Wykład 3
Rekurencyjne wywołanie podprogramu
Algorytmy sortowania tablic- sortowanie szybkie, sortowanie przez łączenie

1. Rekurencja

1.1. Obliczanie silni w sposób iteracyjny

Wzór na obliczenie n! jest następujący:

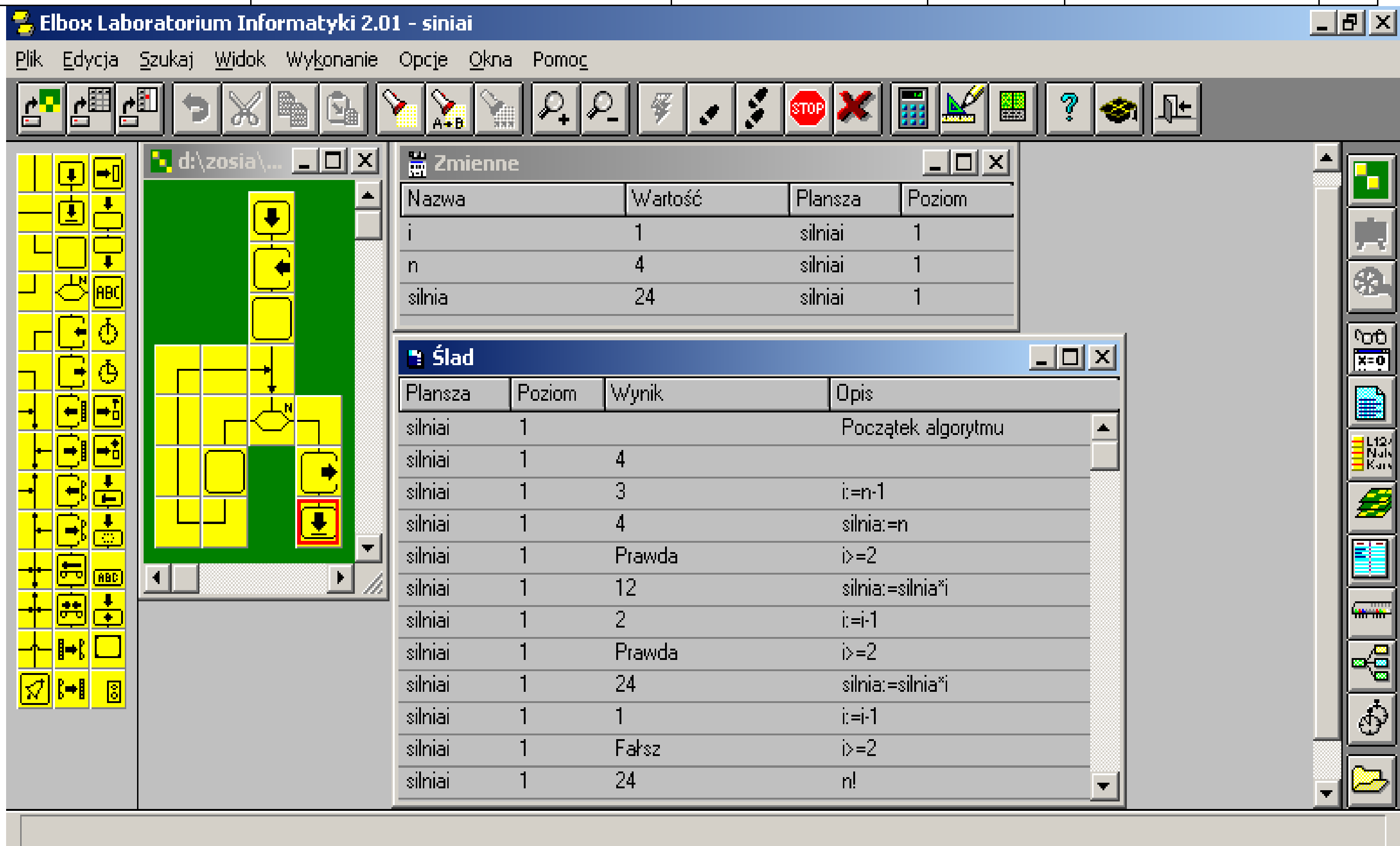
n! = n* (n-1)*(n-2)...2*1

≡ n* (n-1)*(n-2)...2,

gdzie: 2!=2, 1!=1

```
#include <stdio.h>
void main ()
 {long n=11, silnia=n;
  for (long i=n-1;i>=2;i--) silnia=silnia*i;
  printf ("%ld!=%ld", n, silnia);}
```

n = 4					
Numer iteracji	Iteracje; si = si*i, i = i-1	i, pocz: i = n-1	i>=2	si, pocz: si = n	x
		3		4	
1	x = si*i si = x i = i-1	3 2	prawda	4 12	12
2	x = si*i si = x i = i-1	2 1	prawda	12 24	24
3			falsz		



1.2. Obliczanie silni w sposób rekurencyjny

Wzór na obliczenie n! jest następujący:

n!≡ n* (n-1)*(n-2)...2 = n*(n-1)!, czyli

n! = n* (n-1)!

(n-1)!=(n-1)*(n-2)!

....

3!=3*2!

2!= 2

gdzie 2!=2, 1!=1

```
#include <stdio.h>
long silnia(long n)
 { if (n>2)  return n* silnia(n-1);
   else      return n;
 }
void main()
{  long n=4l;
   printf ("%ld!=%ld", n, silnia(n));
}
```

n = 4					
Wywołania rekurencyjne n >2 n!=n*(n-1)! Koniec wywołań rekurencyjnych: 1<=n<=2, 2!=2				Powrót: Podstawienia do **y**	Poziom wywołań f.rek.
n=4	4! = 4*(4-1)!	4!=4*3!	y = 4***y** ↓	y=4***6**=**24** ↑	1
n=3	(4-1)! = (4-1)*(4-2)!	3!=3*2!	y = 3***y** ↓	y=3***2**=**6** ↑	2
n=2	(4-2)! = 2!	2!=2	**y** = 2	**2** ↑	3

Elbox Laboratorium Informatyki 2.01 - silniare

Plik Edycja Szukaj Widok Wykonanie Opcje Okna Pomoc

Ślad

Plansza	Poziom	Wynik	Opis
silniare	1		Początek algorytmu
silniare	1	4	Podaj dane
silniare	1		Wywołanie procedury: silnia <n> -> si
siniap	2		Początek procedury: silnia <n>
siniap	2	Prawda	n>2
siniap	2		Wywołanie procedury: silnia <n-1> -> si
siniap	3		Początek procedury: silnia <n>
siniap	3	Prawda	n>2
siniap	3		Wywołanie procedury: silnia <n-1> -> si
siniap	4		Początek procedury: silnia <n>
siniap	4	Fałsz	n>2

Zmienne

Nazwa	Wartość	Plansz	
n	4	silniar	
si	?	silniar	
n	4	siniap	
si	?	siniap	
n	3	siniap	3
si	?	siniap	3
n	2	siniap	4
si	?	siniap	4

Stos

Poziom	Plansza	Kolumna	Wiersz	Wywołanie
0	silniare	1	1	Początek algorytmu
1	silniare	1	3	Wywołanie procedury: silnia <n> -> si
2	siniap	1	3	Wywołanie procedury: silnia <n-1> ->
3	siniap	1	3	Wywołanie procedury: silnia <n-1> ->

2. Sortowanie szybkie

Algorytm sortowania szybkiego należy do algorytmów „dziel i zwyciężaj".

Algorytm sortowania szybkiego - poziom konceptualny

(1) ***Dziel****: Ciąg jest dzielony (jego elementy są przestawiane) na dwa niepuste podciągi takie, że każdy element pierwszego podciągu jest nie większy niż każdy element drugiego podciągu. Podział ciągu jest realizowany przez procedurę dzielącą*

(2) ***Zwyciężaj****: Dwa podciągi są sortowane za pomocą rekurencyjnych wywołań algorytmu sortowania szybkiego*

(3) ***Połącz****: Ponieważ podciągi są sortowane w miejscu, nie potrzeba nic robić, żeby je połączyć: cały ciąg jest już posortowany.*

