Algorytmy i struktury danych

Wykład 10
Algorytmy sortowania wewnętrznego
Algorytmy sortowania zewnętrznego

Temat: Sortowanie wewnętrzne

Metody sortowania wewnętrznego

Podstawowe algorytmy sortowania

- ◆Sortowanie przez wstawianie (*ang. insertion sort*)
- ◆Sortowanie przez wybieranie (selekcję) (*ang. selection* sort)
- ◆Sortowanie przez zamianę (bąbelkowe) (*ang. exchange sort, bubble sort*)

Efektywne algorytmy sortowania

- ◆Sortowanie przez kopcowanie (*ang. heap sort*)
- ◆Sortowanie metodą malejących przyrostów (Shella)
- ◆Sortowanie szybkie (*ang. quicksort*)
- ◆Sortowanie przez scalanie (*ang. merge sort*)

Klucze początkowe	44	55	12	42	94	18	06	67

Sortowanie przez wstawianie

Kroki algorytmu

Klucze początkowe	44	55	12	42	94	18	06	67
i = 2	44	55	12	42	94	18	06	67
i = 3	12	44	55	42	94	18	06	67
i = 4	12	42	44	55	94	18	06	67
i = 5	12	42	44	55	94	18	06	67
i = 6	12	18	42	44	55	94	06	67
i = 7	06	12	18	42	44	55	94	67
i = 8	06	12	18	42	44	55	67	94

Idea algorytmu sortowania przez wstawianie

```
void InsertionSort (int tab[ ], int n) {
for ( i=1;  i<n; i++)  {
    przesuń wszystkie elementy tab[j], j<i, większe od tab[i] o jedną
    pozycję w prawo;
    umieść tab[i] w odpowiednim miejscu;
    }
}
```


Przykładowy algorytm sortowania przez wstawianie

```
void InsertionSort (int tab[ ], int n) {
    for ( int i=1, j;  i<n; i++)  {
          int temp=tab[i];
          for (j=i; j>0 && temp<tab[j-1]; j--)
                  tab[j]= tab[j-1];
          tab[j]=temp;
    }
}
```

Sortowanie przez wybieranie

Kroki algorytmu

Klucze początkowe	44	55	12	42	94	18	06	67
i = 2	06	55	12	42	94	18	44	67
i = 3	06	12	55	42	94	18	44	67
i = 4	06	12	18	42	94	55	44	67
i = 5	06	12	18	42	44	55	94	67
i = 6	06	12	18	42	44	55	67	94

Idea algorytmu sortowania przez wybieranie

```
void SelectionSort (int tab[ ], int n) {
    for (i=0; i<n-1; i++)  {
          wybierz najmniejszy element spośród tab[i], ..., tab[n-1];
          zamień wybrany element z tab[i];
    }
}
```


Przykładowy algorytm sortowania przez wybieranie

```
void SelectionSort (int tab[ ], int n) {
    int i, min, j;
    for ( i=0; i<n-1; i++ ) {
          for (j=i+1, min=i; j<n; j++)
                    if (tab[j] < tab[min])
                    min=j;
          // zamiana (tab[min], tab[i]);
          int temp= tab[min];
          tab[min] = tab[i];
          tab[i ]=temp;
    }
}
```


Sortowanie przez zamianę (bąbelkowe)

Kroki algorytmu

Klucze początkowe	i = 2	i = 3	i = 4	i = 5	i = 6	i = 7	i = 8
44	06	06	06	06	06	06	06
55	44	12	12	12	12	12	12
12	55	44	18	18	18	18	18
42	12	55	44	42	42	42	42
94	42	18	55	44	44	44	44
18	94	42	42	55	55	55	55
06	18	94	67	67	67	67	67
67	67	67	94	94	94	94	94

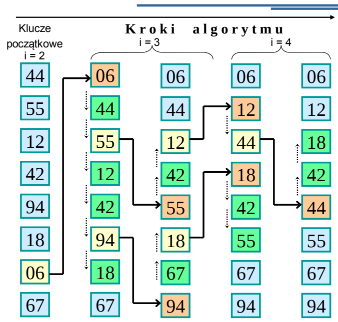
Idea algorytmu sortowania przez zamianę

```
void BubbleSort (int tab[ ], int n) {
    for (i=0; i<n-1; i++)
          for (j=n–1; j>i; j--)
                    jeśli tab[j] < tab[j-1] , zamień je miejscami;
}
```


Przykładowy algorytm sortowania przez zamianę

```
void BubbleSort (int tab[ ], int n) {
    for (int i=0; i<n-1; i++)
          for (int j=n-1; j>i; j--)
                    if  (tab[j]<tab[j-1])                    {
                              // zamiana (tab[j-1] ,  tab[j])
                              int temp= tab[j-1];
                              tab[j-1] = tab[j];
                              tab[j]=temp;
                    }
}
```

Sortowanie mieszane



Przykładowy algorytm sortowania mieszanego

```
void ShakeSort (int tab[ ], int n) {
int left=0, right=n-1, k=n-1;
do {
        for (int j= right; j>left; j--)
                if  (tab[j-1]>tab[j ])
                {
                        zamiana (tab[j-1], tab[j ]);
                        k=j;
                }
        left=k+1;
        for (j= left; j<right; j++)
                if  (tab[j-1]>tab[j ])
                {
                        zamiana (tab[j-1], tab[j ]);
                        k=j;
                }
        right=k-1;
    } while (left<right);
}
```


Złożoność obliczeniowa algorytmów sortowania

- Sortowanie przez wstawianie

Przypadek optymistyczny: dane są właściwie posortowane

```
void InsertionSort (int tab[ ], int n) {
  for ( int i=1, j;  i<n; i++) {
        int temp=tab[i];
        for (j=i; j>0 && temp<tab[j-1]; j--)
              tab[j]= tab[j-1];
        tab[j]=temp;
  }
}
```

