Podstawy algorytmiki i programowania - wykład 6

Sortowanie- algorytmy

Treści prezentowane w wykładzie zostały oparte o:

- S. Prata, Język C++. Szkoła programowania. Wydanie VI, Helion, 2012
- *www.cplusplus.com*
- Jerzy Grębosz, Opus magnum C++11, Helion, 2017
- B. Stroustrup, Język C++. Kompendium wiedzy. Wydanie IV, Helion, 2014
- S. B. Lippman, J. Lajoie, Podstawy języka C++, WNT, Warszawa 2003.

Sortowanie szybkie - *quicksort*

- Strategia "dziel i zwyciężaj": polega na dzieleniu problemu na kilka mniejszych podproblemów podobnych do początkowego problemu. Problemy te rozwiązywane są rekurencyjnie, a następnie rozwiązania wszystkich podproblemów są łączone w celu utworzenia rozwiązania całego problemu.
- Szybkie sortowanie (ang. ***quicksort***) to jeden z popularnych algorytmów sortowania działających na zasadzie "dziel i zwyciężaj"
- Średnia złożoność obliczeniowa tego algorytmu jest rzędu O(n log n), pesymistyczna $O(n^2)$.
- Quicksort jest powszechnie używany. Jego implementacje znajdują się w bibliotekach standardowych wielu środowisk programowania.

Sortowanie szybkie – *quicksort* - algorytm

ALGORYTM

- Quicksort polega na dzieleniu tablicy na dwie części, a następnie rekurencyjnym sortowaniu każdej z nich.
- Tablica jest dzielona przez umieszczenie wszystkich elementów mniejszych niż pewien wybrany element (element rozdzielający) przed tym elementem oraz wszystkich elementów które są od niego większe za nim.
- Elementem rozdzialającym, progowym (ang. *pivot*) może być dowolny element np. element pierwszy, środkowy albo losowy.
- Potem sortuje się osobno obie części tablicy - podtablice.
- Rekurencja kończy się, gdy otrzymana z podziału podtablica jest jednoelementowa (czyli nie wymaga już sortowania).

Sortowanie szybkie – *quicksort* - algorytm

Ogólna strategia podziału tablicy:

1. wybieramy *pivot*,
2. przeglądamy tablicę od jej lewego końca, aż znajdziemy element większy niż *pivot*
3. przeglądamy tablicę od jej prawego końca, aż znajdziemy element mniejszy niż *pivot*
4. oba elementy, które zatrzymują przeglądanie tablicy, są na pewno nie na swoich miejscach w tablicy, więc je zamieniamy ze sobą

Sortowanie szybkie – *quicksort* - przykład

Rozważmy tablicę

l = 0	1	2	3	4	5	6	*p* = 7
	65	**21**	**48**	**87**	**5**	**59**	**74**

KROK 1: Wyznaczamy element rozdzielający **pivot** – element środkowy,

```
x=a[(0+7)/2]=a[3]=48
```

Ustawiamy liczniki `i` na pierwszy element tablicy, natomiast `j` na ostatni element tablicy:

l = 0	1	2	3	4	5	6	*p* = 7
32	**65**	**21**	**48**	**87**	**5**	**59**	**74**
i							***j***

Sortowanie szybkie – *quicksort* - przykład

KROK 2: W kolejnym kroku szukamy

- pierwszej liczby z lewej strony tablicy poruszając się licznikiem `i` ku wartości *pivot*, która jest nie mniejsza niż pivot (czyli dopóki elementy są mniejsze od `x` to zwiększamy `i`)
- oraz pierwszej liczby z prawej strony, która jest nie większa niż pivot (dopóki elementy są większe od `x` to zmniejszamy `j`).