Algorytm sortowania szybkiego - poziom projektowy

(1) *l ← 1; p ← N;*

(2) *Sort_szybki (T, l, p):*

  (2.1) *dopóki l < p, wykonuje co następuje:*

    (2.1.1) *i, j, T← Podzial(T, l, p);*

    (2.1.2) *Sort_szybki(T, l, j); // przejdź do kroku (2), gdzie p=j*

    (2.1.3) *Sort_szybki(T, i, p); // przejdź do kroku (2), gdzie l=i*

Algorytm Podział - poziom konceptualny

(1) *wskaż na środkowy element tablicy*

(2) *zapamiętaj wartość wskazanego elementu jako klucza*

(3) *wskaż na elementy lewy (pierwszy) i prawy (ostatni) ciągu*

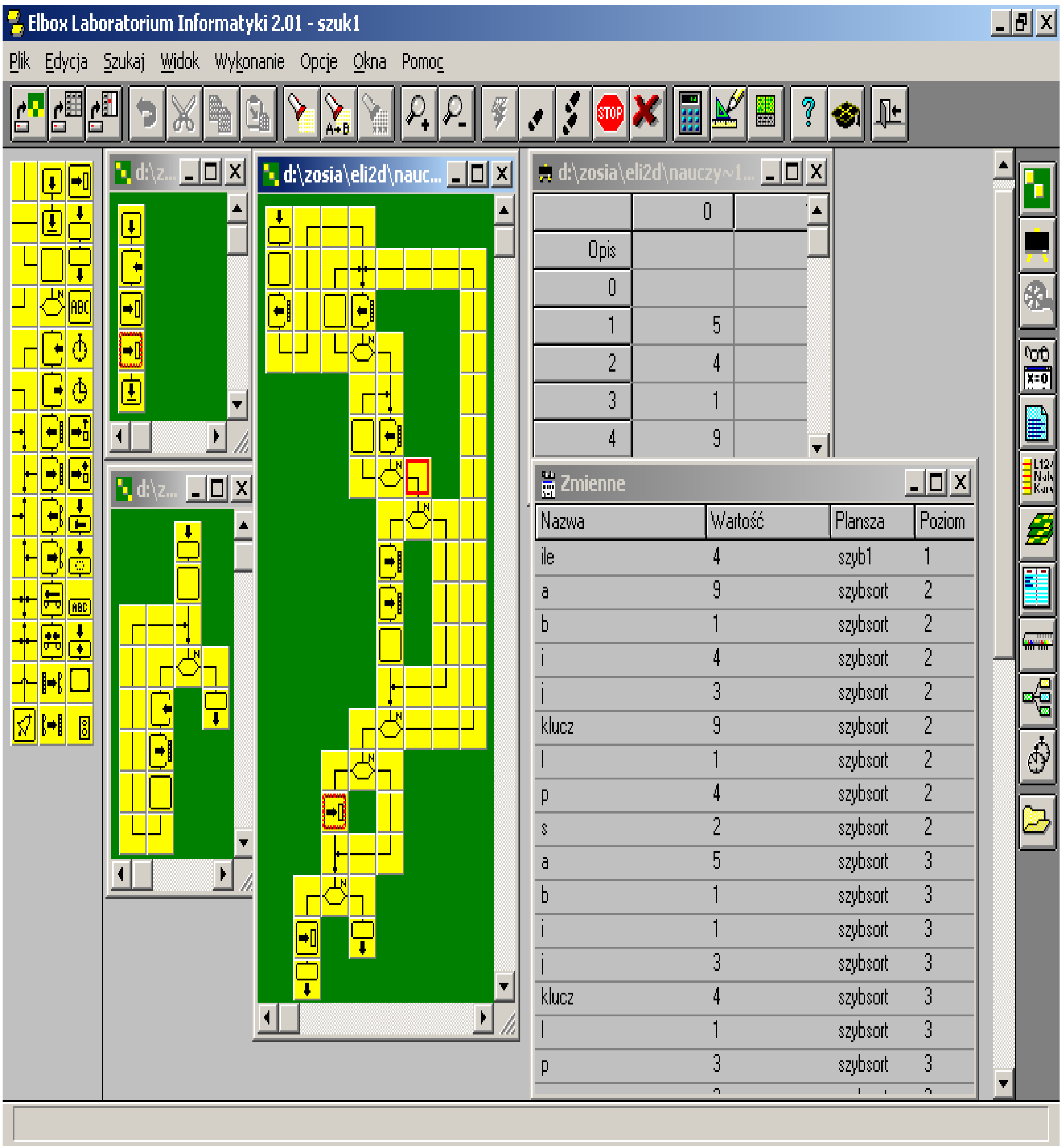
(4) *wykonuj, aż numer elementu lewego stanie się większy od numeru elementu prawego:*

- ***(3.1)*** *dopóki, zaczynając od wskazanego lewego elementu ciągu, nie znajdziesz elementu lewego większego lub równego kluczowi, testuj kolejne elementy*
- ***(3.2)*** *dopóki, zaczynając od wskazanego prawego elementu ciągu, nie znajdziesz elementu prawego mniejszego lub równego kluczowi, testuj kolejne elementy*
- ***(3.3)*** *jeśli numer lewego wskazanego elementu jest mniejszy lub równy numerowi prawego wskazanego elementu, to wykonaj:*
  - ***(3.3.1)*** *zamień te wskazane elementy*
  - ***(3.3.2)*** *wskaż następne elementy lewy i prawy*

(4) *zapamiętaj numery elementów lewego i prawego, na których zakończono przeszukiwania ciągu zaczynając od początku i od końca – dzielą one ciąg danych na dwie części.*

Algorytm Podział - poziom projektowy

```
Podzial(T, l, p)-> T, i, j;

(1) y ← ( l + p) div 2;
(2) klucz ← T(y);
(3) i ← l; j ← p;
(4) wykonuj, co następuje aż i > j:
    (4.1) dopóki T(i) < klucz, wykonuj, co następuje:
          (4.1.1) i ← i + 1;
    (4.2) dopóki T(j) > klucz, wykonuj co następuje:
          (4.2.1) j ← j - 1;
    (4.3) jeśli i <= j, to:
          (4.3.1) x ← T(i);
          (4.3.2) T(i) ← T(j)
          (4.3.3) T(j) ← x;
          (4.3.4) i ← i + 1;
          (4.3.5) j ← j -1.
```



Elbox Laboratorium Informatyki 2.01 - szuk1
Plik Edycja Szukaj Widok Wykonanie Opcje Okna Pomoc
d:\z...
d:\zosia\eli2d\nauc...
d:\zosia\eli2d\nauczy~1...
0
Opis
0
1 5
2 4
3 1
4 9
Zmienne
Nazwa Wartość Plansza Poziom
ile 4 szyb1 1
a 9 szybsort 2
b 1 szybsort 2
i 4 szybsort 2
j 3 szybsort 2
klucz 9 szybsort 2
l 1 szybsort 2
p 4 szybsort 2
s 2 szybsort 2
a 5 szybsort 3
b 1 szybsort 3
i 1 szybsort 3
j 3 szybsort 3
klucz 4 szybsort 3
l 1 szybsort 3
p 3 szybsort 3

Przykład sortowania szybkiego

Działanie		Dane działań						T							
		l	*p*	*i*	*j*	*y*	*klucz*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
								3	6	4	1	3	4	1	4
Sort	P(1,8)	1	8	**1**→2	6←**7**	4	1	**3**	6	4	1	3	4	**1**	4
		1	8	**2**→3	3←**4**	4	1	1	**6**	4	**1**	3	4	3	4
Podział: (1,2),(3,8)		1	8	3	2	4	1	1	1	4	6	3	4	3	4
Sort	P(1,2)	1	2	**1**→2	1←**2**	1	1	**1**	**1**						
Podział: (1,1),(2,2)		1	2	2	1	1	1	1	1						
		1	1												
Sort	K(1,2)	2	2					**1**	**1**						
Sort	P(3,8)	3	8	**3**→4	6←**7**	5	3			**4**	6	3	4	**3**	4
		3	8	**4**→5	4←**5**	5	3			3	**6**	**3**	4	4	4
Podział: (3,4),(5,8)		3	8	5	4	5	3			3	3	6	4	4	4
Sort	P(3,4)	3	4	**3**→4	3←**4**	3	3			**3**	**3**				
Podział: (3,3),(4,4)		3	4	4	3	3	3			3	3				
		3	3												
Sort	K(3,4)	4	4							**3**	**3**				
Sort	P(5,8)	5	8	**5**→6	7←**8**	6	4					**6**	4	4	**4**
		5	8	**6**→7	6←**7**	6	4					4	**4**	**4**	6
Podział: (5,6),(7,8)		5	8	7	6	6	4					4	4	4	6
Sort	P(5,6)	5	6	**5**→6	5←**6**	5	4					**4**	**4**		
Podział: (5,5),(6,6)		5	6	6	5	7	4					4	4		
		5	5												
Sort	K(5,6)	6	6									**4**	**4**		
Sort	P(7,8)	7	8	**7**→8	6←**7**	7	4							**4**	6
Podział: (7,6),(8,8)		7	8	8	6	7	4							4	6
		7	6												
Sort	K(7,8)	8	8											**4**	**6**
Sort	K(5,8)	5	8	7	6	6	4					**4**	**4**	**4**	**6**
Sort	K(3,8)	3	8	5	4	5	3			**3**	**3**	**4**	**4**	**4**	**6**
Sort	K(1,8)	1	8	3	2	4	1	**1**	**1**	**3**	**3**	**4**	**4**	**4**	**6**

P (l, p) - Początek wywołania nowego egzemplarza funkcji sortującej, czyli początek podziału podtablicy
Podział (l, j)(i, p) – Koniec podziału podtablicy na część (l, i) oraz (j, p)
K(l, p) - zakończenie wykonania danego egzemplarza funkcji sortującej