Dlaczego?

$$T_{O\,min} = \sum_{i=1}^{n-1} 1 = n-1 = O(n)$$

Dlaczego?

$$T_{R\,min} = \sum_{i=1}^{n-1} 2 = 2(n-1) = O(n)$$

n - liczba elementów w tablicy *tab*
T_O - liczba porównań związanych z *tab*
T_R - liczba przypisań związanych z *tab*

Złożoność obliczeniowa algorytmów sortowania

- Sortowanie przez wstawianie

Przypadek pesymistyczny: dane są posortowane w odwrotnej kolejności

```
void InsertionSort (int tab[ ], int n)
{
for ( int i=1, j;  i<n; i++)
   {
        int temp=tab[i];
        for (j=i; j>0 && temp<tab[j-1]; j--)
              tab[j]= tab[j-1];
        tab[j]=temp;
   }
}
```

$$T_{Omax} = \sum_{i=1}^{n-1}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n-1} i = 1 + 2 + \ldots + n-1 = \frac{n(n-1)}{2} = 0{,}5\,(n^2 - n) = O(n^2)$$

$$T_{Rmax} = \sum_{i=1}^{n-1}\left(1 + \sum_{j=1}^{i} 1 + 1\right) = 2(n-1) + \sum_{i=1}^{n-1} i = 2(n-1) + 1 + 2 + \ldots + n-1 = 2(n-1) + \frac{n(n-1)}{2} = 0{,}5\,(n^2 + 3n - 4) = O(n^2)$$

Złożoność obliczeniowa algorytmów sortowania

- Sortowanie przez wstawianie

Przypadek „średni”

$T_{Ośr} = 0.25\,(n^2 + n - 2)$ $T_{Rśr} = 0.25\,(n^2 + 5n - 6)$

n - liczba elementów w tablicy,
T_O - liczba porównań klucza,
T_R - liczba przesunięć elementów w tablicy

Złożoność obliczeniowa algorytmów sortowania, cd.

- Sortowanie przez wybieranie

Przypadek optymistyczny = Przypadek pesymistyczny

```
void SelectionSort (int tab[ ], int n) {
    for (int i=0, min, j; i<n-1; i++)    {
          for (j=i+1, min=i; j<n; j++)
                  if (tab[j] < tab[min])
                          min=j;
          //  zamiana (tab[min], tab[i]);
          int temp= tab[min];
          tab[min] = tab[i];
          tab[i ]=temp;
    }
}
```

$$T_{O\,min} = T_{O\,max} = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1-i) = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = O(n^2)$$

Dlaczego?

$$T_{R\,min} = T_{R\,max} = 3(n-1) = O(n)$$

n - liczba elementów w tablicy *tab*
T_O - liczba porównań związanych z *tab*
T_R - liczba przypisań związanych z *tab*

Złożoność obliczeniowa algorytmów sortowania, cd.

- Sortowanie przez zamianę (bąbelkowe)
  Przypadek optymistyczny

$$T_{O\,min} = T_{O\,max} = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1-i) = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = O(n^2)$$

```
void BubbleSort (int tab[ ], int n) {
   for (int i=0; i<n-1; i++)
      for (int j=n–1; j > i; j--)
               if  (tab[j]<tab[j –1]) {
                     //  zamiana (tab[j –1] ,  tab[j ])
                     int temp= tab[j –1];
                     tab[j –1] = tab[j ];
                     tab[j ]=temp;
               }
}
```

Dlaczego?

$$T_{R\,min} = 0$$

n - liczba elementów w tablicy *tab*
T_O - liczba porównań związanych z *tab*
T_R - liczba przypisań związanych z *tab*

Złożoność obliczeniowa algorytmów sortowania, cd.

- Sortowanie przez zamianę (bąbelkowe)
  Przypadek pesymistyczny

$$T_{O\,min} = T_{O\,max} = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1-i) = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = O(n^2)$$

```
void BubbleSort (int tab[ ], int n) {
   for (int i=0; i<n-1; i++)
      for (int j=n–1; j>i; j--)
               if  (tab[j]<tab[j –1]) {
                     //  zamiana (tab[j –1] ,  tab[j ])
                     int temp= tab[j –1];
                     tab[j –1] = tab[j ];
                     tab[j ]=temp;
               }
}
```

$$T_{R\,max} = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 3 = O(n^2)$$

Złożoność obliczeniowa algorytmów sortowania, cd.

- Sortowanie przez zamianę (bąbelkowe)
  Przypadek „średni”

$T_O = 0{,}5\ (n^2 - n)$ $\qquad$ $T_{Rśr} = 0{,}75\ (n^2 - n)$

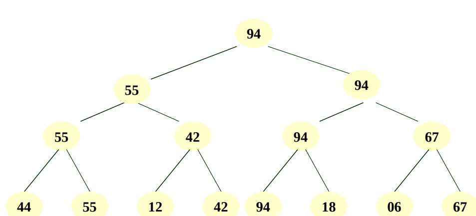
n - liczba elementów w tablicy,
T_O - liczba porównań klucza,
T_{Ri} - liczba przesunięć elementów w tablicy

Metody sortowania wewnętrznego

Efektywne algorytmy sortowania

- Sortowanie przez kopcowanie (*ang. heap sort*)
- Sortowanie metodą malejących przyrostów (sortowanie *Shella*)
- Sortowanie szybkie (*ang. quicksort*)
- Sortowanie przez scalanie (*ang. merge sort*)

