Laborator 02 - Noțiuni de C++ și Sortare

Responsabili

- Octavian Rînciog [mailto:octavian.rinciog@cs.pub.ro] (2013)
- Clementin Cercel [mailto:clementin.cercel@gmail.com] (2012)

Obiective

În urma parcurgerii acestui laborator studentul va:

- înțelege conceptul de referințe din C++
- realiza supraîncarcarea operatorilor din C++
- aprofunda tehnica divide et impera în vederea implementării unor algoritmi rapizi de sortare
- putea estima complexitatea algoritmilor la un nivel de bază

Referințe

In C++ există două modalități de a lucra cu adrese de memorie:

- pointeri(la fel ca cei din C)
- referințe.

Referinţa poate fi privită ca un pointer constant inteligent, a cărui iniţializare este forţată de către compilator(la definire) şi care este dereferenţiat automat.

Semantic, referințele reprezintă aliasuri ale unor variabile existente. La crearea unei referinţe, aceasta trebuie iniţializată cu adresa unui obiect(nu cu o valoare constantă).

Sintaxa pentru declararea unei referințe este:

```
tip& referinta = valoare;
```

Exemplu:

```
    int x=1, y=2;
    int& rx = x; //referinta
    rx = 4; //modificarea variabilei prin referinta
    rx = 15; //modificarea variabilei prin referinta
    rx =y; //atribuirea are ca efect copierea continutului
           //din y in x si nu modificarea adresei referintei
```

Spre deosebire de pointeri:

- referinţele sunt iniţializate la creare (pointerii se pot iniţializa oricând)
- referinţa este legată de un singur obiect şi această legătură nu poate fi modificată pentru un alt obiect
- referințele nu au operații speciale, toți operatorii aplicați asupra referințelor sunt de fapt aplicați asupra variabilei referite(de exemplu extragerea adresei unei referințe va returna adresa variabilei referite)
- nu există referinţe nule – ele sunt totdeauna legate de locaţii de memorie

Referinţele se folosesc:

- în listele de parametri ale funcţiilor
- ca valori de întoarcere ale funcţiilor

Motivul pentru aceste tipuri de utilizări este unul destul de simplu: când se transmit parametri funcțiilor, se copiază conținutul variabilelor transmise pe stivă, lucru destul de costisitor. Prin transmiterea de referințe, nu se mai copiază nimic, așadar intrarea sau ieșirea dintr-o funcție sunt mult mai putin costisitoare.

Keyword const

În C++, există mai multe întrebuințări ale cuvântului cheie **const**:

- specifică un obiect a cărui valoare nu pot fi modificată
- specifică metodele unui obiect constant care pot fi apelate

Pentru a specifica, un obiect a cărui valoare nu poate fi modificată, **const** se poate folosi în următoarele feluri:

- `const tip variabila` ⇒ specifică o variabilă constantă
- `tip const& referinta_ct = variabilă;` ⇒ specifică o referință constantă la un obiect, obiectul neputând fi modificat
- `const int *p_int` ⇒ specifică un pointer la int care nu poate fi modificat (Variabilei `p_int` nu i se poate asigna nici o valoare, dar conținutul locației de memorie către care `p_int` arată se poate modifica)
- `int * const p_int` ⇒ specifică un pointer la int modificabil, dar conținutul locației de memorie către care `p_int` arată nu se poate modifica.

Orice obiect constant poate apela doar funcții declarate constante. O funcție constantă se declară folosind sintaxa:

```
     void fct_nu_modifica_obiect() const; //am utilizat cuvântul cheie const
                         //dupa declarația funcției fct_nu_modifica_obiect
```

Această declaratie a functiei garantează faptul că obiectul pentru care va fi apelată nu se va modifica.

Regula de bază a apelării membrilor de tip funcție ai claselor este:

- funcțiile `const` pot fi apelate pe toate obiectele
- funcțiile non-const pot fi apelate doar pe obiectele non-const.