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <conio.h>

typedef int element;
const long N=20000L;
const int m=10;

inline void zamien(element &a, element &b);
void szybki(element t[], long l, long p);

void wypelnij(element t[], long& ile);
void wyswietl(element t[], long ile);

void main()
{ element * t=new element[N];
  long ile=0;
  wypelnij(t,ile);
        szybki(t,0,ile-1);
 wyswietl(t, ile);
 getch();
}
void wypelnij(element t[], long& ile)
  { srand(3);
    for(long i=0; i<N; i++)
           t[i]=rand();
    ile=N;}

void wyswietl(element t[], long ile)
 { for(long i=0; i<ile; i++)
          { printf("%d \n", t[i]);
         /*if (i%m==0)
          {char z=getch();
           if (z=='k') return; } */
       }
  }
```

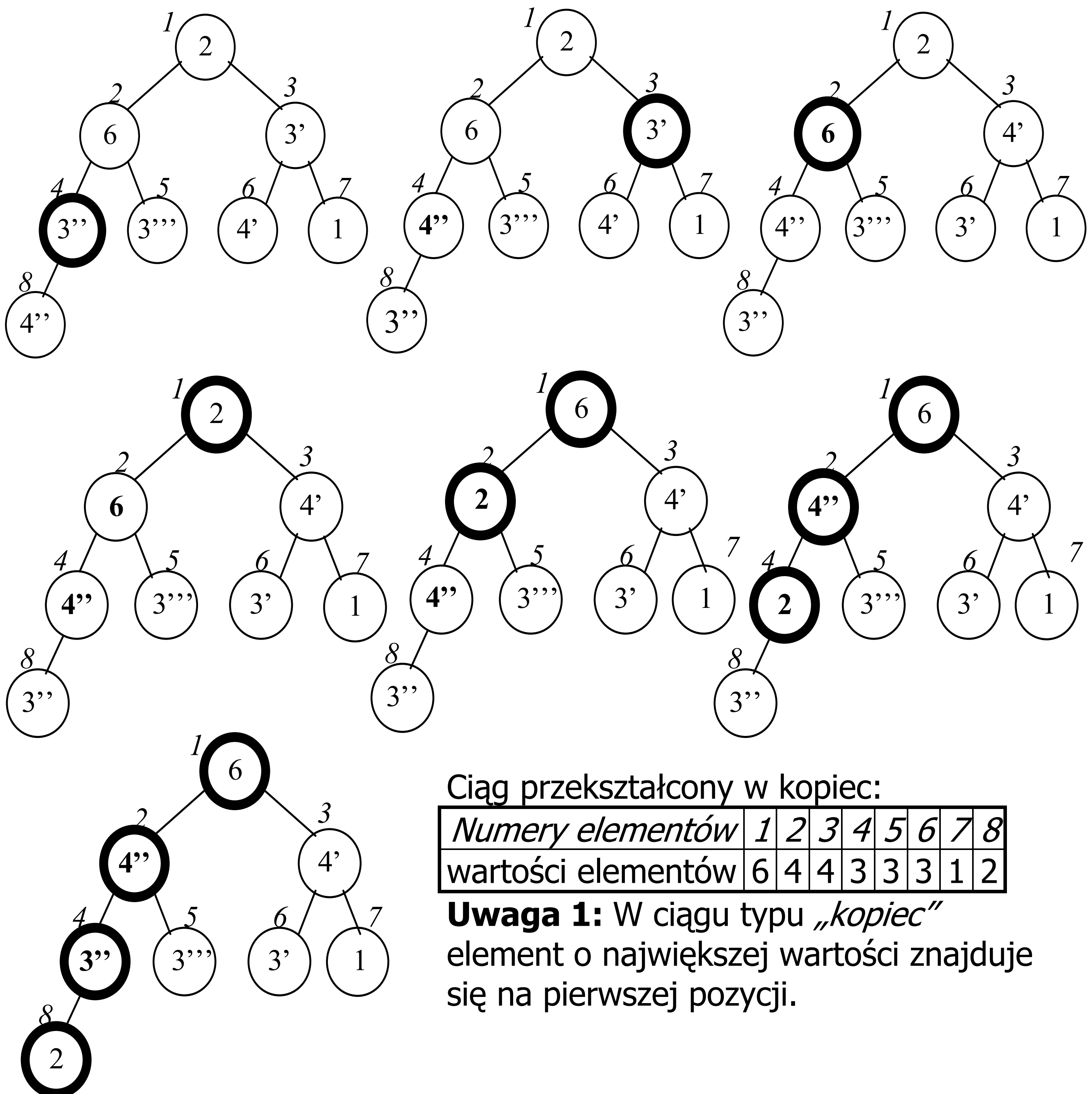
```
inline void zamien(element  &a, element  &b)
 { element pom=a;
   a=b;
   b=pom; }