Sortowanie drzewiaste

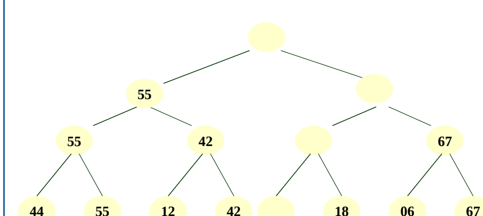


Tablica wynikowa:

...		94

Wyznaczenie największego klucza poprzez wielokrotny wybór

Sortowanie drzewiaste

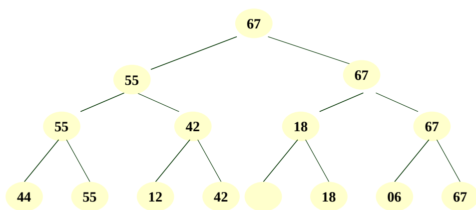


Tablica wynikowa:

...		94

Zdjęcie z kopca największego klucza

Sortowanie drzewiaste

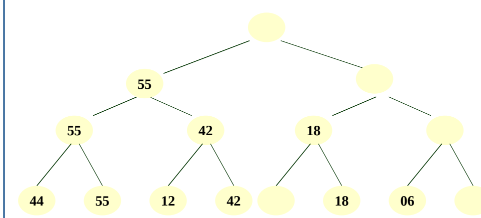


Tablica wynikowa:

...		94

Utworzenie nowego kopca

Sortowanie drzewiaste

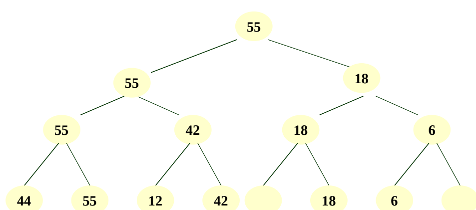


Tablica wynikowa:

...	67	94

Zdjęcie z kopca największego klucza

Sortowanie drzewiaste



Tablica wynikowa:

...	67	94

Utworzenie kolejnego kopca itd.

Sortowanie przez kopcowanie

- Metodę sortowania drzewiastego udoskonalił J.Williams (1964)
- Sortowanie przez kopcowanie (ang. *heap sort*) dowolnej tablicy A składa się z dwóch etapów:
  1. Najpierw tablicę A przetwarzamy na tablicę reprezentującą kopiec
  2. Następnie w pętli od *0* do *n-1* wywołujemy funkcję usuwania z kopca (tablicy A) elementu największego;
- Po wykonaniu ostatniego punktu tablica A posortowana jest rosnąco

Sortowanie przez kopcowanie

Przekształcanie tablicy w kopiec (tzw. metodą wstępującą R.Floyda (1964)):

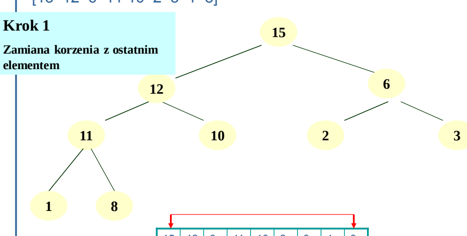
for (i=*indeks ostatniego węzła-nie liścia*; i>=0; i--)
- **odtwórz warunki kopca dla drzewa, którego korzeniem jest data[i] wywołując funkcję MoveDown(data, i, n-1);**

Sortowanie przez kopcowanie

Działanie algorytmu sortowania przez kopcowanie dla tablicy [15 12 6 11 10 2 3 1 8]