Exemple:

```
//declarație
class Complex {
    int re;
    int im;
    int GetRe() const;
    int GetIm() const;
    void  SetRe(int re);
    void SetIm(int im);
};


//apelare
Complex c1;
const Complex c2;
c1.GetRe();   //corect
c1.SetRe(5);  //corect
c2.GetRe();   //corect
c2.SetRe(5);  //incorect
```

Funcții care returnează referințe

Pentru clasa Complex, definim funcţiile care asigură accesul la partea reală, respectiv imaginară a unui număr complex:

```
 double getRe(){ return re; }
 double getIm(){ return im; }
```

Dacă am dori modificarea părţii reale a unui număr complex printr-o atribuire de forma:

```
 z.getRe()=2.;
```

constatăm că funcţia astfel definită nu poate apărea în partea stângă a unei atribuiri.

Acest neajuns se remediază impunând funcţiei să returneze o referinţă la obiect, adică:

```
 double& getRe(){ return re; }
```

Codul de mai sus returnează o referință către membrul `re` al obiectului `Complex z`, așadar orice atribuire efectuată asupra acestui câmp va fi vizibilă și în obiect.

Clase/metode prietene

Așa cum am văzut în primul laborator, fiecare membru al clasei poate avea 3 specificatori de acces:

- public
- private
- protected

Alegerea specificatorilor se face în special în funcție de ce funcționalitate vrem să exportăm din clasa respectivă.

Dacă vrem să accesăm datele private/protejate din afara clasei, avem următoarele opțiuni:

- Funcții care ne întorc/setează valorile membre
- Funcții/Clase prietene (friend) cu clasa curentă.

O funcție prieten are următoarele proprietăți:

- O funcţie este considerată prietenă al unei clase, dacă în declararea clasei, este declarată funcţia respectivă precedată de specificatorul **friend**
- Declararea unei funcţii prieten poate fi făcută în orice parte a clasei(publică, privată sau protejată).
- Definiţia funcţiei prieten se face global, în afara clasei.
- Funcția declarața ca **friend** care acces liber la orice membru din interiorul clasei.

O clasă prieten are următoarele proprietăți:

- O clasă B este considerată prieten al unei clase A, dacă în declararea clasei A s-a întâlnit expresia: `friend class B`
- Clasa B poate accesa orice membru din clasa A, fără nici o restricție.

Exemplu:

```
class Complex{

private:
    int re;
    int im;
public:
    int GetRe();
    int GetIm();
    friend double ComplexModul(Complex c);   //am declarat fct ComplexModul ca prieten
    friend class Polinom;   //Acum clasa Polinom care acces deplin la membrii **re** și **im**
};

double ComplexModul(Complex c)
{
   return sqrt(c.re*c.re+c.im*c.im);  //are voie, intrucat e prietena
}
```

Supraîncarcarea operatorilor

Un mecanism specific C++ este supraîncarcarea operatorilor, prin care programatorul poate asocia noi semnificaţii operatorilor deja existenţi. De exemplu, dacă dorim ca două numere complexe să fie adunate, în C trebuie să scriem funcții specifice, nenaturale. În C++ putem scrie foarte ușor:

```
Complex a(2,3);
Complex b(4,5);
Complex c=a+b; //operatorul + a fost supraîncarcat pentru a aduna două numere complexe
```

Acest lucru este posibil, întrucât un operator este văzut o funcție, cu declarația:

```
   tip_rezultat operator#(listă_argumente);
```

Așadar pentru a supraîncărca un operator pentru o anumită clasă, este necesar să declarăm funcția următoare în corpul acesteia:

```

   tip_rezultat operator#(listă_argumente);
```

Există câteva restricții cu privire la supraîncarcare:

- Nu pot fi supraîncărcaţi operatorii: ::, ., .*, ?:, sizeof.
- Setul de operatori ai limbajul C++ nu poate fi extins prin asocierea de semnificaţii noi unor caractere, care nu sunt operatori, de exemplu nu putem defini operatorul === .
- Prin supraîncărcarea unui operator nu i se poate modifica aritatea (astfel operatorul ! este unar şi poate fi redefinit numai ca operator unar).
- Asociativitatea şi precedenţa operatorului se menţin.
- La supraîncărcarea unui operator nu se pot specifica argumente cu valori implicite.