void szybki(element t[], long l, long p)
  {    long i=l,j=p;
       element pom=t[(i+j)/2];
       do
        {  while(t[i]<pom) i++;
           while(t[j]>pom) j--;
            if (i<=j)
                { zamien(t[i],t[j]);
                   i++; j--;  }
         } while(i<=j);
       if (l<j)
            szybki(t,l,j);
       if (p>i)
            szybki(t,i,p);
   }
```

3. Algorytmy sortowania tablic- sortowanie przez kopcowanie (stogowe)

Przykład: Ciąg wejściowy:

Numery elementów	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
wartości elementów	2	6	3	3	3	4	1	4

Zadanie 1: Należy tak ustawić elementy w ciągu, aby wartość każdego elementu o numerze *„i"* (element zwany *„ojcem"*) była nie mniejsza niż wartość elementów o numerach: *„2 ∗ i"* (element *„lewy"*) oraz *„2 ∗ i + 1"* (element *„prawy"*).

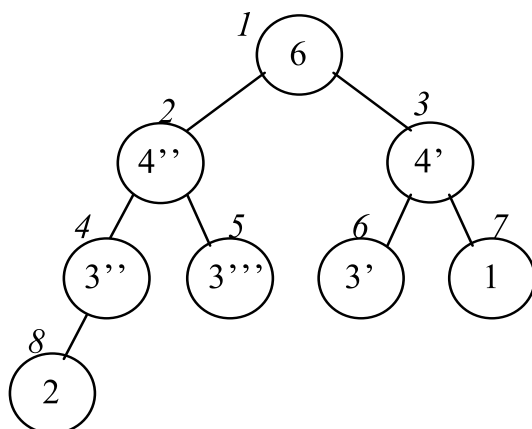


Ciąg przekształcony w kopiec:

Numery elementów	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
wartości elementów	6	4	4	3	3	3	1	2

Uwaga 1: W ciągu typu *„kopiec"* element o największej wartości znajduje się na pierwszej pozycji.

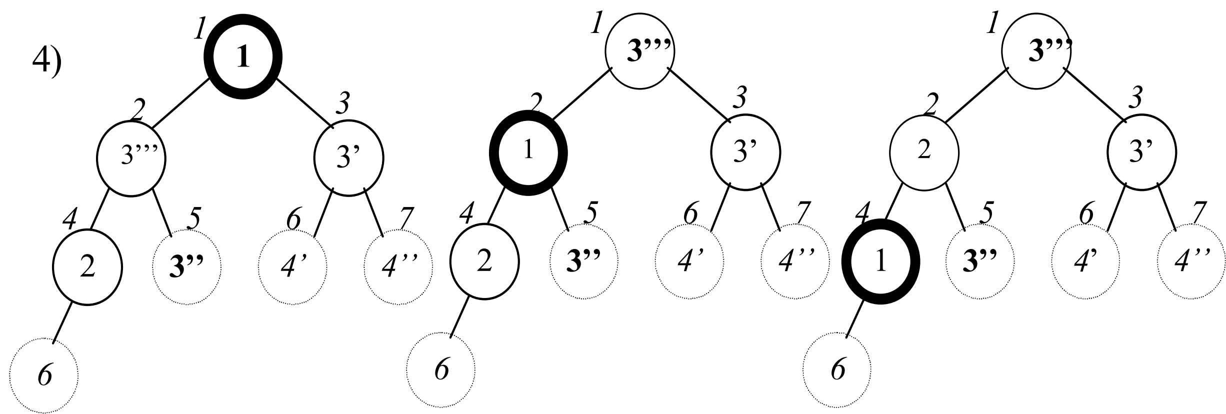
Zadanie 2: Należy ustawić elementy w ciągu w porządku rosnącym za pomocą kolejnego „przesiewania kopca" - czyli zamiany największego elementu z początku kopca z elementem ostatnim i odbudowy kopca za pomocą algorytmu przywracania kopca z pozostałych elementów (bez elementu ustawionego na końcu kopca). Proces „przesiewania kopca" należy powtarzać *„N - 1"* razy, gdzie *„N"* jest liczbą elementów ciągu.

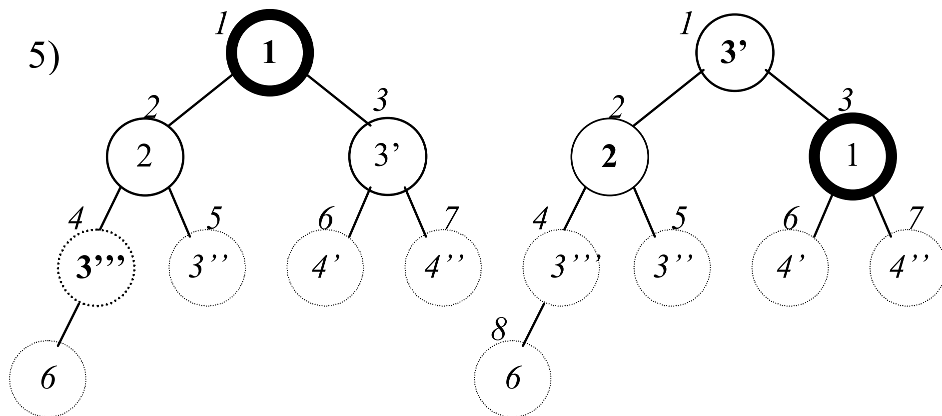


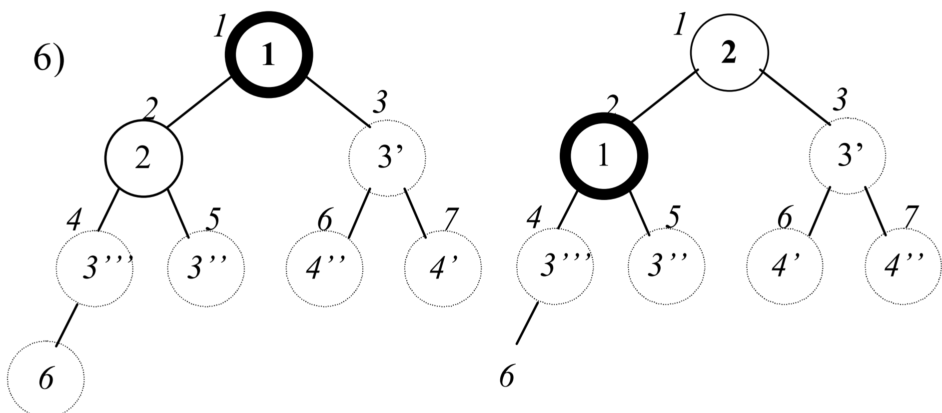
1)

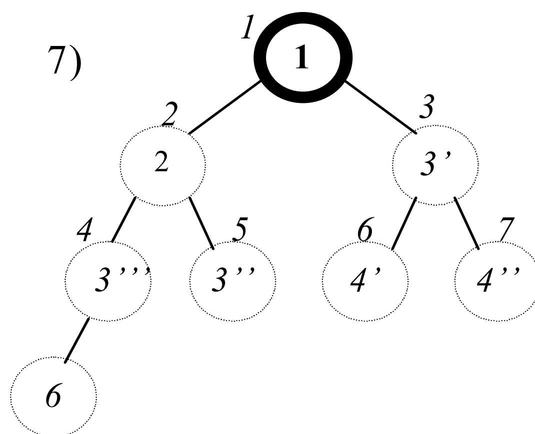
2)

3)









Uwaga 2:

Ciąg posortowany posiada również własności „kopca”, gdzie element o numerze *i* jest mniejszy lub równy elementom o numerach *2*i* oraz *2*i+1.*

*Algorytm **sortowania przez kopcowanie** - poziom konceptualny*

(1) *„Zbuduj kopiec" z ciągu złożonego z N elementów;*
(2) *wskaż na ostatnie miejsce „kopca";*
(3) *dopóki nie osiągniesz elementu o numerze 1, wykonuj dokładnie N -1 razy:*
- ***(3.1)*** *zamień element na wskazanym miejscu z elementem o numerze 1;*
- ***(3.2)*** *zmniejsz uzyskany ciąg o element umieszczony na wskazanym miejscu;*
- ***(3.3)*** *„Przywróć kopiec" z uzyskanego podciągu rozpoczynając od „ojca" o numerze 1;*
- ***(3.4)*** *wskaż na ostatnie miejsce odbudowanego „kopca".*

*Algorytm „Z**buduj kopiec"** - poziom konceptualny*

(1) *wskaż na środkowy element ciągu - jest element typu „ojciec";*
(2) *dopóki nie wyczerpiesz elementów z pociągu ograniczonego od góry wskazanym elementem, wykonuj:*
- ***(2.1)*** *„Przywróć kopiec" rozpoczynając od „ojca" o wskazanym numerze;*
- ***(2.2)*** *wskaż na element mniejszy o 1 od numeru „ojca" - jest to nowy wskazany „ojciec".*

*Algorytm **„Przywróć kopiec** - poziom konceptualny:*

(1) *Dopóki numer ojca jest mniejszy od rozmiaru kopca, wykonuj:*
- ***(1.1)*** *wskaż na element „lewy" odpowiadający danemu „ojcu";*
- ***(1.2)*** *wskaż na element „prawy" odpowiadający danemu „ojcu";*
- ***(1.3)*** *wybierz element o największej wartości z pośród elementów: „lewego", „prawego" i ich ojca i zapamiętaj jego numer*
- ***(1.5)*** *jeśli numer elementu największego jest różny od numeru „ojca" to wykonaj:*
  - ***(5.1)*** *zamień wartość „ojca" z wartością elementu największego;*
  - ***(5.2)*** *wskaż na numer nowego „ojca", równy numerowi dotąd elementu największego;*
  - ***(5.3)*** *„Przywróć kopiec" rozpoczynając od „ojca" o wskazanym numerze czyli zacznij od kroku (1).*
- ***(1.6)*** *przerwij algorytm, jeśli nie wykonano kroku 1.5*

*Algorytm **sortowania przez kopcowanie** - poziom projektowy*

Sort_kopiec(T, Rozmiar_tablicy)

(1) *Rozmiar_kopca ← Rozmiar_tablicy;*
(2) *Zbuduj_kopiec(T, Rozmiar_kopca);*
(3) *i ← Rozmiar_kopca*
(4) *dopóki i >= 2 wykonuj, co następuje:*
- ***(4.1)*** *x ← T(1);*
- ***(4.2)*** *T(1) ← T(i);*
- ***(4.3)*** *T(i) ← x;*
- ***(4.4)*** *Rozmiar_kopca ← Rozmiar_kopca - 1;*
- ***(4.5)*** *Ojciec ← 1;*
- ***(4.6)*** *Przywróć_kopiec(T, Rozmiar_kopca, Ojciec);*
- ***(4.7)*** *i ← i - 1;*

*Algorytm **„Zbuduj kopiec”** - poziom projektowy*

Zbuduj_kopiec(T, Rozmiar_kopca)

(1) *i ← Rozmiar_kopca **div** 2;*
(2) *dopóki i >= 1, wykonuj, co następuje:*
- ***(2.1)*** *Ojciec ← i;*
- ***(2.2)*** *Przywróć_kopiec(T, Rozmiar_kopca, Ojciec);*
- ***(2.3)*** *i ← i - 1.*

*Algorytm **„Przywróć kopiec”** - poziom projektowy:*

Przywróć_kopiec(T, Rozmiar_kopca, Ojciec):

(1) *dopóki Ojciec < Rozmiar_kopca, wykonuj, co następuje:*
(1.1) *l ← 2 ∗ Ojciec;*
(1.2) *p ← 2 ∗ Ojciec + 1;*
(1.3) *jeśli l <= Rozmiar_kopca i T(l) > T(Ojciec), to wykonaj:*
- ***(1.3.1)*** *Numer_największego_elementu ← l*
- ***(1.3.2)*** *w przeciwnym przypadku: Numer_największego elementu ← Ojciec;*

(1.4) *jeśli p <= Rozmiar_kopca i T(p) > T(Numer_największego_elementu), to wykonaj:*
- ***(1.4.1)*** *Numer_największego_elementu ← p;*
- ***(1.4.2)*** *w przeciwnym przypadku: Numer_największego elementu bez zmian;*

(1.5) *jeśli Numer_największego elementu <> Ojciec, to wykonaj:*
- ***(1.5.1)*** *x ← T(Numer_największego_elementu);*
- ***(1.5.2)*** *T(Numer_największego_elementu) ← T(Ojciec);*
- ***(1.5.3)*** *T(Ojciec) ← x;*
- ***(1.5.4)*** *Ojciec ← Numer_największego_elementu;*
- ***(1.5.5)*** *Przejdź do kroku (1)*

(1.6) *lub przerwij algorytm, jeśli Numer_największego elementu == Ojciec, czyli nie wykonano punktu 1.5.*

Przykład sortowania przez kopcowanie

1. Pierwsza faza algorytmu - budowa kopca.

Działanie	Dane działań [indeksy]						T							
	lewy	*prawy*	*i*	*Ojciec*	*maks*	*Rozmiar_kopca*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Sort_kopiec						8	2	6	3	3	3	4	1	4
Zbuduj_kopiec			4			8				**3**				
Przywroc_kopiec	8	9		4	8	8				**3**				4
Przywroc_kopiec	8	9		4	8	8				4				**3**
Przywroc_kopiec				8		8								
Zbuduj_kopiec			3			8			**3**					
Przywroc_kopiec	6	7		3	6	8			**3**			4	1	
Przywroc_kopiec	6	7		3	6	8			4			**3**	1	
Przywroc_kopiec	12	13		6		8								
Zbuduj_kopiec			2			8		**6**						
Przywroc_kopiec	4	5	2	2	2	8		**6**		4	3			
Zbuduj_kopiec			1			8	**2**							
Przywroc_kopiec	2	3	1	1	2	8	**2**	6	4					
Przywroc_kopiec	2	3	1	1	2	8	6	**2**	4					
Przywroc_kopiec	4	5	1	2	4	8		**2**		4	3			
Przywroc_kopiec	4	5	1	2	4	8		4		**2**	3			
Przywroc_kopiec	8	9	1	4	8	8				**2**				3
Przywroc_kopiec	8	9	1	4	8	8				3				**2**
Sort_kopiec						8	6	4	4	3	3	3	1	2

2. Druga faza algorytmu - sortowanie przez przesiewanie kopca.

Działanie	Dane działań [indeksy]						T							
	lewy	*prawy*	*i*	*Ojciec*	*maks*	*Rozmiar_kopca*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
							6	4	4	3	3	3	1	2
Sort_kopiec			8	1		7	2	4	4	3	3	3	1	6
Przywroc_kopiec	2	3		1	2	7	2	4	4					6
Przywroc_kopiec	2	3		1	2	7	4	2	4					
Przywroc_kopiec	4	5		2	4	7		2		3	3			
Przywroc_kopiec	4	5		2	4	7		3		2	3			
Przywroc_kopiec	8	9		4		7	4	3	4	2	3	3	1	6
Sort_kopiec			7	1		6	1	3	4	2	3	3	4	6
Przywroc_kopiec	2	3		1	3	6	1	3	4				4	6
Przywroc_kopiec	2	3		1	3	6	4	3	1				4	6
Przywroc_kopiec	6	7		3	6	6			1			3	4	6
Przywroc_kopiec	6	7		3	6	6			3			1	4	6
Przywroc_kopiec				6		6	4	3	3	2	3	1	4	6
Sort_kopiec			6	1		5	1	3	3	2	3	4	4	6
Przywroc_kopiec	2	3		1	2	5	1	3	3			4	4	6
Przywroc_kopiec	2	3		1	2	5	3	1	3			4	4	6
Przywroc_kopiec	4	5		2	5	5		1		2	3	4	4	6
Przywroc_kopiec	4	5		2	5	5		3		2	1	4	4	6
Przywroc_kopiec				5		5	3	3	3	2	1	4	4	6
Sort_kopiec			5	1		4	1	3	3	2	4	4	4	6
Przywroc_kopiec	2	3		1	2	4	1	3	3		4	4	4	6
Przywroc_kopiec	2	3		1	2	4	3	1	3		4	4	4	6
Przywroc_kopiec	4	5		2	4	4		1		2	4	4	4	6
Przywroc_kopiec	4	5		2	4	4		2		1	4	4	4	6
Przywroc_kopiec				4			3	2	3	1	4	4	4	6
Sort_kopiec			4	1		3	1	2	3	3	4	4	4	6
Przywroc_kopiec	2	3		1	3	3	1	2	3	3	4	4	4	6
Przywroc_kopiec	2	3		1	3	3	3	2	1	3	4	4	4	6
Przywroc_kopiec				3			3	2	1	3	4	4	4	6
Sort_kopiec			3	1		2	1	2	3	3	4	4	4	6
Przywroc_kopiec	2	3		1	2	2	1	2	3	3	4	4	4	6
Przywroc_kopiec	2	3		1	2	2	2	1	3	3	4	4	4	6
Przywroc_kopiec				2		2	2	1	3	3	4	4	4	6
Sort_kopiec			2	1		1	1	2	3	3	4	4	4	6
Przywroc_kopiec				1		1	1	2	3	3	4	4	4	6
Sort_kopiec							1	2	3	3	4	4	4	6

Program w C++

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <conio.h>

typedef int element;
const long N=20000L;
const int m=10;

inline void zamien(element &a, element &b);
void zbuduj_kopiec(element t[], long l, long p);
void przywroc_kopiec(element t[], long p, long ojciec);
void sort_kopiec(element t[], long l, long p);

void wypelnij(element t[], long& ile);
void wyswietl(element t[], long ile);

void main()
{ element * t=new element[N];
  long ile=0;
  wypelnij(t,ile);
  wyswietl(t,ile); //ile musi być większe od 1
  sort_kopiec(t,0,ile-1);
  wyswietl(t,ile);
  getch();
}

void wypelnij(element t[], long& ile)
 { srand(3);
   for(long i=0; i<N; i++)
    t[i]=rand();
  ile=N; }

void wyswietl(element t[], long ile)
 { for(long i=0; i<ile; i++)
    { printf("%d \n", t[i]);
      if (i%m==0)
        {char z=getch();
         if (z=='k') return;}
    }
 }
```

```
void sort_kopiec(element t[], long l, long p)
 { zbuduj_kopiec(t, l, p);
   for (long i = p; i>l; i--)
    { zamien(t[l],t[i]);      //na pozycji p umieszczono kolejny największy element
      p -=1;
      przywroc_kopiec(t, p, l);
   }
 }

void zbuduj_kopiec(element t[], long l, long p)
 {  if (p<=l) return; //tablica jednoelementowa lub brak tablicy
    for  (long i = (l+p-1)/2; i>=l; i--)
    przywroc_kopiec(t, p, i);}

void przywroc_kopiec(element t[], long p, long ojciec)
 {  long ll, pp, maks;
   while (ojciec < p)
     {  ll =  ojciec*2+1;
         pp = ll+1;
        if (ll <= p && t[ll] > t[ojciec])   //wybór największego z trzech elementów
            maks = ll;
         else maks = ojciec;
         if (pp <= p && t[pp] > t[maks])
            maks = pp;
         if (maks != ojciec) //naruszono zasadę kopca w elemencie 2*ojciec lub 2*ojciec+1
            {  zamien(t[maks], t[ojciec]);
              ojciec=maks; }         //należy ją więc przywrócić
        else break;                  //przerwanie przywracania kopca, gdy ostatnio sprawdzany
     }                               //węzeł spełniał zasadę kopca
 }


inline void zamien(element  &a, element  &b)
   {
     element pom=a;
     a=b;
     b=pom;
  }
```