Krok 1

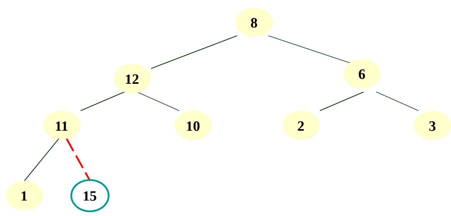
Zamiana korzenia z ostatnim elementem



Sortowanie przez kopcowanie

Krok 2

Zdjęcie z kopca elementu największego

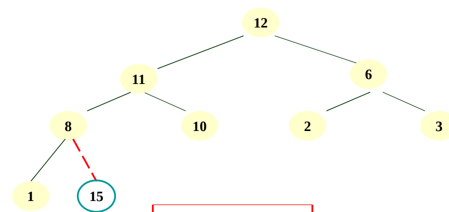


8	12	6	11	10	2	3	1	15

Sortowanie przez kopcowanie

Krok 3

Przywrócenie drzewu własności kopca

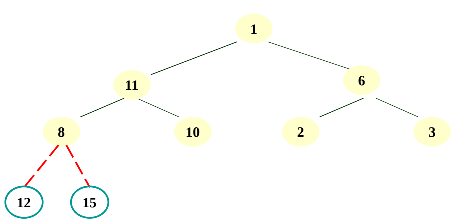


12	11	6	8	10	2	3	1	15

Sortowanie przez kopcowanie

Krok 4

Zamiana korzenia z ostatnim elementem

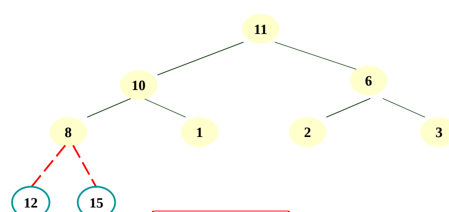


1	11	6	8	10	2	3	12	15

Sortowanie przez kopcowanie

Krok 5

Przywrócenie drzewu własności kopca

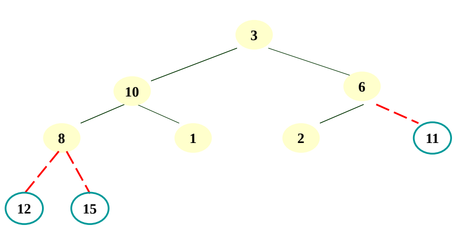


11	10	6	8	1	2	3	12	15

Sortowanie przez kopcowanie

Krok 6

Zamiana korzenia z ostatnim elementem

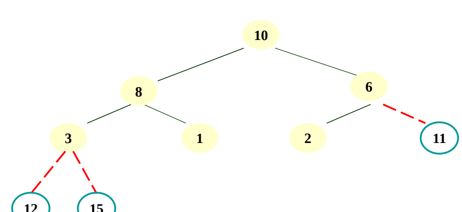


3	10	6	8	1	2	11	12	15

Sortowanie przez kopcowanie

Krok 7

Przywrócenie drzewu własności kopca

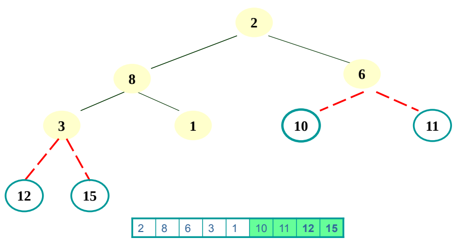


10	8	6	3	1	2	11	12	15

Sortowanie przez kopcowanie

Krok 8

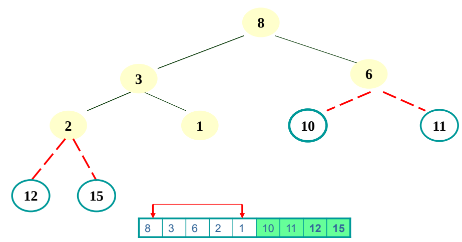
Zamiana korzenia z ostatnim elementem



Sortowanie przez kopcowanie

Krok 9

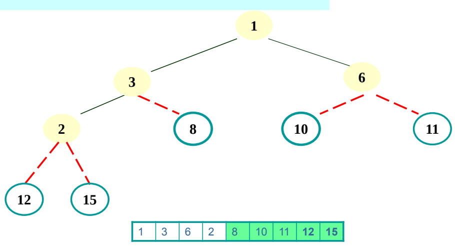
Przywrócenie drzewu własności kopca



Sortowanie przez kopcowanie

Krok 10

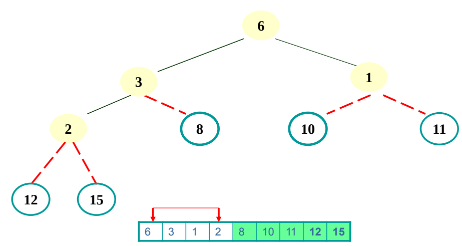
Zamiana korzenia z ostatnim elementem



Sortowanie przez kopcowanie

Krok 11

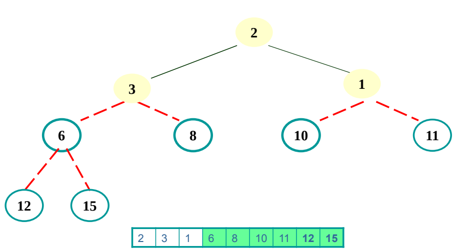
Przywrócenie drzewu własności kopca



Sortowanie przez kopcowanie

Krok 12

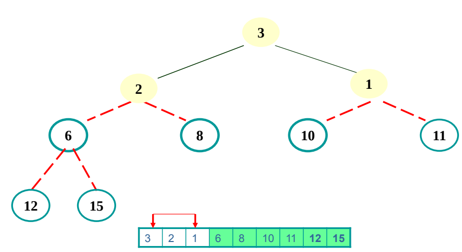
Zamiana korzenia z ostatnim elementem



Sortowanie przez kopcowanie

Krok 13

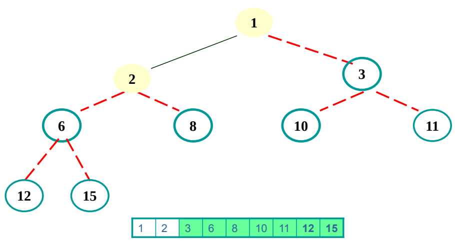
Przywrócenie drzewu własności kopca



Sortowanie przez kopcowanie

Krok 14

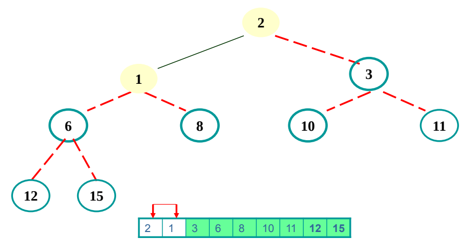
Zamiana korzenia z ostatnim elementem



Sortowanie przez kopcowanie

Krok 15

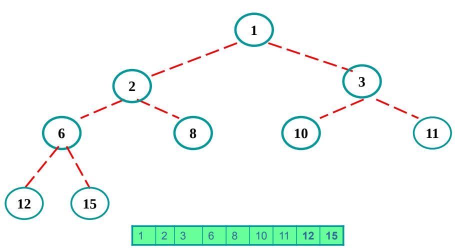
Przywrócenie drzewu własności kopca



Sortowanie przez kopcowanie

Krok 16

Zamiana korzenia z ostatnim elementem – sortowanie zakończone



Sortowanie przez kopcowanie

Idea algorytmu sortowania przez kopcowanie (*ang. heap sort*):

heapsort(data[ ] ,n)
1) **przekształć tablicę w kopiec;**
2) **for (i=n-1; i>1; i--) {**
   zamień korzeń z elementem na pozycji i;
   odtwórz własność kopca dla drzewa data[0], data[1], ..., data[i-1];
   }

Sortowanie przez kopcowanie

Algorytm sortowania przez kopcowanie

```
void heapsort( T data[ ], int n)
{
    int i;
    for (i=n/2-1;  i>=0;  i--)          // utworzenie kopca
          MoveDown(data, i, n-1);
    for (i=n-1;  i>=1;  i--)
    {
      Swap(data[0],data[i]);   // zdjęcie korzenia z kopca
      MoveDown(data, 0, i -1); // przywrócenie własności kopca
    }
}
```