Stan liczników po wyszukaniu odpowiednich liczb:

l = 0	1	2	3	4	5	6	*p* = 7
32	**65**	**21**	**48**	**87**	**5**	**59**	**74**
i →	***i***				***j*** ←	***j*** ←	***j***

Sortowanie szybkie – *quicksort* - przykład

Jeśli indeksy się nie minęły (czyli `i<=j`), to dokonujemy zamiany elementów : `65 <-> 5`, a następnie zwiększamy `i` (`i++`), i zmniejszamy `j` (`j--`):

l = 0	1	2	3	4	5	6	*p* = 7
32	**65**	**21**	**48**	**87**	**5**	**59**	**74**
	i				***j***		

Otrzymujemy

l = 0	1	2	3	4	5	6	*p* = 7
32	**5**	**21**	**48**	**87**	**65**	**59**	**74**
		i		***j***			

Czynności powtarzamy do momentu minięcia się liczników:

Sortowanie szybkie – *quicksort* - przykład

KROK 2: Ponownie szukamy

- pierwszej liczby z lewej strony tablicy która jest nie mniejsza niż *pivot* (czyli dopóki elementy są mniejsze od `x` to zwiększamy `i`)
- oraz pierwszej liczby z prawej strony, która jest nie większa niż *pivot* (dopóki elementy są większe od `x` to zmniejszamy `j`).

l = 0	1	2	3	4	5	6	*p* = 7
32	**5**	**21**	**48**	**87**	**65**	**59**	**74**
			i,j				

Sprawdzamy czy `i<=j` (czy się nie minęły)
u nas `i=j=3` zatem wymieniamy 48 z 48 i robimy `i++, j--`

l = 0	1	2	3	4	5	6	*p* = 7
32	**5**	**21**	**48**	**87**	**65**	**59**	**74**
		j		***i***			

Sortowanie szybkie – *quicksort* - przykład

Indeksy `i` i `j` się minęły- otrzymaliśmy podzial na podciągi

- zielony elementy`<=x`
- niebieski elementy `>=x`
- i być może środkowy : elementy `=x`

l = 0	1	2	3	4	5	6	*p* = 7
32	**5**	**21**	**48**	**87**	**65**	**59**	**74**
		j		***i***			

KROK 3. W kolejnym kroku wykonujemy sortowanie dwóch tablic `tab[l,j]`,`tab[i,p ]`osobno (wywołanie rekurencyjne), o ile nie są one jednoelementowe, czyli `l<j` i `i<p`

`Quicksort(0,2,a)`

`Quicksort(4,7,a)`

l = 0	1	*p*=2
32	**5**	**21**
i		***j***

l=4	5	6	*p* = 7
87	**65**	**59**	**74**
i			***j***

Sortowanie szybkie – *quicksort* - przykład

l = 0	1	*p*=2
32	**5**	**21**
i		***j***

`Quicksort(0,2,a)`

KROK 1. `i=l=0  j=p=2, x=a[(0+2)/2]=a[1]=5`

KROK 2:

- dopóki elementy są mniejsze od `x` to zwiększamy `i`
- dopóki elementy są większe od `x` to zmniejszamy `j`

l = 0	1	*p*=2
32	**5**	**21**
i	***j*** ←	***j***

Sprawdzamy czy `i<=j`, wymieniamy `32` z `5` i robimy `i++, j--`

l = 0	1	*p*=2
5	**32**	**21**
j	***i***	

Sortowanie szybkie – *quicksort* - przykład

Indeksy się minęły- otrzymaliśmy podział na podciągi, zatem

l = 0	1	*p*=2
5	**32**	**21**
j	***i***	

KROK 3 :

- `l=j` – tablica 1 elementowa- nie ma już czego sortować
- `i<p` wywołujemy quicksort rekrencyjnie

Quicksort(1,2,a)

KROK 1: `i=l=1` `j=p=2` **`x`**`=a[(1+2)/2]=a[1]=32`

l=1	*p*=2
32	**21**
i	***j***

KROK 2:

- dopóki elementy są mniejsze od `x` to zwiększamy `i`
- dopóki elementy są większe od `x` to zmniejszamy `j`

l=1	*p*=2
32	**21**
i	***j***

Sortowanie szybkie – *quicksort* - przykład

Sprawdzamy czy `i<=j` (czy się nie minely), wymieniamy `32` z `21` i robimy `i++, j--`

l=1	*p*=2
21	**32**
j	***i***

Indeksy się minęły- otrzymaliśmy podział na podciągi

KROK 3 :