4. Analiza algorytmów sortowania – podsumowanie
[T.Cormen, C.Leiserson, R.Rivest:"Wprowadzenie do algorytmów"]

Podział algorytmów sortowania ze względu na sposób ustalania kolejności sortowanego ciągu:

1. przez *porównywanie elementów* - sortowanie bąbelkowe, przez kopcowanie, przez selekcję, sortowanie przez wstawianie;
2. przez *wyznaczanie pozycji* danej w ciągu posortowanym na podstawie liczby elementów czyli sortowanie przez zliczanie, przez wykorzystanie sposobu reprezentowania wartości elementów czyli sortowanie pozycyjne.

1. Rząd wielkości funkcji

Czas działania algorytmu dla konkretnych danych wejściowych jest wyrażony liczbą wykonanych prostych (elementarnych) operacji lub „kroków". Zakłada się, że operacja elementarna jest maszynowo niezależna. Czas wykonania *i-tego* wiersza programu wymaga czasu c_j.

Dla dostatecznie dużych danych wejściowych liczymy jedynie rząd wielkości czasu działania algorytmu - oznacza to **asymptotyczną złożoność algorytmu,** która określa, jak szybko wzrasta czas działania algorytmu, gdy rozmiar danych dąży do nieskończoności.

Notacje asymptotyczne

Opis asymptotyczny czasu działania algorytmów korzysta z funkcji, których zbiorem argumentów jest zbiór liczb naturalnych $N=\{0,1,2,..\}$. Jest to zgodne z typem rozmiaru danych, będącego liczbą naturalną.

Notacja Θ

Dla danej funkcji g(n) oznaczamy przez Θ(g(n)) zbiór funkcji:

$\Theta(g(n)) = \{f(n):$ istnieją dodatnie stałe c_1, c_2 i n_0 takie, że

$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ dla wszystkich $n \ge n_0\}$

Notacja ta określa ***asymptotycznie dokładne oszacowanie*** funkcji $f(n)$. Funkcja $f(n)$ należy do zbioru $\Theta(g(n))$, jeśli istnieją dodatnie stałe c_1 oraz c_2 takie, że funkcja może być „wstawiona między" $c_1\, g(n)$ i $c_2\, g(n)$ dla dostatecznie dużych n.

Piszemy

$$f(n) = \Theta(g(n)),$$

co oznacza, że funkcja $f(n)$ jest elementem zbioru $\Theta(g(n))$, lub

$$f(n) \in \Theta(g(n)),$$

Notacja Θ asymptotycznie ogranicza funkcję od góry i od dołu.

Notacja O

Dla danej funkcji $g(n)$ oznaczamy przez $O(g(n))$ zbiór funkcji

$$O(g(n)) = \{ f(n): \text{istnieją dodatnie stałe } c \text{ i } n_0 \text{ takie, że } 0 \leq f(n) \leq cg(n) \text{ dla wszystkich } n \geq n_0\}$$

Notacja asymptotyczna O określa ***asymptotyczną granicę górną*** funkcji $f(n)$, gdyż dla każdego n większego od n_0 wartość funkcji $f(n)$ nie przekracza $g(n)$.

Mamy:

$$\Theta(g(n)) \subseteq O(g(n))$$

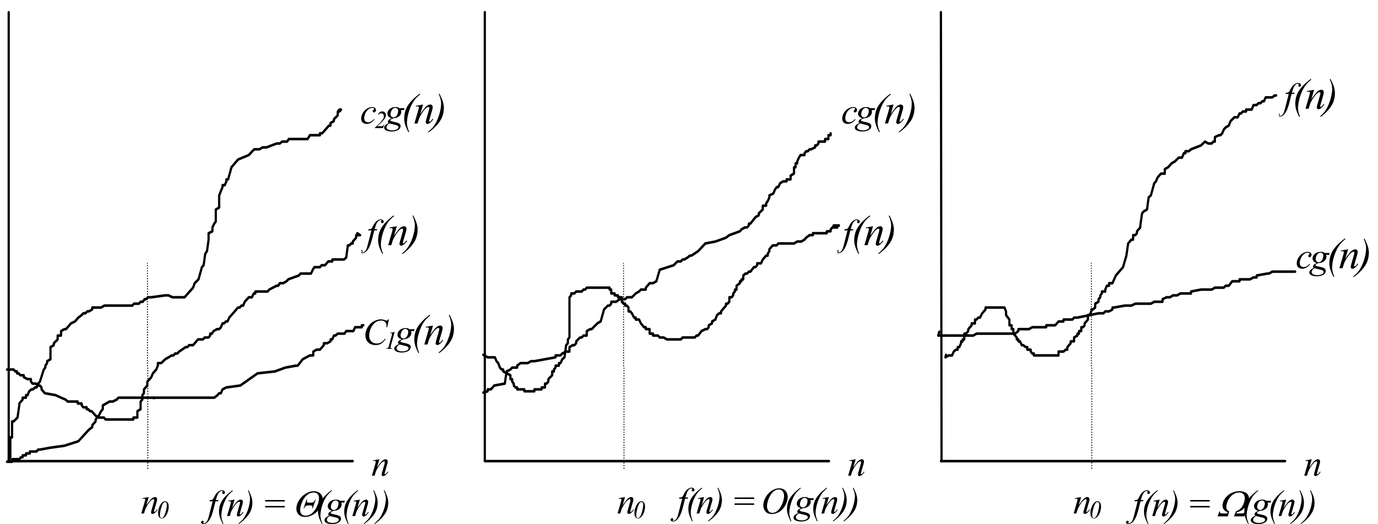
Np. każda funkcja $a\, n^2 + bn + c$, gdzie $a > 0$ należy do zbioru $\Theta(n^2)$, stąd każda funkcja kwadratowa należy także do zbioru $O(n^2)$.