Operatori supraîncărcaţi ca funcţii prieten

Un operator binar va fi reprezentat printr-o funcţie nemembră cu două argumente, iar un operator unar, printr-o funcţie nemembră cu un singur argument.

Utilizarea unui operator binar sub forma **a#b** este interpretată ca **operator#(a,b)**.

Argumentele sunt clase sau referinţe constante la clase.

Supraîncărcarea operatorilor << şi >>

În C++, orice dispozitiv de I/O este văzut drept un stream, așadar operațiile de I/O sunt operații cu stream-uri, care se definesc în felul următor:

- **Citire**: se execută cu operatorul de extracție », membru al clasei istream
- **Scriere**: se execută cu operatorul de inserție «, membru al clasei ostream

Acești operatori pot fi supraîncărcați pentru o clasă pentru a defini operații de I/O direct pe obiectele clasei.

Supraîncărcarea se poate efectua folosind funcții friend utilizând următoarea sintaxă:

```
istream& operator>> (istream& f, clasa & ob);        //Acum pot scrie in >> ob
ostream& operator<< (ostream& f, const clasa & ob);  //Acum pot scrie out << ob
```

Operatorii » și « întorc fluxul original, pentru a scrie înlănțuiri de tipul `f»ob1»ob2`.

Funcţiile operator pentru supraîncărcarea operatorilor de I/O le vom declara ca funcţii prieten al clasei care interacţionează cu fluxul.

Complex.h

```
#include <iostream>

class Complex
{
public:
    double re;
    double im;

    Complex(double real=0, double imag=0): re(real), im(imag) {};

    //supraîncărcarea  operatorilor +, - ca functii de tip "friend"
    friend Complex operator+(const Complex& s, const Complex& d);
    friend Complex operator-(const Complex& s, const Complex& d);

    //funcţii operator pentru supraîncărcarea operatorilor de intrare/ieşire
    //declarate ca funcţii de tip "friend"
    friend std::ostream& operator<< (std::ostream& out, const Complex& z);
    friend std::istream& operator>> (std::istream& is, Complex& z);
};
```

Complex.cpp

```
#include "complex.h"

Complex operator+(const Complex& s, const Complex& d){
  return Complex(s.re+d.re,s.im+d.im);
}

Complex operator-(const Complex& s, const Complex& d){
  return Complex(s.re+d.re,s.im+d.im);
}

std::ostream& operator<<(std::ostream& out, const Complex& z){
   out << "(" << z.re << "," << z.im << ")"<< std::endl;
   return out;
}

std::istream& operator>>(std::istream& is, Complex& z){
  is >> z.re >> z.im;
  return is;
}
```

main.cpp

```
#include "complex.h"

int main() {
 Complex a(1,1), b(-1,2);
 std::cout << "A: " << a << "B: " << b;
 std::cout << "A+B: " << (a+b);
 std::cin >> b;
 std::cout << "B: " << b;
 a=b;
 std::cout << "A: " << a << "B: " << b;
}
```

Operatori supraîncărcaţi ca funcţii membre

Funcţiilor membru li se transmite un argument implicit **this**(adresa obiectului curent), motiv pentru care un operator binar poate fi implementat printr-o funcţie membru nestatică cu un singur argument.

Operatorii sunt interpretați în modul următor:

- Operatorul binar **a#b** este interpretat ca **a.operator#(b)**
- Operatorul unar prefixat **#a** este interpretat ca **a.operator#()**

- Operatorul unar postfixat **a#** este interpretat ca **a.operator#(int)**

Complex.h

```
#include <iostream>

class Complex
{
public:
    double re;
    double im;

    Complex(double real, double imag): re(real), im(imag) {};

    //operatori supraîncărcaţi ca funcţii membre
    Complex operator+(const Complex& d);
    Complex operator-(const Complex& d);
    Complex& operator+=(const Complex& d);

    friend std::ostream& operator<< (std::ostream& out, const Complex& z);
    friend std::istream& operator>> (std::istream& is, Complex& z);
};
```