- `l=j` – tablica 1 elementowa- nie ma już czego sortować
- `i=p` – tablica 1 elementowa- nie ma już czego sortować

Zatem lewa podtablica jest już posortowana:

0	1	2
5	**21**	**32**

Sortowanie szybkie – *quicksort* - przykład

Teraz prawa podtablica **`Quicksort(4,7,a)`**

l=4	5	6	*p* = 7
87	**65**	**59**	**74**
i			***j***

KROK 1: `i=l=4  j=p=7` **`x`**`=a[(4+7)/2]=a[5]=65`

KROK 2:

- dopóki elementy są mniejsze od `x` to zwiększamy `i`
- dopóki elementy są większe od `x` to zmniejszamy `j`

l=4	5	6	*p* = 7
87	**65**	**59**	**74**
i		***j*** ←	***j***

Sortowanie szybkie – *quicksort* - przykład

Sprawdzamy czy `i<=j` wymieniamy `87` z `59` i robimy `i++, j--`

l=4	5	6	*p* = 7
59	**65**	**87**	**74**
	i j		

`i<=j` zatem ponownie

KROK 2:

- dopóki elementy są mniejsze od `x` to zwiększamy `i`
- dopóki elementy są większe od `x` to zmniejszamy `j`

l=4	5	6	*p* = 7
59	**65**	**87**	**74**
	i j		

Sprawdzamy czy `i<=j` wymieniamy `65` z `65` i robimy `i++, j--`

Sortowanie szybkie – *quicksort* - przykład

Otrzymujemy

l=4	5	6	*p* = 7
59	**65**	**87**	**74**
j		***i***	

Indeksy się minęły- otrzymaliśmy podział na podciągi

KROK 3 :

- `l=j` – tablica 1 elementowa- nie ma już czego sortować
- `i<p` – wywołujemy quicksort rekurencyjnie Quicksort(6,7,a)

l=6	*p* = 7
87	**74**
i	***j***

KROK 1: `i=l=6  j=p=7` **`x`**`=a[(6+7)/2]=a[6]=87`

KROK 2:

- dopóki elementy są mniejsze od `x` to zwiększamy `i`
- dopóki elementy są większe od `x` to zmniejszamy `j`

l=6	*p* = 7
87	**74**
i	***j***

Sortowanie szybkie – *quicksort* - przykład

Sprawdzamy czy `i<=j` wymieniamy `87` z `74` i robimy `i++, j--`

l=6	*p* = 7
74	**87**
j	***i***

Indeksy się minęły- otrzymaliśmy podział na podciągi

KROK 3:

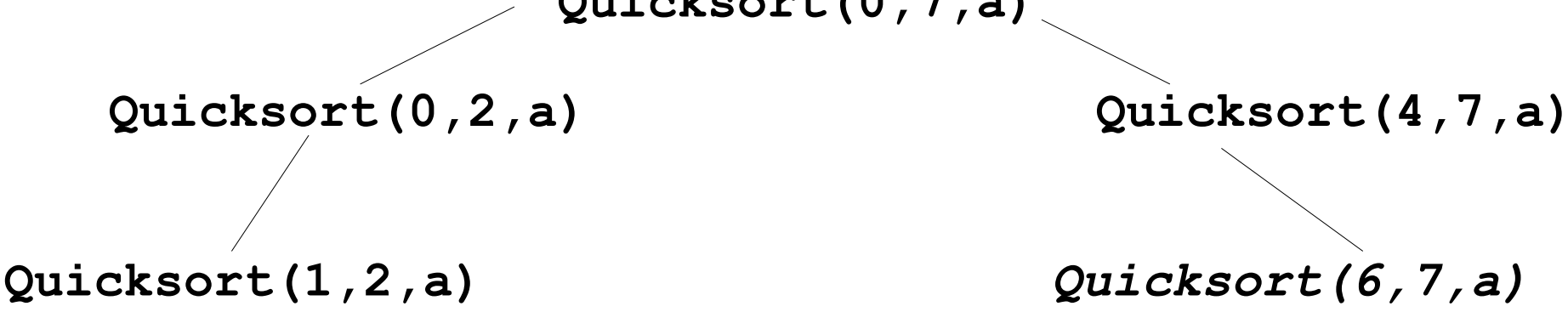
- `l=j` – tablica 1 elementowa- nie ma już czego sortować
- `i=p` – tablica 1 elementowa- nie ma już czego sortować