Notacja Ω

Dla danej funkcji g(n) oznaczamy przez Ω(g(n)) zbiór funkcji

$$\Omega(g(n)) = \{f(n) : \text{istnieją dodatnie stałe } c \text{ i } n_0 \text{ takie, że } 0 \leq cg(n) \leq f(n) \text{ dla wszystkich } n \geq n_0\}$$

Notacja ta określa ***asymptotyczną granicę dolną*** funkcji $f(n)$, gdyż dla wszystkich wartości n większych od n_0 wartość funkcji $f(n)$ jest nie mniejsza niż $cg(n)$.



Graficzne przykłady notacji Θ, O, Ω

Aksjomatyczne własności porównywania funkcji:

1) Przechodniość:

$f(n) = \Theta(g(n))$ i $g(n) = \Theta(h(n))$ implikuje $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ i $g(n) = O(h(n))$ implikuje $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ i $g(n) = \Omega(h(n))$ implikuje $f(n) = \Omega(h(n))$

2) Zwrotność

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

$f(n) = \Omega(f(n))$

3) Symetria

$f(n) = \Theta(g(n))$ wtedy i tylko wtedy, gdy $g(n) = \Theta(f(n))$

4) Symetria transpozycyjna

$f(n) = O(g(n))$ wtedy i tylko wtedy, gdy $g(n) = \Omega(f(n))$

Analogia między porównywaniem dwóch funkcji f i g oraz porównywaniem dwóch liczb rzeczywistych:

$f(n) = O(g(n)) \approx a \le b$

$f(n) = \Omega(g(n)) \approx a \ge b$

$f(n) = \Theta(g(n)) \approx a = b$

Trichotomia dla liczb

Dla każdych dwóch liczb rzeczywistych a i b zachodzi dokładnie jeden z następujących faktów: $a < b$, $a = b$ oraz $a > b$

Własność **trichotomii nie przenosi się na funkcje w sensie asymptotycznym:**

np. $f(n) = n^{1 + sin(n)}$ (wartość wykładnika waha się miedzy 0 i 2) i

$g(n)=n$ nie wystąpią w zależnościach:

$f(n) = O(g(n))$ oraz $f(n) = \Omega(g(n))$.

Przykłady:

1) $\Theta(1)$ oznacza albo wartość stałą albo funkcję stałą względem pewnej zmiennej.

2) $2n^2+\Theta(n)= \Theta(n^2)$ oznacza, że niezależnie od tego, jak anonimowe funkcje są po lewej stronie znaku równości, można wybrać anonimowe funkcje po prawej stronie równości tak, że równanie będzie zachodzić.

Np. $2n^2+n= \Theta(n^2)$ *czyli* $2n^2+n<4n^2$ *oraz* $2n^2+n>2n^2$

3) $2n^2+3n+1= 2n^2 + \Theta(n)$

Oznacza to, że dla $\mathbf{f(n)} \in \mathbf{\Theta(n)}$ *jest prawdziwe dla wszystkich następujące równanie* $2n^2+3n+1=2n^2+f(n)$

oraz dla $\mathbf{h(n)} \in \mathbf{\Theta(n^2)}$ *jest prawdziwe równanie* $2n^2+f(n)=h(n)$

4) $2n^2-3n+1= O(n^2)$, gdyż $2n^2-3n+1=f(n)$ i $\mathbf{f(n)} \in \mathbf{O(n^2)}$ *np.* $2n^2-3n+1< 2n^2$

2. Uzyskiwanie asymptotycznych oszacowań rozwiązań przy zastosowaniu notacji Θ i O - rekurencje

Przykład rekurencji dla algorytmu ***sortowania przez łączenie***

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & \text{jeśli } n > 1 \end{cases}$$

gdzie: *dla liczb rzeczywistych x mamy*

$\lceil x \rceil < x+1$ - najmniejsza liczba całkowita nie mniejsza niż x *np.* $\lceil 3.14 \rceil = 4$

$x-1 < \lfloor x \rfloor$ - największa liczba całkowita nie większa niż x *np.* $\lfloor 3.14 \rfloor = 3$

Dla każdej liczby całkowitej n mamy $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

Dla każdej liczby całkowitej n oraz liczb całkowitych $a \neq 0$ i $b \neq 0$ mamy:

$\lceil \lceil n/a \rceil /b \rceil = \lceil n/ab \rceil$

$\lfloor \lfloor n/a \rfloor /b \rfloor = \lfloor n/ab \rfloor$

Metody rozwiązywania rekurencji:

1) Metoda iteracyjna

2) Metoda podstawiania

3) Metoda rekurencji uniwersalnej

Założenia:

- argumentami funkcji są liczby całkowite
- czas działania algorytmu dla danych wejściowych o stałym rozmiarze jest stały, stąd przyjmuje się, że czas działania algorytmu da małych rozmiarów danych wejściowych N jest stały.

Metoda iteracyjna

Metoda iteracyjna polega na iterowaniu rekurencji i wyrażania jej jako sumy składników zależnych od n oraz od warunków brzegowych.

Na podstawie tego założenia iteracje rekurencji dla algorytmu sortowania bąbelkowego są następujące:

$$T(n) = T(n-1) + O(n)$$

$$T(n-1)= T(n-2) +O(n-1)$$

..

$$T(2)= T(1) + O(2)$$

$$T(1)= O(1)$$

$$T(n)+T(n-1)+...+T(2)+T(1)=T(n-1)+T(n-2)+..+T(1)+ O(n) +O(n-1) +...+O(2)+ O(1)$$

czyli mamy:

$T(n)+T(n-1)+...+T(2)+T(1)=$

$T(n-1)+T(n-2)+..+T(1)+ O(n) +O (n-1)+ ...+ O(2) + O(1)$

Po dodaniu stronami i uproszczeniu otrzymano:

$$T(n) = T(1) + \sum_{n=1}^{N} O(n) = T(1) + O(N^2)\ dla\ n \in O(n)$$

Mamy więc dla sortowania bąbelkowego: $T(n)= O(N^2)$

Metoda podstawiania

Metoda rozwiązywania równań rekurencyjnych przez podstawianie polega na przyjęciu postaci rozwiązania, a następnie wykazaniu przez indukcję, że jest ono poprawne. Metoda może być użyta do określenia albo górnego albo dolnego oszacowania wartości rozwiązania rekurencji.

Udowodnij, że $T(n)=O(n^2)$ dla $T(n)<=T(n-1)+n$, gdzie T(n) jest rekurencją dla **sortowania bąbelkowego**

Otrzymano: $T(n)<=(n-1)^2+n=n^2-2n+1+n=n^2-n+1<=n^2$

czyli $\boldsymbol{T(n)=O(n^2)}$

Przykład 1

Przykład rekurencji dla algorytmu ***sortowania przez łączenie***

$$T(n) = \begin{cases} \Theta(1), & \text{jeśli } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & \text{jeśli } n > 1 \end{cases}$$

a) Założenia:

- argumentami funkcji są liczby całkowite
- czas działania algorytmu dla danych wejściowych o stałym rozmiarze jest stały, stąd przyjmuje się, że czas działania algorytmu da małych rozmiarów danych wejściowych N jest stały. Na podstawie tego założenia rekurencja dla algorytmu sortowania przez łączenie bez podawania wartości dla małych N jest następująca:

$$T(n) = 2T(n/2) + \Theta(n)$$

Metoda rozwiązywania równań rekurencyjnych przez podstawianie polega na przyjęciu postaci rozwiązania, a następnie wykazaniu przez indukcję, że jest ono poprawne. Metoda może być użyta do określenia albo górnego albo dolnego oszacowania wartości rozwiązania rekurencji.

b) Rozwiązanie rekurencji dla algorytmu sortowania przez łączenie

- $T(n) = 2T(n/2) + \Theta(n) \rightarrow T(n) = 2T(\lfloor n/2 \rfloor) + n$
- odgadnięte rozwiązanie: $T(n) = O(n \lg n)$, należy udowodnić, że $T(n) \leq c\ n \lg n$ *(gdzie* $\lg n \equiv \lg_2 n$*)*
- rozwiązanie: $T(\lfloor n/2 \rfloor) \leq (c\lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor))$,

  czyli

$$T(n) \leq 2\ (c\lfloor n/2 \rfloor \lg (\lfloor n/2 \rfloor)) + n \leq cn \lg(n/2) + n = cn \lg n - cn\lg 2 + n = c\ n \lg n - cn + n$$

$$T(n) \leq cn \lg n \text{ dla } c \geq 2 \text{ i } n \geq 2.$$

c) Wynik: czas działania procedury sortowania przez łączenie jest równy *O(n lgn).*

Przykład 2 - rozwiązanie rekurencji dla algorytmu ***sortowania szybkiego***

a) przypadek pesymistyczny:

- $T(n) = \max\limits_{1\le q\le n-1}(T(q)+T(n-q))+\Theta(n)$, gdzie q oznacza podział obszaru danych, z których jeden ma co najmniej 1 element.
- odgadnięte rozwiązanie $O(n^2)$, stąd należy udowodnić, że $T(n) \le cn^2$
- rozwiązanie $$T(n) \le \max_{1\le q\le n-1}(cq^2 + c(n-q)^2) + \Theta(n)$$

$$T(n)\le c\max_{1\le q\le n-1}(q^2+(n-q)^2)+\Theta(n)\le c(1^2+c(n-1)^2 = cn^2 - 2c(n-1)\le cn^2,$$

W przypadku pesymistycznym czas działania procedury sortowania szybkiego jest równy *O(n²)*.

