wartości ‘n’ Radixsort może być szybszy niż quicksort.

Po modyfikacjach może być użyty do sortowania:

- słów w porządku alfabetycznym
- liczb rzeczywistych z pewnego przedziału

Wady:

- Algorytm jest wolny dla małych rozmiarów ‘n’,
- Duże zużycie pamięci dla dużych ‘n’.

Sortowanie tablic przez łączenie:

Mamy dany ciąg rekordów: r_1, ..., r_n typu

```
type rek = rekord
      klucz : typ_klucza;
      pole_1: typ_pola_1;
      ...
      pole_m : typ_pola_m
end
```

Zakładamy, że typ_klucza jest uporządkowany liniowo (np. tak jak jest dla typów prostych).

Naszym zadaniem jest takie przestawienie elementów ciągu tak, by otrzymany ciąg r_{p1},..., r_{pn} miał własność:
r_{p1}.klucz <=.. .<= r_{pn}.klucz

Idea algorytmu polega na tym, że ciąg dzieli się na uporządkowane fragmenty r_i, ..., r_j spełniające:

r_k.klucz <= r_{k+1}.klucz dla k=i,...,j-1
r_{k-1}.klucz > r_k.klucz dla k=i,
r_k.klucz > r_{k+1}.klucz dla k=j

Ciągi r_i,...,r_j będziemy nazywać seriami.

Rozważmy ciąg:
43, 18, 21, 30, 13, 52, 51, 75, 80, 62
zawiera następujące serie:

- 43
- 18, 21, 30
- 13, 52
- 51, 75 80
- 62

Po zdefiniowaniu serii wybieramy dwie serie, np.

- 18, 21, 30
- 13, 52

łącząc je otrzymujemy nową serię w następujący sposób:

13	18, 21, 30
	52

13,18	21, 30
	52

13,18,21	30
	52

13, 18, 21,30	
	52

13, 18, 21,30,52	

Na tej podstawie można ułożyć następujący algorytm (oznaczmy go *): dane są dwie tablice

x: array[1..m] of rek
y: array[1..n] of rek.

Rezultatem ich łączenia jest
z: array[1..n+m] of rek

Algorytm abstrakcyjny tej operacji jest następujący:

```
ustal wartości początkowe indeksów;
while (i<=m) and (j<=n) do
  begin
    if element serii x <= elementu serii y then
        begin
            dopisz do tablicy z element serii x
            zwiększ indeks tablicy x  i z
        end else
        begin
            dopisz do tablicy element serii y
            zwiększ indeks tablicy y i z
        end
    end
    dołączenie końca serii x;
    dołączenie końca serii y
```

Rozwijając poszczególne instrukcje abstrakcyjne możemy otrzymać następujący program:

```
i:=1; j:=1; k:=1;
while (i<=m) and (j<=n) do
begin
    if x[i].klucz<=y[j].klucz then
    begin
        z[k]:=x[i]; k:=k+1; i:=i+1
    end else
    begin
        z[k]:=y[j]; k:=k+1; j:=j+1
    end
end
dołączenie końca serii x;
dołączenie końca serii y;
```

Uściślenie operacji dołączania końca jest następujące:

```
{dołączenie końca serii x;}
    while i <=m do
    begin
        z[k]:=x[i];
        k:=k+1; i:=i+1;
    end;
{dołączenie końca serii y}
    while j<=n do
    begin
        z[k]:=y[j];
        k:=k+1; j:=j+1;
    end;
```

Czyli powinniśmy łączyć coraz dłuższe serie, aż do otrzymania dwóch serii, a połączenie tych serii da jeden ciąg posortowany. Sugeruje to następującą ideę:

- mamy ciąg danych,
- wybieramy dwie serie
- łączymy je ze sobą,
- powtarzamy operacje aż będzie jedna seria.

