

Laboratorium 11: Algorytmy sortowania.

1. Algorytm HeapSort

Algorytm sortowania przez kopcowanie składa się z dwóch faz. W pierwszej sortowane elementy reorganizowane są w celu utworzenia kopca. W drugiej zaś dokonywane jest właściwe sortowanie.

Faza I - tworzenie kopca

Podstawową zaletą algorytmu jest to, że do stworzenia kopca wykorzystać można tę samą tablicę, w której początkowo znajdują się nieposortowane elementy. Dzięki temu uzyskuje się stałą złożoność pamięciową. Efektywne tworzenie kopca w czasie liniowym zapewnia procedura *construct* korzystająca z pomocniczej procedury *downheap*.

Faza II - Sortowanie

Po utworzeniu kopca następuje właściwe sortowanie. Polega ono na usunięciu wierzchołka kopca, zawierającego element maksymalny (minimalny), a następnie wstawieniu w jego miejsce elementu z końca kopca i odtworzenie porządku kopcowego. W zwolnione w ten sposób miejsce, zaraz za końcem zmniejszonego kopca wstawia się usunięty element maksymalny. Operacje te powtarza się aż do wyczerpania elementów w kopcu. Sortowanie implementuje procedura *heapsort* wraz z pomocniczą funkcją *deletemax* oraz procedurą *downheap*.

(na podstawie Wikipedii i wykładów prof. Penczka)

```
procedure heapsort;
{a[1 .. n] - lista do posortowania}
var m,i : integer;
begin
  m := n;
  construct;
  for i := m downto 2 do
    a[i] := deletemax
  n := m
end heapsort
```

```
procedure construct;
{elementy listy q= [a_1, ..., a_n] znajdują
 się w tablicy a[1 .. n]}
var i : integer;
begin
  for i := n div 2 downto 1 do downheap(i)
end construct;

function deletemax : integer;
begin
  deletemax := a[1];
  a[1] := a[n];
  n := n - 1;
  downheap(1)
end deletemax;
```

```
procedure downheap(k : integer);
label L;
var j,v: integer;
begin
  v:= a[k];
  while k <= n div 2 do
    begin
      j := 2 * k; {j jest następnikiem k}
      if j < n then if a[j] < a[j + 1] then j := j + 1 ;
      if v >= a[j] then goto L;
      a[k] := a[j];
      k := j
    end;
  L: a[k]:= v
end downheap;
```

2. Sortowanie przez zliczanie.

Sortowanie przez zliczanie polega na sprawdzeniu ile wystąpień kluczy mniejszych od danego występuje w sortowanej tablicy. Na podstawie obliczonych liczników częstości umieszczamy każdy element a, we właściwym miejscu w ciągu wynikowym. Algorytm zakłada, że klucze elementów należą do skończonego zbioru (np. są to liczby całkowite z przedziału 0..100).

```
procedure countsort,
{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; {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-1 do count[j] := count[j-1] + count[j]; {count[j] to liczba wystąpień elementów <= j}
  for i := n downto 1 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;
```

3. Sortowanie szybkie - quicksort

Sortowanie szybkie to jeden z algorytmów sortowania działających na zasadzie "dziel i zwyciężaj", opracowany w 1962 przez Antony'ego Hoare'a. Algorytm działa rekurencyjnie. Najpierw zostaje wybrany pewien element tablicy, tzw. element osiowy. Następnie na początek tablicy przenoszone są wszystkie elementy mniejsze od osiowego, a na koniec wszystkie większe. W powstałe między tymi obszarami puste miejsca trafia wybrany element. Potem sortuje się osobno początkową i końcową część tablicy. Rekursja kończy się, gdy fragment uzyskany z podziału zawiera pojedynczy element, ponieważ tablica jednoelementowa jest zawsze posortowana. Poniższy pseudokod przedstawia ideę algorytmu quicksort:

```
PROCEDURE Quicksort(left, right)
  BEGIN
    IF left < right THEN                                //jeśli fragment dłuższy niż 1 element
      BEGIN
        i = PodzielTablice(left, right); //podziel i zapamiętaj punkt podziału
        Quicksort(left, i-1);           //posortuj lewą część
        Quicksort(i+1, right);          //posortuj prawą część
      END
    END
  END
```

Algorytm quicksort występuje w wielu odmianach. Znane są np. różne sposoby wyznaczania elementu osiowego. Jeden z prostszych to wybór pierwszego elementu tablicy, tak jak w poniższym pseudokodzie:

```
function PodzielTablice(l,r):integer;
  var v,i,j: integer;
  begin
    v := A[l];
    i := l;
    j := r + 1;
    while i < j do
      begin
        repeat i := i + 1
          until (A[i] >= v) // może być też >
        repeat j := j - 1;
          until (A[j] <= v) // może być też <
        if i < j then A[i] <-> A[j]; // zamiana elementów miejscami
        end;
      A[l] := A[j];
      A[j] := v;
      PodzielTablice := j // zwróć indeks element osiowego
    end;
```

4. Klasy potrzebne do rozwiązania zadań.

Dana jest klasa abstrakcyjna Sort2, którą należy pobrać bezpośrednio ze strony WWW.

5. Zadania

1. Utworzyć klasę *MySort2* dziedziczącą z klasy *Sort2*. Zaimplementować metody abstrakcyjne *wypelnij*, *losuj*, *wypisz*. Zaimplementuj metodę *wypiszDrzewo*, która wypisuje elementy tablicy w postaci drzewa zgodnie z numeracją węzłów kopca zupełnego. Napisać krótki program wykorzystujący wszystkie metody utworzonej klasy, który zaprezentuje ich działanie. [1p]
2. Zaimplementuj metody *downHeap* oraz *construct* ustawiające elementy tablicy zgodnie z porządkiem kopca. [2p]
3. Zaimplementuj metody *deleteMax* (usuwanie el. maksymalnego z kopca) oraz *heapSort* (sortowanie przez kopcowanie). [2p]
4. Zaimplementuj metodę *quicksort*. [3p]
5. Zaimplementuj metodę *countsort*. [2p]