Zatem prawa podtablica jest posortowana

l=4	5	6	*p* = 7
59	**65**	**74**	**87**

Sortowanie szybkie – *quicksort* - przykład

PODSUMOWANIE: drzewo wywołań:



Sortowanie szybkie – *quicksort* - implementacja

```
#include <iostream>
#include <string>
#include <ctime>
using namespace std;

const int NMAX=20; // stała globalna określająca
                   //  max rozmiar tablic

//QuickSort- szybkie sortowanie
void quicksort(int l, int p, int a[])
{  //krok 1
   int i, j, pom;
   //punkt progowy (pivot)
   int x = a[(l+p)/2];
   i = l; //i ustawiamy na lewym
   j = p; //  j na prawym
```

Sortowanie szybkie – *quicksort* - implementacja

```
//dopóki indeksy się nie minęły
while(i<=j)
{
  // krok 2
  //najpierw z lewej strony,szukamy najmniejszego
  // indeksu i, takiego, ze a[i]>=x
  //czyli dopóki elementy są mniejsze od x to i++
  while (a[i] < x)
        i++;
  //teraz od prawej: szukamy największego indeksu
  // j, takiego, że a[i]<=x
  //czyli dopóki elementy są większe od x to
  // zmniejszamy j
   while (a[j] > x)
         j--;
```

Sortowanie szybkie – *quicksort* - implementacja

```
  // jeśli i<=j, to przestawiamy elementy
  //a[i] z a[j] i i++, j--
  if(i<=j)
  {
        pom  = a[i];
        a[i] = a[j];
        a[j] = pom;
        i++;
        j--;
  }
   //jesli nadal i<=j to wroc do kroku 2,
   // w przeciwnym przypadku do kroku 3
}//while
```

Sortowanie szybkie – *quicksort* - implementacja

```
    //krok 3
    // otrzymaliśmy podział na 3 podciągi
    // a[k]<=x dla l<=k<=j
    // a[k]= x dla j+1<=k<=i-1
    // a[k]>=x dla i<=k<=p
    //jeśli podtablice nie są jednoelementowe
    if (l<j)
         quicksort(l, j, a);//wywołanie rekurencyjne
    if (i<p)
         quicksort(i, p, a);//wywołanie rekurencyjne

}//quicksort
```

Sortowanie szybkie – *quicksort* - implementacja

```
//WYSWIETLANIE TABLICY
//f-cja wyswietla na standardowym wyjsciu tablice
// o n-elementach:[a_0,a_1,  ,a_{n-1}]
void printTab(int n, int t[])
{
  cout<<"[";
  for (int i=0; i < n; i++)
  {
    cout<<t[i];
    //jeśli nie jest to ostatni elem,to wyswietlamy ,
    if (i < n-1)
        cout<<",";
  }
  cout<<"]";
}
```

Sortowanie szybkie – *quicksort* - implementacja

```
//POBIERANIE TABLICY
//funkcja pobierajaca od uzytkownika  rozmiar n,
// a następnie elementy tablicy
void pobTab(int &n, int t[] )
{  //w pętli zaporowej pob.n z przedzialu [1,  NMAX]
  do{
        cout << "Podaj rozmiar tablicy: ";
        cin >> n;
  }while(n < 1 || n > NMAX);
  //pobieramy elementy tablicy: t[0],...,t[n-1]
   for(int i=0; i < n; i++)
   {
      cout << "Podaj "<<i<<" el tabl: ";
      cin >> t[i];
   }
}
```