Sortowanie przez kopcowanie

Analiza metody sortowania przez kopcowanie

- dla dużych *n* metoda jest bardzo efektywna
- złożoność czasowa algorytmu (nawet w najgorszym przypadku):
  T(n)= O(n lg n)

Sortowanie metodą malejących przyrostów (Shella)

- **Podstawą działania algorytmu sortowania Shella jest dzielenie tablicy na podtablice, zawierające elementy oddalone od siebie o *h* pozycji;**
- **Idea algorytmu:**

  określ liczby h_t, h_{t-1}, ..., h_1=1 określające sposób podziału tablicy *Tab* na podtablice;
  for (h= h_t; t > 1; t--, h= h_t)
  podziel *Tab* na h podtablic;
  for (i=1; i≤h; i++)
  posortuj i-tą podtablicę; // dowolną metodą
  posortuj tablicę *Tab*; // dowolną metodą

Sortowanie metodą malejących przyrostów (Shella)

- **Sortowanie podtablic może być realizowane dowolną metodą sortowania (najczęściej wykorzystuje się sortowanie przez wstawianie);**
- **Tablica wyjściowa dzielona jest na h_t podtablic, tworzonych co h_t-ty element; powstaje zatem h_t podtablic, przy czym dla każdej wartości *h=1, 2, ..., h_t* zachodzi:**

$$Tab_{h_t h}[i] = Tab[i \cdot h_t + h - 1]$$

- np. dla h_t=3 tablica Tab = [Tab [0], Tab [1], Tab [2], Tab [3],...] zostanie podzielona na 3 podtablice: Tab1, Tab2 oraz Tab3 w następujący sposób:
  Tab_{31}[0]=Tab[0], Tab_{31}[1]=Tab[3], ..., Tab_{31}[i]=Tab[3*i], ...
  Tab_{32}[0]=Tab[1], Tab_{32}[1]=Tab[4], ..., Tab_{32}[i]=Tab[3*i+1], ...
  Tab_{33}[0]=Tab[2], Tab_{33}[1]=Tab[5], ..., Tab_{33}[i]=Tab[3*i+2], ...
- wszystkie podtablice sortowane są niezależnie;
- następnie tworzone są nowe podtablice, przy czym h_{t-1}< h_t

Sortowanie metodą malejących przyrostów (Shella)

Przykład

- Tab=[5, 3, 6, 7, 2, 1, 9, 4, 8]
- h_t=3
- Wówczas:

  Tab1=[5, 7, 9]
  Tab2=[3, 2, 4]
  Tab3=[6, 1, 8]

Ile byłoby podtablic, gdyby Tab=[5, 3, 6, 7, 2, 1, 9, 4, 8, 4]?

Sortowanie metodą malejących przyrostów (Shella)

Przykład sortowania metodą malejących przyrostów (Shella) z przyrostami: 5, 3, 1

		10	8	6	20	4	3	22	1	0	15	16
h=5	1	10					3					16
	2		8					22				
	3			6					1			
	4				20					0		
	5					4					15	
	1	3					10					16
	2		8					22				
	3			1					6			
	4				0					20		
	5					4					15	
		3	8	1	0	4	10	22	6	20	15	16
h=3	1	3			0			22			15	
	2		8			4			6			16
	3			1			10			20		
	1	0			3			15			22	
	2		4			6			8			16
	3			1			10			20		
		0	4	1	3	6	10	15	8	20	22	16
h=1	1	0	1	3	4	6	8	10	15	16	20	22

Sortowanie metodą malejących przyrostów (Shella)

- Algorytm Shella ma dwie cechy, które mogą ulegać zmianie w różnych implementacjach:
  - sekwencja wartości przyrostów,
  - algorytm sortowania wykorzystywany do sortowania podtablic;
- Dobrym rozwiązaniem (wynikającym z praktyki) jest wykorzystanie sekwencji przyrostów spełniających warunki:

  h_1 = 1
  h_{i+1} = 3 h_i + 1

  oraz zakończenie dzielenia tablicy na podtablice dla takiego h_i dla którego zachodzi h_{i+2} ≥ n;
- np. jeśli n=10000 sekwencja przyrostów ma postać:

  1, 4, 13, 40, 121, 364, 1093, 3280
  tzn. i=8; h_9= 3 h_8 + 1=9841; h_{10}= 3 h_9 + 1=29524

Sortowanie metodą malejących przyrostów (Shella)

- **O złożoności algorytmu decyduje h_t i sposób zmiany *h***
  - h_t = 1 oznacza zwykłe sortowanie (np. przez wstawianie)
  - dla h_{i+1} = 3 h_i + 1 (...., 40, 13, 4, 1)
    algorytm ma złożoność **O($n^{1.25}$)**
  - dla h_i = 2^i - 1 (..., 31, 15, 7, 3, 1)
    algorytm ma złożoność **O($n^{1.2}$)**

Sortowanie metodą malejących przyrostów (Shella)

Algorytm sortowania metodą malejących przyrostów (Shella)

```
void ShellSort(T Tab[ ],  int Size) {
register int i, j hCnt, h;
int increments[20], k;  // stworzenie odpowiedniej liczby przyrostów h
  for (h=1,  i=0;  h<Size;  i++){
       increments[i]=h;
       h=3*h+1;
  }
  for (i--;  i≥0;  i--){  // pętla po wszystkich różnych przyrostach h
       h= increments[i];
       for (hCnt=h;  hCnt<2*h;  hCnt++){  // pętla po liczbie podtablic „co h” w danym przebiegu
             for (j=hCnt;  j<Size;){  // sortowanie przez wstawianie podtablicy
                        T tmp=Tab[j];
                        k=j;
                        while (k-h>=0 && tmp<Tab[k-h]){
                                 Tab[k]=Tab[k-h];
                                 k-=h;
                        }
                        Tab[k]=tmp;
                        j+=h;
             }
       }
  }}
```