Complex.cpp

```
#include "complex.h"

Complex Complex::operator+(const Complex& d){
  return Complex(re+d.re, im+d.im);
}

Complex Complex::operator-(const Complex& d){
  return Complex(re-d.re, im-d.im);
}

Complex& Complex::operator+=(const Complex& d){
  re+=d.re;
  im+=d.im;
  return *this;
}

std::ostream& operator<<(std::ostream& out, const Complex& z){
   out << "(" << z.re << "," << z.im << ")"<< std::endl;
   return out;
}

std::istream& operator>>(std::istream& is, Complex& z){
  is >> z.re >> z.im;
  return is;
}
```

Supraîncărcarea operatorului de atribuire

Așa cum am amintit mai sus, majoritatea operatorilor pot fi supraîncărcați. O atenție importantă trebuie acordată operatorului de atribuire, dacă nu este supraîncărcat, realizează o copiere membru cu membru.

Pentru obiectele care nu conţin date alocate dinamic la iniţializare, atribuirea prin copiere membru cu membru funcţionează corect, motiv pentru care nu se supraîncarcă operatorul de atribuire.

Pentru clasele ce conţin date alocate dinamic, copierea membru cu membru, executată în mod implicit la atribuire conduce la copierea pointerilor la datele alocate dinamic, în loc de a copia datele.

Operatorul de atribuire poate fi redefinit numai ca funcţie membră, el fiind legat de obiectul din stânga operatorului =, motiv pentru care va întoarce o referinţă la obiect.

String.h

```
class String{
  char* s;
  int n;
```

```
  public:
        String();
        String(const char* p);
        String(const String& r);
        ~String();
        String& operator=(const String& d);
        String& operator=(const char* p);
};
```

String.cpp

```
#include "String.h"
#include <string.h>

String& String::operator=(const String& d){
  if(this != &d){    //evitare autoatribuire
    if(s)            //curatire
      delete [] s;
    n=d.n;           //copiere
    s=new char[n];
    strncpy(s, d.s, n);
  }
  return *this;      //intoarce referinta la obiectul modificat
}

String& String::operator=(const char* p){
  if(s)
      delete [] s;
  n=strlen(p);
  s=new char[n + 1];
  strncpy(s, p, n);
  return *this;
}
```

Algoritmi de sortare

O modalitate de a rezolva problemele este divizarea lor în subprobleme de dimensiune mică în speranţa că, prin divizarea repetată şi combinarea soluţiilor, complexitatea rezultată va fi mai bună decât cea initială. Metoda divide et impera presupune următoarea abordare:

- descompunerea problemei în subprobleme de dimensiuni mai mici
- rezolvarea succesivă şi independentă a fiecărei probleme rezultate
- combinarea soluţiilor în vederea furnizării rezultatului final

Timpul de execuţie al algoritmului se calculează astfel:

$$f(n) = \begin{cases} \Theta(1), n_0 \geq n \\ aT(\frac{n}{b}) + D(n) + c, n > c \end{cases}$$

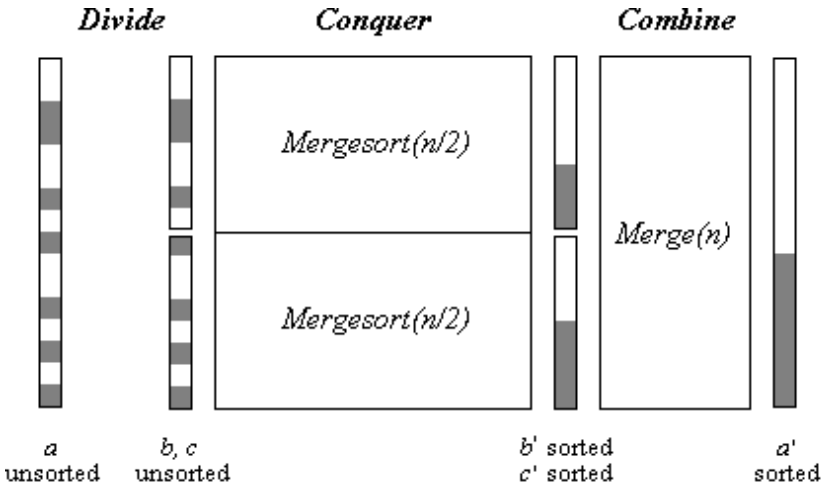
```
Pentru o dimensiune a problemei mai mică decât o valoare
n0(uzual 0, 1, 2),algoritmul se rezolvă în timp constant,în timp ce
pentru dimensiuni mai mari, complexitatea se obţine recurent.
Soluţia recurenţei se poate determina prin dezvoltarea recurenţei
până cand subproblemele devin probleme cu dimensiuni triviale sau
prin aplicarea Teoremei Master.
```