Za pomocą abstrakcyjnego algorytmu możemy zapisać to następująco:

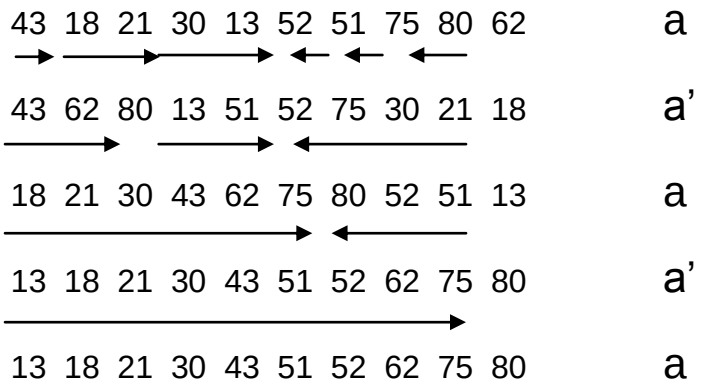
repeat
    Podział ciągu;
    Łączenie ciągów
until jedna seria

Należy zauważyć, że ciągi mogą być w postaci tablicy, listy lub plików.
Rozważamy tablice, ale sam algorytm będzie zależał od reprezentacji. Ponieważ dane są zapisane w tablicy, to możemy uzyskiwać bardzo łatwo do nich dostęp. Niech tablica będzie postaci:

a : array[1..n] of rek.

Jako serie możemy wybierać dowolne ciągi, ale najlepiej będzie wybierać ciągi z początku tablicy i z końca. Połączone serie będą umieszczone w tablicy a’ takiego samego typu jak i tablica a.

Proces łączenia można przedstawić następująco na przykładzie:



Wprowadzając te ustalenia otrzymujemy następujący abstrakcyjny algorytm:

apa :=false;
repeat
Ustawienie początkowe i,j,k,p;
m:=0 {liczba utworzonych serii}
Łączenie serii od i,j do k,p;
apa=not apa
until m=1
if apa then kopiowanie a’ do a

- apa – przełącznik mówiący o tym, że gdy apa = true to przenosimy dane z a’ do a i gdy apa = false to przenosimy w odwrotnym kierunku,

Powyższy algorytm może być realizowany za pomocą jednej tablicy A, ale mającej dwa razy większy rozmiar. Pierwsza część tablicy odpowiada za tablicę a[1..n], a druga za tablicę a' = a[n+1..2*n].

Przy tych założeniach
{Ustawienie początkowe i,j,k,p}
można uściślić następująco:

```
if apa then
 begin
     i:=1; j:=n; k:=n+1; p:=2*n
 end else
 begin
     k:=1; p:=n; i:=n+1; j:=2*n
end;
```

Natomiast instrukcję
{Łączenie serii od i,j do k,p}
można wyrazić jako instrukcję iteracyjną, w której podstawową czynnością w każdym powtórzeniu jest łączenie dwóch serii w jedną. Elementy wybiera się z punktów określonych przez i oraz j, a umieszcza się w na przemian w punktach określonych przez k i p.
Można tak postąpić, by program zawsze umieszczał elementy w miejscu określonym przez k. W tym celu należy zamieniać wartości k i p po połączeniu dwóch serii oraz określić kierunek zmian indeksu (gdy dodajemy do oryginalnej

wartości k to zwiększamy o +1 a gdy do p to o –1). Wprowadzamy zmienną h.

```
h:=1;
repeat
    łączenie serii od i,j w jedną k;
    h:=-h;
    m:=m+1;
    zamiana k i p;
until i=j
```

Uściślenie instrukcji
zamiana k i p;
polega na następujących czynnościach:

```
t:=k; k:=p; p:=t
```

Natomiast uściślenie instrukcji
łączenie serii od i,j w jedną k;
jest podobne do algorytmu (*) i można zapisać następująco:

```
{łączenie serii od i,j w jedną k;}
repeat
    if A[i].klucz<A[j].klucz then
    begin
        A[k]:=A[i]; i:=i+1;
        kons:=A[i-1].klucz > A[i].klucz
         (**)
    end else
    begin
        A[k]:=A[j]; j:=j-1;
        kons:=A[j+1].klucz > A[j].klucz
```

```
        end
        k:=k+h
    until kons
    Dołączenie końca serii od i;
    Dołaczenie końca serii od j
```

Gdzie kons jest zmienną, która określa, czy został osiągnięty koniec serii. Jeżeli dojdziemy do końca serii, to przerywamy pętlę repeat.

Operacje dołączania końca serii mogą być zapisane następująco (użyto w nich pętli while aby operacja dołączania serii była pomijana gdy koniec jest pusty).

```
    {dołączenie końca serii od i}
    while A[i].klucz <= A[i+1].klucz do
    begin
        A[k]:=A[i];
        k:=k+h; i:=i+1
    end
   {dołączenie końca serii od j}
    while A[j].klucz<=A[j-1].klucz do
    begin
        A[k]:=A[j];
        k:=k+h; j:=j-1
    end
```

Należy zauważyć, że w algorytmie (**) przy przetwarzaniu ostatniego elementu danej serii wskaźnik przesunie się do nowej serii. Tak więc przesuwanie powinno być wykonywane warunkowo, gdy nie został osiągnięty koniec serii (koniec serii jest wtedy, gdy zmienna kons ma wartość true).