Sortowanie metodą malejących przyrostów (Shella)

- Złożoność obliczeniowa algorytmu sortowania metodą malejących przyrostów (Shella) - **$O(n^{1,25})$**

$$O(n\log(n)) \le O(n^{1,25}) \le O(n^2)$$

Wykres

Sortowanie szybkie (quicksort)

Sortowanie szybkie (quicksort)

- Metoda opracowana przez C.A.R. Hoarea
- Podobnie jak metoda Shella zakłada dekompozycję tablicy na mniejsze podtablice (które jest łatwiej posortować)

Sortowanie szybkie (quicksort)

Idea metody:

Krok 1. Rozdzielić elementy danego ciągu $e_1, e_2, \ldots, e_n$ na dwie części względem pewnego ustalonego elementu, tzw. **mediany (elementu osiowego)**, tak aby na lewo od niego znajdowały się elementy mniejsze, a na prawo elementy większe.

Krok 2. Posortować elementy na lewo od mediany.

Krok 3. Posortować elementy znajdujące się na prawo od mediany.

Sortowanie szybkie - przykład

10 5 7 8 14 12 3 4 1

Rozdzielanie ze względu na wybraną medianę: 5 7 8 3 4 1 10 14 12

Stosujemy rekurencyjnie tę samą zasadę do obu części: 3 4 1 5 7 8 12 14

1 3 4 7 8

Końcowy rezultat: 1 3 4 5 7 8 10 12 14

Sortowanie szybkie (quicksort)

- **Algorytm *QuickSort* jest typowym algorytmem rekurencyjnym;**
- **W celu podzielenia tablicy konieczne jest wykonanie dwóch operacji:**
  - **znalezienie elementu osiowego (wyznaczającego podział tablicy na dwie podtablice),**
  - **przejrzeniu tablicy w celu umieszczenia jej elementów we właściwych podtablicach;**
- **Wybór dobrego elementu osiowego nie jest zadaniem łatwym (obie podtablice powinny mieć zbliżoną wielkość);**
- **Najczęściej stosowane strategie wyboru elementu osiowego:**
  - **wybranie pierwszego elementu tablicy;**
  - **wybranie elementu znajdującego się pośrodku tablicy.**

Sortowanie szybkie (quicksort)

Algorytm QuickSort

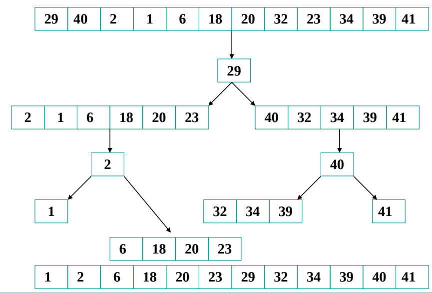
```
void QuickSort(int Tab[ ], int left, int right) {
  if (left < right) {
      int med=left;
      for ( int i=left+1;  i < right;  i++ )
           if ( Tab[i] < Tab[left] )
                    Zamiana( Tab[++med], Tab[i] );
      Zamiana( Tab[left], Tab[med] );
      QuickSort(Tab, left, med-1);
      QuickSort(Tab, med+1, right);
  }
}
```

Program 12.1

Program 12.2

Sortowanie szybkie (quicksort)

Algorytm QuickSort - przykład



Sortowanie szybkie – przypadek pesymistyczny

- Przypadek pesymistyczny zachodzi wówczas, gdy wektor jest uporządkowany (np. rosnąco lub malejąco)
- Pesymistyczna złożoność algorytmu QuickSort mierzona **liczbą porównań wartości** wynosi **O(n²)**

Jeśli jako medianę wybiera się zawsze pierwszy element, to w wyniku rozdzielenia, jedna część „młodsza" będzie pusta , a druga „starsza" będzie zawierała o jeden element mniej niż w poprzednim kroku.

Koszt operacji rozdzielania dla n elementowego ciągu wynosi n-1 porównań.

Złożoność pesymistyczna: $T(n) = n - 1 + T(n-1) = \sum_{i=2}^{n}(i-1) = O(n^2)$

Sortowanie szybkie – przypadek optymistyczny

- Założenie: **n=2^k^, k=1,2,3,...**
- Przypadek optymistyczny ma miejsce wówczas, gdy tablice tablica jest dzielona na równe części
- Średnia złożoność algorytmu QuickSort, wynosi wówczas **T(n) = O(n lg n)**
- Uzasadnienie: liczba porównań w kolejnych iteracjach pętli wynosi:

$$T(n) = n + 2\frac{n}{2} + 2\cdot 2\frac{n}{4} + 2\cdot 2\cdot 2\frac{n}{8} + \ldots + n\frac{n}{n} =$$
$$= n + \sum_{k=1}^{\lg n} n = n(1 + \lg n)$$

Złożoność optymistyczna: $T(n) = O(n \lg n)$

Sortowanie szybkie – przypadek średni

Z badań wynika, że średnia złożoność algorytmu QuickSort jest bliższa przypadkowi optymistycznemu, tzn. wynosi

T(n) = O(n lg n)

Sortowanie przez scalanie (MergeSort)

Sortowanie przez scalanie (MergeSort)

- Jeden z pierwszych algorytmów sortowania komputerowego;
- Autor metody: John von Neumann
- Podobnie jak metoda Shella algorytm sortowania przez scalanie zakłada podział tablicy na dwie mniejsze podtablice (które jest łatwiej posortować)

Sortowanie przez scalanie

Idea metody:

(1) Dzielimy sortowany ciąg na dwa podciągi;

(2) Sortujemy lewy i prawy podciąg (rekurencja);

(2) Scalamy dwa uporządkowane podciągi otrzymując posortowany ciąg wyjściowy.