b) przypadek średni

- $T(n) = \frac{1}{n}(T(1)+T(n-1)+\sum_{q=1}^{n-1}(T(q)+T(n-q)))+\Theta(n)$
- odgadnięte rozwiązanie *O(n lgn),* stąd należy udowodnić, że $T(n) \le an\ lgn + b$
- rozwiązanie: $\frac{1}{n}(T(1)+T(n-1)=\frac{1}{n}(\Theta(1)+\Theta(n^{21}))=O(n)$, czyli

$$T(n)=\frac{1}{n}\sum_{q=1}^{n-1}(T(q)+T(n-q))+\Theta(n)=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+\Theta(n)\le\frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k+b)+\Theta(n)$$
$$=\frac{2a}{n}\sum_{k=1}^{n-1}k\lg k+\frac{2b}{n}(n-1)+\Theta(n),$$

gdzie zakładamy $\sum_{k=1}^{n-1}k\lg k\le\frac{1}{2}n^2\lg n-\frac{1}{8}n^2$, gdyż $\sum_{k=1}^{n-1}k\lg k\le\frac{1}{2}n^2\lg n-\Omega(n^2)$, czyli

$$\sum_{k=1}^{n-1}k\lg k=\sum_{k=1}^{\lceil n/2\rceil-1}k\lg k+\sum_{k=\lceil n/2\rceil}^{n-1}k\lg k\le(\lg n-1)\sum_{k=1}^{\lceil n/2\rceil-1}k+\lg n\sum_{k=\lceil n/2\rceil}^{n-1}k=\lg n\sum_{k=1}^{n-1}k-\sum_{k=1}^{\lceil n/2\rceil-1}k$$
$$\le\frac{1}{2}n(n-1)\lg n-\frac{1}{2}(\frac{n}{2}-1)\frac{n}{2}\le\frac{1}{2}n^2\lg n-\frac{1}{8}n^2$$,stąd

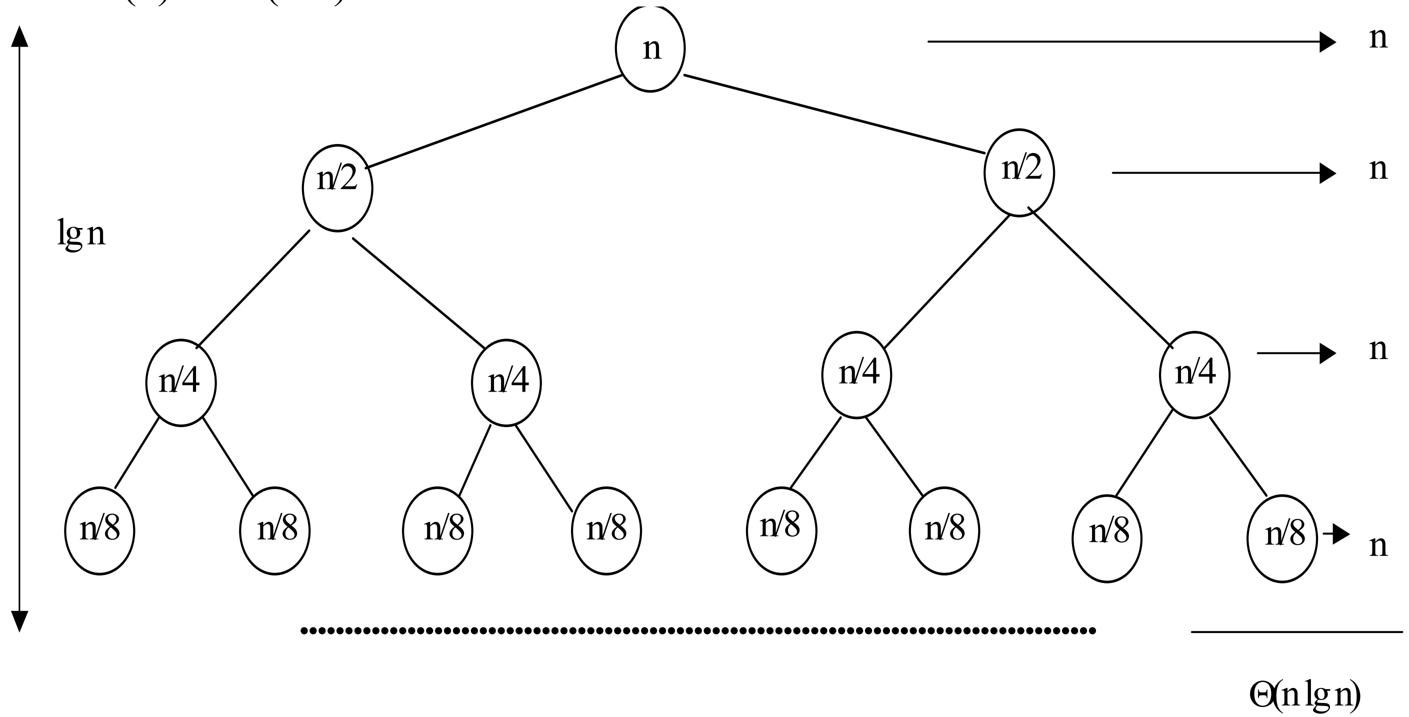
$$T(n)\le\frac{2a}{n}(\frac{1}{2}n^2\lg n-\frac{1}{8}n^2)+\frac{2b}{n}(n-1)+\Theta(n)\le\ a\ n\ lgn - \frac{a}{4}n + 2b + \Theta(n)$$
$$= a\ n\ lgn + b + (b - \frac{a}{4}n + \Theta(n)),$$ czyli $T(n) \le a\ n\ lgn + b$

W przypadku średnim czas działania procedury sortowania szybkiego jest równy *O(n lgn).*