Poprawiony algorytm jest więc postaci:

```
{łączenie serii od i,j w jedną k;}
repeat
    if A[i].klucz<A[j].klucz then
    begin
        A[k]:=A[i];
        if A[i].klucz > A[i+1].klucz
            then kons:=true
            else  i:=i+1;
     end else
    (poprawiony **)
    begin
        A[k]:=A[j]; j:=j-1;
        if A[j].klucz > A[j-1].klucz
            then kons:=true
             else  j:=j-1;
    end
    k:=k+h
until kons
Dołączenie końca serii od i;
Dołaczenie końca serii od j
```

Podany algorytm nie działa jeszcze poprawnie. Mianowicie w sytuacji, gdy serie zachodzą na siebie to pewne elementy mogą być dwukrotnie skopiowane. Aby tego uniknąć, należy zmodyfikować następująco odpowiednie instrukcje:

```
{dołączenie końca serii od i}
while (A[i].klucz<=A[i+1].klucz) and (i<j)do
begin
    A[k]:=A[i];
    k:=k+h; i:=i+1
end

{dołączenie końca serii od j}
while (A[j].klucz<=A[j-1].klucz) and (i<j) do
begin
    A[k]:=A[j];
    k:=k+h; j:=j-1
end
```

Natomiast instrukcja kopiowanie z a' do a jest postaci:

```
for i:=1 to n do
    A[i]:=A[i+n];
```

Sortowanie zewnętrzne

Elementy znajdują się w pliku zewnętrznym i do bufora pamięci wewnętrznej można zapisać maksymalnie **m** elementów.

Przez blok lub serię rozumiemy dowolną posortowaną część listy.

Algorytmy sortowania zewnętrznego składają się z dwóch głównych kroków:

1) tworzenia bloków początkowych,
2) scalania wielofazowego tak długo aż pozostanie tylko jeden blok.

Scalanie wielofazowe z 4 plikami

Bloki zostały równo rozłożone na pliki P_0 i P_1.
Pliki P_2 i P_3 są początkowo puste.

i1 := 0, i2 := 1 {otwarte do czytania}
j1 := 2, j2 := 3 {otwarte do pisania}

while jest więcej niż jeden blok **do**

1) scal pierwszy blok z P_i1 z pierwszym z P_i2 i zapisz do P_j1,
2) scal następny blok z P_i1 z następnym z P_i2 i zapisz do P_j2,
3) powtarzaj 1) i 2) aż do końca plików P_i1 i P_i2,
4) przewiń pliki do początku, dodaj 2 mod 4 do i1,i2,j1,j2, zamieniając role plików wejściowych i wyjściowych.

Algorytm wymaga log(n/m) faz, gdzie

n – liczba elementów

m – liczba elementów w bloku

Scalanie wielofazowe z 3 plikami

Mamy do dyspozycji trzy pliki: P_0, P_1, P_2.

P_0 zawiera F_k bloków a P_1 F_k+1 bloków,

Gdzie F_k to k-ta liczba Fibonacciego.

F_0 = 0
F_1 = 1
F_k+1 = F_k-1+ F_k dla k > 1

i1 := 0; i2 := 1 {pliki wejściowe}
j := 2 {plik wyjściowy}

while jest więcej niż jeden blok **do**

1) scal pierwszy blok z P_i1 z pierwszym z P_i2 i zapisz do P_j,
2) powtarzaj 1) aż do końca pliku P_i1,
3) przewiń pliki P_i1 i p_j do początku, dodaj 1 mod 3 do i1,i2,j.

Złożoność: 1.04nlog(n/m) + O(n) porównań
2 * 1.04nlog(n/m) + O(n/m) przesłań

Przykład:

P0: 8
P1: 13
P2:

P0:
P1: 5
P2: 8

P0: 5
P1:
P2: 3

P0: 2
P1: 3
P2:

P0:
P1: 1
P2: 2

P0: 1
P1:
P2: 1

P0:
P1: 1
P2:

.