Sortowanie przez scalanie - przykład

16 5 12 4 10 6 1 13 15 7 1 14 9 3 8 11

16 5 12 4 10 6 1 13 | 15 7 1 14 9 3 8 11

16 5 12 4 | 10 6 1 13 | 15 7 1 14 | 9 3 8 11

16 5 | 12 4 | 10 6 | 1 13

5 16 | 4 12 | 6 10 | 1 13

4 5 12 16 | 1 6 10 13

1 4 5 6 10 12 13 16 | 1 3 7 8 9 11 14 15

1 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16

itd.

Sortowanie przez scalanie (MergeSort)

Idea scalania dwóch podtablic w jedną (obie podtablice są uporządkowane):

1 4 5 6 10 12 13 16 | 1 3 7 8 9 11 14 15

1 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Sortowanie przez scalanie (MergeSort)

Idea scalania dwóch podtablic w jedną (obie podtablice są uporządkowane):

```
scal(Tab, pierwszy, ostatni) {
srd=(pierwszy+ostatni)/2;
i1=0;  i2=pierwszy;  i3=srd+1;
while (zarówno prawa, jak i lewa podtablica zawierają elementy)
        if (Tab[i2] <Tab[i3])
                pom[i1++]=Tab[i2++];
        else
                pom[i1++]= Tab[i3++];
umieść w pom pozostałe elementy Tab;
skopiuj zawartość pom do Tab;
}
```

Złożoność:
T(n)=O(n)

Sortowanie przez scalanie (MergeSort)

Idea algorytmu *MergeSort*:

```
MergeSort(Tab, pierwszy, ostatni) {
  if (pierwszy < ostatni) {
    srd=(pierwszy+ostatni)/2;
    MergeSort(Tab, pierwszy, srd);
    MergeSort(Tab, srd+1, ostatni);
    scal(Tab, pierwszy, ostatni);
  }
}
```


Sortowanie przez scalanie (MergeSort)

Złożoność obliczeniowa algorytmu *MergeSort*

- $$\begin{cases} T(1) = 0 \\ T(n) = 2T(\frac{n}{2}) + n \end{cases}$$
- Można pokazać, że:

T(n)=O(n lgn)

(także w przypadku pesymistycznym)

Efektywne algorytmy sortowania

Złożoność sortowania

Sortowanie zbioru n=1,000,000
(słowniki, książki telefoniczne, bazy danych)

Proste metody sortowania $O(n^2)$ (wstawianie, wybieranie, bąbelkowe)

Sprzęt	Czas
1mln op/s	⇒ 12 dni
100,000 op/s	⇒ 4 miesiące
10,000 op/s	⇒ 3 lata

Zaawansowane metody sortowania O(n*log(n)) (stogowe, przez podział)

Sprzęt	Czas
1mln op/s	⇒ 6 s
100,000 op/s	⇒ 1 min
10,000 op/s	⇒ 10 min

(op/s – dotyczy operacji na elementach sortowanego zbioru, które mogą być bardziej złożone niż elementarne operacje procesora)

Efektywne algorytmy sortowania

Złożoność sortowania

Sortowanie zbioru n=1,000,000,000
(symulacje fizyczne, astronomiczne, biologiczne)

Proste metody sortowania $O(n^2)$ (wstawianie, wybieranie, bąbelkowe)

Sprzęt	Czas
1mln op/s	⇒ 31700 lat

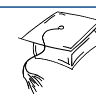
Zaawansowane metody sortowania O(n*log(n)) (stogowe, przez podział)

Sprzęt	Czas
1mln op/s	⇒ 2.5h
100,000 op/s	⇒ 1 dzień
10,000 op/s	⇒ 10 dni

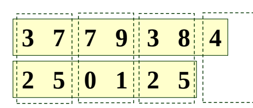
(op/s – dotyczy operacji na elementach sortowanego zbioru, które mogą być bardziej złożone niż elementarne operacje procesora)

Demonstracja algorytmów sortowania

- http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html



Temat: Sortowanie zewnętrzne



Sortowanie danych w plikach zewnętrznych

- Dane, które są przechowywane w plikach zewnętrznych, także mogą być sortowane; wymaga to stosowania innych metod sortowania;
- Wynika to przede wszystkim z innej organizacji dostępu do danych, które są przechowywane w plikach zewnętrznych, gdzie ogranicza nas przed wszystkim sekwencyjność ich zapisu;
- Omówione zostaną dwie metody sortowania danych w plikach zewnętrznych:
  - Metoda łączenia prostego
  - Metoda łączenia naturalnego

METODA ŁĄCZENIA PROSTEGO

- Metoda łączenia prostego polega na sukcesywnym dzieleniu danych w pliku na **podciągi (serie) o ustalonej długości maksymalnej** i łączeniu tych podciągów wraz z równoczesnym porządkowaniem zawartych w nich elementów w uporządkowane serie danych; porządek ten może być dowolny, np. rosnący lub malejący.
- Zapis logiki algorytmu sortowania metodą łączenia prostego może być następujący:

```
co_ile = 1;
do {
     dalej=0;
     rozdzielanie;
     if (rozdzielono) {
                    łączenie;
                    dalej = 1;
     }
     co_ile = 2 * co_ile;
} while (dalej=1);
```

METODA ŁĄCZENIA PROSTEGO - niemalejąco

- Sortowany plik:

3 7 2 5 7 9 1 0 8 3 2 5 4

- co_ile = 1 - ROZDZIELANIE SERII:

3 | 2 | 7 | 1 | 8 | 2 | 4

7 | 5 | 9 | 0 | 3 | 5

13 wygenerowanych podciągów, które potem będą łączone w ustalonym porządku sortowania

- co_ile = 1 - ŁĄCZENIE SERII – w ustalonym porządku sortowania:

3 7 | 2 5 | 7 9 | 0 1 | 3 8 | 2 5 | 4

ŁĄCZENIE PROSTE – niemalejąco, krok 2

- Zawartość sortowanego pliku:

3 7 2 5 7 9 0 1 3 8 2 5 4

- co_ile = 2 - ROZDZIELANIE SERII:

3 7 | 7 9 | 3 8 | 4

2 5 | 0 1 | 2 5

7 wygenerowanych podciągów liczb, które potem będą łączone w ustalonym porządku sortowania

- co_ile = 2 - ŁĄCZENIE SERII – w ustalonym porządku sortowania:

2 3 5 7 | 0 1 7 9 | 2 3 5 8 | 4

ŁĄCZENIE PROSTE – niemalejąco, krok 3

- Zawartość sortowanego pliku:

2 3 5 7 0 1 7 9 2 3 5 8 4

- co_ile = 4 - ROZDZIELANIE SERII:

2 3 5 7 | 2 3 5 8

0 1 7 9 | 4

4 wygenerowane podciągi liczb, które potem będą łączone w ustalonym porządku sortowania

- co_ile = 4 - ŁĄCZENIE SERII – w ustalonym porządku sortowania:

0 1 2 3 5 7 7 9 | 2 3 4 5 8

ŁĄCZENIE PROSTE – niemalejąco, krok 4

- Zawartość sortowanego pliku:

0 1 2 3 5 7 7 9 2 3 4 5 8

- co_ile = 8 - ROZDZIELANIE SERII:

0 1 2 3 5 7 7 9

2 3 4 5 8

2 wygenerowane serie liczb, które potem będą łączone w ustalonym porządku sortowania

- co_ile = 8 - ŁĄCZENIE SERII – w ustalonym porządku sortowania:

0 1 2 2 3 3 4 5 5 7 7 8 9

ŁĄCZENIE PROSTE – niemalejąco, krok 5

- Zawartość sortowanego pliku:

0 1 2 2 3 3 4 5 5 7 7 8 9

- co_ile = 16 - ROZDZIELANIE SERII:

0 1 2 2 3 3 4 5 5 7 7 8 9

- NIE ROZDZIELONO SERII – KONIEC ALGORYTMU SORTOWANIA PLIKU METODĄ ŁĄCZENIA PROSTEGO

- PLIK POSORTOWANY WYGLĄDA NASTĘPUJĄCO:

0 1 2 2 3 3 4 5 5 7 7 8 9

METODA ŁĄCZENIA NATURALNEGO

- Metoda łączenia naturalnego polega na sukcesywnym dzieleniu danych w pliku wg uporządkowanych podciągów i łączeniu tych podciągów wraz z równoczesnych porządkowaniem zawartych w nich elementów.
- Nie ustala się długości tych podciągów – zależy to wyłącznie od ułożenia wartości sortowanych elementów
- Zapis logiki algorytmu sortowania metodą łączenia naturalnego może być następujący:

```
do {
    dalej=0;
    rozdzielanie;
    if (rozdzielono) {
        łączenie;
        dalej = 1;
    }
} while (dalej=1);
```

METODA ŁĄCZENIA NATURALNEGO - niemalejąco

- Sortowany plik:

3 7 2 5 7 9 1 0 8 3 2 5 4

- ROZDZIELANIE SERII – podciągi o wartościach niemalejących:

3 7 | 1 | 3 | 4

2 5 7 9 | 0 8 | 2 5

7 serii, które ułożyły się w porządku niemalejącym

- ŁĄCZENIE SERII – w ustalonym porządku sortowania:

2 3 5 7 7 9 | 0 1 8 | 2 3 5 | 4

ŁĄCZENIE NATURALNE – niemalejąco, krok 2

- Zawartość sortowanego pliku:

2 3 5 7 7 9 0 1 8 2 3 5 4

- ROZDZIELANIE SERII – podciągi o wartościach niemalejących:

2 3 5 7 7 9 | 2 3 5

0 1 8 | 4

4 serie, które ułożyły się w porządku niemalejącym

- ŁĄCZENIE SERII – w ustalonym porządku sortowania:

0 1 2 3 5 7 7 8 9 | 2 3 4 5

ŁĄCZENIE NATURALNE – niemalejąco, krok 3

- Zawartość sortowanego pliku:

0 1 2 3 5 7 7 8 9 2 3 4 5

- ROZDZIELANIE SERII – podciągi o wartościach niemalejących:

0 1 2 3 5 7 7 8 9

2 3 4 5

2 serie, które ułożyły się w porządku niemalejącym

- ŁĄCZENIE SERII – w ustalonym porządku sortowania:

0 1 2 2 3 3 4 5 5 7 7 8 9

ŁĄCZENIE NATURALNE – niemalejąco, krok 4

- Zawartość sortowanego pliku:

0 1 2 2 3 3 4 5 5 7 7 8 9

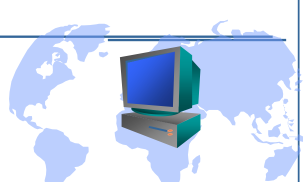
- ROZDZIELANIE SERII – podciągi o wartościach niemalejących:

0 1 2 2 3 3 4 5 5 7 7 8 9

- NIE ROZDZIELONO SERII – KONIEC ALGORYTMU SORTOWANIA PLIKU METODĄ ŁĄCZENIA NATURALNEGO

- PLIK POSORTOWANY:

0 1 2 2 3 3 4 5 5 7 7 8 9



Dziękuję za uwagę