Drzewa algorytmów

1) Przypadek najlepszy

$T(n) = 2T(n/2) + n$

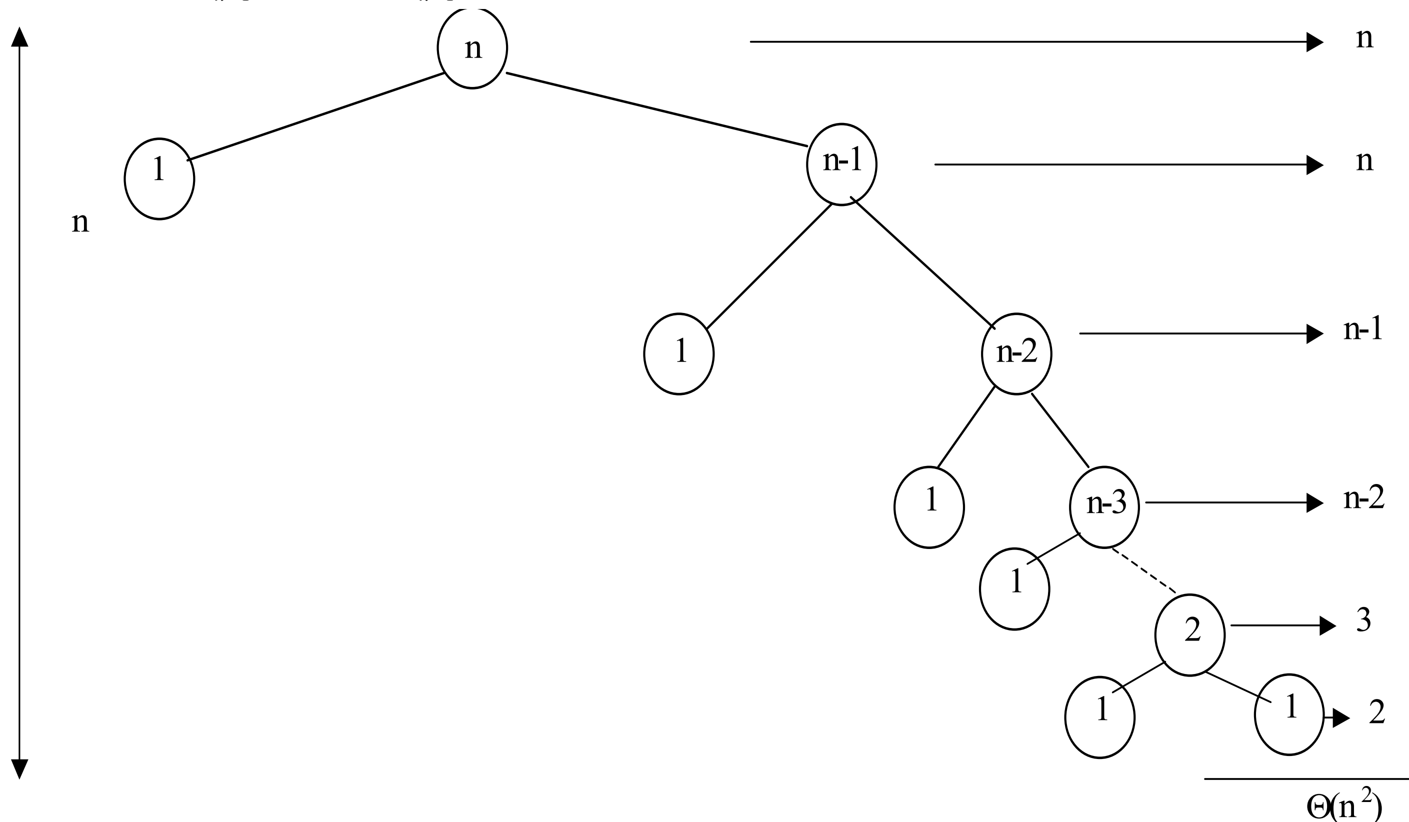


2) Przypadek najgorszy

$T(n) = T(1)+ T(n-1) + \Theta(n)$, gdzie $T(1)= \Theta(1)$

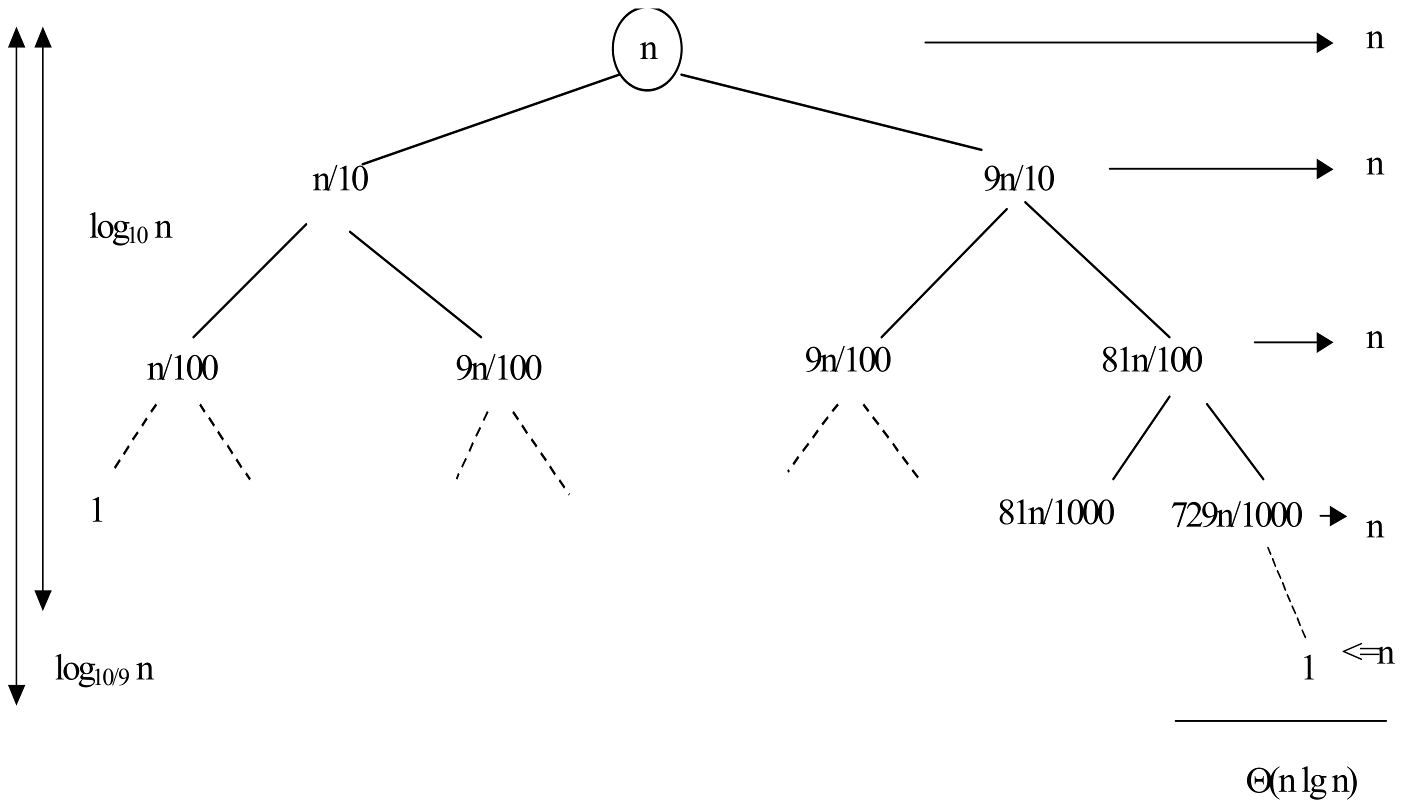
$T(n) = T(n-1) +\Theta(n)$, czyli po zastosowaniu metody iteracyjnej

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$$



3) Przypadek zrównoważony

T(n) = T(n/10) + T(9n/10) + n



4) Przypadek średni

Stanowi połączenie przypadku najgorszego, najlepszego i zrównoważonego i jest równy O(n lg n).

Metoda rekurencji uniwersalnej

Twierdzenie o rekurencji uniwersalnej

Niech $a \geq 1$ i $b > 1$ będą stałymi, niech $f(n)$ będzie pewną funkcją i niech $T(n)$ będzie zdefiniowane dla nieujemnych liczb całkowitych przez rekurencję:

$$T(n) = aT(n/b) + f(n),$$

gdzie n/b oznacza $\lfloor n/b \rfloor$ lub $\lceil n/b \rceil$. Wtedy funkcja $T(n)$ może być ograniczona asymptotycznie w następujący sposób:

1. Jeśli $f(n) = O(n^{\log_b a-\varepsilon})$ dla pewnej stałej ε >0, to $T(n) = \Theta(n^{\log_b a})$.
2. Jeśli $f(n) = \Theta(n^{\log_b a})$, to $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Jeśli $f(n) = \Omega(n^{\log_b a+\varepsilon})$ dla pewnej stałej $\varepsilon > 0$ i jeśli $af(n/b) \leq cf(n)$ dla pewnej stałej $c < 1$ i wszystkich dostatecznie dużych *n*, to $T(n) = \Theta(f(n))$.

Metoda rekurencji uniwersalnej podaje „uniwersalny przepis" rozwiązywania równania rekurencyjnego postaci:

$$T(n) = aT(n/b) + f(n),$$

gdzie $a \geq 1$ i $b>1$ są stałymi, a $f(n)$ jest funkcją asymptotycznie dodatnią. Rekurencja opisuje czas działania algorytmu, który dzieli problem rozmiaru *n* na *a* podproblemów, każdy rozmiaru *b/n,* gdzie *a* i *b* są dodatnimi stałymi. Każdy z *a* podproblemów jest rozwiązywany rekurencyjnie w czasie $T(n/b)$. Koszt dzielenia problemu oraz łączenia rezultatów częściowych jest opisany funkcją $f(n)$. Metoda rekurencji uniwersalnej wymaga rozważenia trzech przypadków. W każdym z trzech przypadków porównujemy funkcję $f(n)$ z funkcją $n^{\log_b a}$. Intuicyjnie, rozwiązanie rekurencji zależy od większej z tych dwóch funkcji. Jeśli funkcja $n^{\log_b a}$ jest większa, to rozwiązaniem jest przypadek (1), jeśli jest mniejsza to rozwiązaniem jest przypadek (3). Jeśli obie funkcje są tego samego rzędu, to prawą stronę mnożymy przez czynnik logarytmiczny i rozwiązaniem jest przypadek (2).

Ograniczenia:

- funkcja $f(n)$ musi być wielomianowo mniejsza niż $n^{\log_b a}$ w przypadku (1), czyli mniejsza o n^{ε}
- funkcja f(n) musi być wielomianowo większa niż $n^{\log_b a}$ w przypadku (3) czyli większa o n^{ε} oraz $af(n) \leq bf(n)$

Przykład 1.

***Zastosowania metody rekurencji uniwersalnej* do określenia rzędu złożoności obliczeniowej sortowania przez kopcowanie (stogowego)**

a) czas działania procedury *Przywroc_kopiec* można wyznaczyć za pomocą metody rekurencji uniwersalnej

$$T(n) = aT(n/b) + f(n)$$

Najgorszy przypadek w sortowaniu występuje, gdy ostatni rząd w drzewie jest wypełniony dokładnie do połowy oraz podrzewo lewe ma wtedy nie więcej niż 2n/3 węzłów. Stąd czas działania procedury *Przywroc_kopiec* jest opisany rekurencją:

$$T2(n) \le T(2n/3) + \Theta(1)$$

Mamy $a = 1$, $b = 3/2$, $f(n) = \Theta(1)$, a $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Zastosowano przypadek (2), gdyż $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, stąd $T(n) = \Theta(\lg n)$.

Oznacza to, że czas działania procedury *Przywroc_kopiec* zależy od wysokości kopca $h = \lg n$, jeśli jest wywołana w węźle o wysokości h.

b) czas działania procedury *Zbuduj_kopiec* (pętla *for*) można wyrazić jako:

$$T1(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}).$$

Wyrażenie $n/2^{h+1}$ wyznacza liczbę węzłów o wysokości h w kopcu zawierającym n elementów. Czas działania algorytmu *Przywroc_kopiec* w węźle o wysokości h **wynosi $O(h)$.**

Dla n $\to\infty$ mamy $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$, stąd czas działania procedury *Zbuduj_kopiec* jest wyrażony jako:

$$T1(n) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

czyli tworzenie kopca w nieuporządkowanej tablicy jest wykonywane w czasie liniowym **O(n)**

c) czas działania procedury *Sort_kopiec* jest równy *O(n lg n)*, gdyż mamy:

$$T(n) = T1(n) + (n-1)\, T2(n) = O(n) + (n-1)\, O(\lg n) \le O(n \lg n)$$

gdzie $T1(n)$ jest czasem działania procedury *Zbuduj_kopiec,* a czas działania procedury *Przywroc_kopiec*, wywoływanej $n-1$ razy, jest określony wyrażeniem $T2(n)$.

5. Wydajność algorytmów sortowania - podsumowanie
(wg R. Sedgewick: Algorytmy w C++)

rodzaj\n	12500	25000	50000	100000	200000	400000	800000
szybkie	2	5	11	24	52	109	241
łączenie zstępujące	5	12	23	53	111	237	524
łączenie wstępujące	5	11	26	59	127	267	568

rodzaj\n	12500	25000	50000	100000	200000	400000	800000
szybkie	2	7	13	27	58	122	261
łączenie zstępujące	5	11	24	52	111	238	520
kopcowanie zstępujące	3	8	18	42	100	232	547

rodzaj\n	12500	25000	50000	100000	200000	400000	800000
szybkie	2	5	10	21	49	102	223
pozycyjne od cyfry najmniej znaczącej D=16	5	8	15	30	56	110	219
pozycyjne od cyfry najbardziej znaczącej D=16	52	54	58	67	296	119398	1532492