Sortowanie szybkie – *quicksort* - implementacja

```
//GENEROWANIE LOSOWE TABLICY
//funkcja generuje losowo rozmiar tablicy,
//a nastepnie generuje losowo jej elementy z
// przedzialu [0,9]
/*int rand (void);
generuje liczbe losowa z przedziału 0 do RAND_MAX.
v1 = rand() % 100;    // v1 z przedziału 0 do 99
v2 = rand() % 100 + 1; // v2 z przedziału 1 do 100*/
void genTabLos (int &nmax, int t[] )
{
   //Dynamiczna wielkosc tablicy:
   nmax = rand()%NMAX + 1;
   //Wypelnienie tablicy liczbami (pseudo)losowymi
   for (int i=0; i<nmax; i++)
       t[i] = rand()%10;
}
```

Sortowanie szybkie – *quicksort* - implementacja

```
//CZY POSORTOWANA
/*fcja sprawdzajaca czy tablica przekazana jako
parametr f-cji jest posortowana */
bool jestSort(int n, int t[] )
{
    for (int i=0; i < n-1; i++)
        if (t[i] > t[i+1])
            return false;
    return true;
}
```

Sortowanie szybkie – *quicksort* - implementacja

```
int main()
{  //1sp. sami deklarujemy testowa tablice
    cout<<"quicksort dla przykladowej tablicy\n";
    int a[]={32,65,21,48,87,5,59,74};
    printTab(8,a);//wyswietlamy tablica
    cout << " -> ";
    quicksort(0,7,a);//sortujemy
    printTab(8,a);//wyswietlamy tablica
    cout << endl;
    //sprawdzamy czy posortowana
    cout<<(jestSort(8,a)?"ok":"blad")<< endl<<endl;
/*quicksort dla przykladowej tablicy:
[32,65,21,48,87,5,59,74] -> [5,21,32,48,59,65,74,87]
ok                                  */
```

Sortowanie szybkie – *quicksort* - implementacja

```
//2sp dane pobieramy od uzytkownika
    cout<<"dane do sortowania od uzytkownika:"<<endl;
    int tab[NMAX];
    int n;
    pobTab(n,tab);  //pobieramy tablice
    printTab(n,tab);  //wydruk przed
    cout<<" -> ";
    quicksort(0,n-1,tab); //sortowanie
    printTab(n,tab);  //wydruk po
    cout<<endl;
    //spr. czy posortowana
    cout<<(jestSort(n,tab)?"ok":"blad")<< endl<<endl;
```

Sortowanie szybkie – *quicksort* - implementacja

```
//3 sposob dane generujemy losowo
    cout<<"losowo generujemy 10 tablic:"<<endl;
    // Ustawia punkt startowy
    //generatora pseudolosowego
    srand( time( NULL ) );
    int t[NMAX] = {0};    //tworzymy i od razu
                          //zerujemy tablice NMAX elementowa
    int nmax ;   //zmienna na rzeczywisty rozmiar
                 // tablicy 0<nmax<=NMAX
```

Sortowanie szybkie – *quicksort* - implementacja

```
  for (int i=0; i<10; i++)
  {   genTabLos(nmax,t) ; //generujemy tablice losowo
      printTab(nmax, t);  //wydruk przed
      quicksort(0, nmax-1, t);//Sortowanie tablicy:
      cout<<" -> ";
      printTab(nmax, t);  //wydruk po
      cout<<endl;
      if (jestSort(nmax,t)) cout<<"OK\n";
      else  cout<<"Blad\n";
  }
  return 0;
}
```

Sortowanie szybkie – *quicksort* - implementacja

- Sortowanie szybkie zostało wynalezione przez angielskiego informatyka, profesora Tony'ego Hoare'a w latach 60-tych ubiegłego wieku.
- W przypadku typowym algorytm ten jest najszybszym algorytmem sortującym z klasy złożoności obliczeniowej O(n log n) - stąd pochodzi jego popularność w zastosowaniach.
- Należy jednak pamiętać, iż w pewnych sytuacjach (zależnych od sposobu wyboru pivotu oraz niekorzystnego ułożenia danych wejściowych np. gdy dane wejściowe są już posortowane, lub gdy są uporządkowane w odwrotnej kolejności) klasa złożoności obliczeniowej tego algorytmu może się degradować do $O(n^2)$, co więcej, poziom wywołań rekurencyjnych może spowodować przepełnienie stosu i zablokowanie komputera.