MergeSort

Algoritmul MergeSort ordonează o secvenţă de obiecte, sortând cele două jumătăţi ale sale şi interclasându-le. Idee: Algoritmul se bazează pe o strategie de tip divide et impera:

- Divide – descompune secvenţa in doua jumătăţi

- Conquer – sorteaza independent fiecare jumătate
- Combine – combină rezultatele obţinute



Pe baza noţiunilor prezentate putem descrie o funcţie care sortează o secvenţă A de la indexul p la indexul q.

Exemplu:

```
void mergesort(int p, int q)
 {
  if (p<q)
  {
   int m=(p+q)/2;
   mergesort(p, m);
   mergesort(m+1, q);
   merge(p, m, q);
  }
 }
```

Remarcăm faptul că, în primul rând se determină indexul median al secvenţei de ordonat, după care, prin apeluri recursive ale funcţiei, sunt sortate cele două subsecvenţe corespunzătoare. În final, funcţia merge realizează combinarea soluţiilor obţinute. Condiţia de oprire a recurenţei este ca lungimea secvenţei de sortat în pasul curent să fie 1 (un vector cu un singur element este întotdeauna sortat).

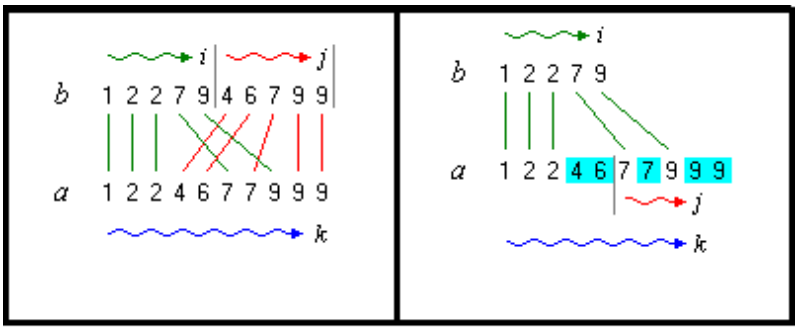
Variante de implementare a funcţiei de combinare

Varianta straightforward

- cele două jumătăţi sortate sunt copiate într-un vector auxiliar b
- parcurge vectorul b cu doi pointeri i şi j şi copiază în a la fiecare pas elementul pentru care toate elementele din setul adiacent sunt mai mari decât el (pentru sortare crescătoare).
- copiază elementele rămase

Varianta eficientă

- Observăm faptul că nu este necesar să copiem cea de-a doua jumătate a vectorului a în vectorul auxiliar b. În acest mod, cerinţele de memorie se reduc, precum şi timpul de parcurgere a elementelor vectorului. De asemenea, dacă toate elementele din prima jumătate a lui b au fost copiate înapoi în a, putem conchide că elementele rămase din cea de-a doua jumătate se află deja pe poziţiile corespunzătoare secvenţei ordonate.



a) Varianta straightforward b) Varianta eficienta

Analiza de complexitate Varianta straightforward a funcţiei merge necesita maxim 2n paşi: n paşi pentru copierea vectorului a în vectorul b şi alţi n paşi pentru copierea în sens invers. Astfel, complexitatea temporală a sortării prin interclasare este dată de recurenţa:

$T(n) = 2T(n/2) + O(n), T(1) = O(1)$

Se obţine soluţia:

$T(n) = O(n\lg(n) + n) = O(n \log(n))$

Quicksort

Algoritmul de sortare rapidă a fost inventat de Hoare şi se bazează pe aceeaşi tehnică de divide et impera ilustrată în secţiunea precedentă. Spre deosebire de Mergesort însă, partea nerecursivă a algoritmului este dedicată construirii subcazurilor şi nu combinării soluţiilor lor. Cele trei etape sunt prezentate în urmatoarea figură:

- Divide – împarte vectorul a în două subseturi b şi c cu proprietatea că toate elementele din b sunt mai mici sau egale decât toate elementele din c
- Conquer – sortează cele două subseturi
- Combine - obţine secvenţa sortată (nu este o etapa propriu-zisă deoarece secvenţa este deja sortată)

Ideea algoritmului se bazează pe alegerea unui pivot x care va reprezenta elementul de comparaţie în vederea stabilirii celor două subseturi de elemente. Astfel, toate elementele mai mici decat x vor fi plasate în primul subset, în timp ce elementele mai mari sau egale cu x îşi vor găsi locul în cel de-al doilea.

Exemplu:

```
void quicksort(int p, int q)
 {
  if (p<q)
  {
   r = partitie(p, q);
   quicksort(p, r);
   quicksort(r+1, q);
  }
 }
```

Funcţia de partiţionare are la bază următorul algoritm:

1. alege drept pivot elementul din mijlocul secvenţei (pentru performanţe superioare se poate face o alegere random)
2. repetă până când i>j
   - caută primul element a[i] mai mare sau egal decât x
   - caută ultimul element a[j] mai mic sau egal decât x
   - dacă i⇐j interschimbă a[i] si a[j] şi actualizează i şi j

Analiza de complexitate

Timpul de execuţie a algoritmului depinde de gradul de ehilibrare a partiţionării. Astfel:

- Dacă elementul pivot se alege în aşa fel încât partiţionarea să se facă în părţi egale (cazul cel mai favorabil), complexitatea este aceeaşi ca a algoritmului Mergesort (O(n log(n)).
- În cazul cel mai defavorabil, partiţionarea se face în vectori de 1, respectiv n-1 elemente, timpul de executie fiind O(n^2), rezultat mai slab decât cel furnizat de MergeSort.

Mai multe metode de implementare quicksort găsiți la [2] .

Complexitatea algoritmilor

Tabelul urmator arata cum creste timpul de executie in raport cu dimensiunea problemei pentru diferite clase de algoritmi. Complexitatea unui algoritm este echivalenta cu rata de crestere a timpului de executie in raport cu dimensiunea problemei.

n	O(log(n))	O(n)	O(n*log(n))	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	3.322	10	33	100	1000	1024
20	4.322	20	86	400	8000	1048576
30	4.907	30	147	900	27000	1073741824
40	5.322	40	213	1600	64000	1.09951E+12
50	5.644	50	282	2500	125000	1.1259E+15
100	6.644	100	664	10000	1000000	1.26765E+30
1000	9.966	1000	9966	1000000	1000000000	1.0715E+301
10000	13.288	10000	132877	100000000	1E+12	1.0E+3001

Exerciții

- Fiind date N numere complexe cu coordonate double, sortați-le în funcție de distanța euclidiană față de centrul sistemului de coordonate cartezian. **Observație** Pentru rezolvarea laboratorului, plecați de la arhiva lab02-tasks.zip

Pentru a putea rezolva exercițiul, aveți în vedere următoarele detalii de implementare:

- (3.5p) Implementați clasa **Complex**, cu următoarele particularități:
  - Vor exista doi constructori:
    - primul, vid, va inițializa coordonatele la 0 (0.25p) (TODO 1.1)
    - al doilea va primi ca argumente coordonatele. (0.25p) (TODO 1.2)
  - Se vor implementa funcţii membre:
    - determinarea părților reale și imaginare (0.5p) (TODO 1.3)
    - supraîncărcarea operatorului de comparație <, conform criteriului de mai sus (0.5p) (TODO 1.4)
    - supraîncărcarea operatorului de comparație >, conform criteriului de mai sus (0.5p) (TODO 1.5)
    - supraîncărcarea operatorului de comparație ==, conform criteriului de mai sus(0.5p) (TODO 1.6)
    - supraîncărcarea operatorilor +, - pentru a permite operaţii cu două argumente numere complexe (0.5p) (TODO 1.7)
  - Se vor implementa funcţii friend(nemembre) pentru:
    - supraîncărcarea operatorului « (scrierea unui număr complex într-un stream) (0.25p) (TODO 1.8)
    - supraîncărcarea operatorului » (citirea unui număr complex dintr-un stream) (0.25p) (TODO 1.9)

- (6.5p) Implementaţi clasa template **Vector** care să permită lucrul cu vectori de obiecte, cu următoarele particularități:
  - Vor exista doi constructori:
    - primul, vid, va inițializa numărul de elemente la 0 și pointerul de elemente la NULL (0.5p) (TODO 2.1)
    - al doilea va primi ca argument numărul de elemente și va aloca memorie pentru pointer (0.5p) (TODO 2.2)
  - Se va defini și un destructor, care va dezaloca memoria alocată dinamic. (1p) (TODO 2.3)
  - Se vor implementa funcţii membre pentru:
    - determinare dimensiune vector (0.5p) (TODO 2.4)
    - supraîncărcarea operatorului de atribuire între două obiecte de tip vector (1p) (TODO 2.5)
    - supraîncărcarea operatorului de indexare [] ce va permite accesul la elementele individuale prin indexare. **Operatorul de indexare** este un operator binar, având ca prim termen obiectul care se indexează, iar ca al doilea termen indicele. (*`obiect[indice]` este interpretat ca `obiect.operator[](indice)`* (1p) (TODO 2.6)
  - Se vor implementa funcţii friend (nemembre) pentru:
    - testul de egalitate a doi vector (supraîncărcarea operatorului ==) (1p) (TODO 2.7)
    - supraîncărcarea operatorului « (1p) (TODO 2.8)
    - supraîncărcarea operatorului » (0.5p) (TODO 2.9)

- (3p) Implementați funcția **sort**, membră a clasei Vector, în care folositi algoritmul Mergesort sau Quicksort pentru sortarea vectorului de numere complexe în funcție de distanța euclidiană față de centrul sistemului de coordonate cartezian. (TODO 3)

Interviu

Această secțiune nu este punctată și încearcă să vă facă o oarecare idee a tipurilor de întrebări pe care le puteți întâlni la un job interview (internship, part-time, full-time, etc.) din materia prezentată în cadrul laboratorului.

1. Dându-se o listă simplu înlănțuită, implementați un algoritm de sortare și argumentați alegerea acelui algoritm.
2. Puteți sorta un vector de întregi în O(n)? Descrieți algoritmul.
3. Dacă ați avea un milion de întregi cum i-ați sorta și câtă memorie s-ar consuma? Dacă avem un fișier de câțiva GB având un string pe linie, descrieți cum ați sorta liniile din acest fișier.
4. Scrieți o metodă de sortare a unui array de string-uri în așa fel încât toate anagramele unui cuvânt să fie unele după altele.
5. Dându-se o matrice a căror linii (de la stânga la dreapta) și coloane(de sus în jos) sunt sortate crescător, implementați o metodă care găsește un element dat.
6. Se dă un vector aproape sortat, în care fiecare element nu este deplasat cu mai mult de k poziții față de poziția în care ar fi, în cazul în care vectorul este sortat. Găsiți un algoritm eficient pentru a sorta vectorul și determinați complexitatea acestuia.

Și multe altele…

Bibliografie

[1] C++ Reference [http://www.cplusplus.com]

[2] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm [http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/quicken.htm]

[3] MergeSort - Wikipedia [http://en.wikipedia.org/wiki/Merge_sort]

[4] QuickSort - Wikipedia [http://en.wikipedia.org/wiki/Quicksort]

sd-ca/laboratoare/laborator-02.txt · Last modified: 2013/03/04 10:17 by sorina.sandu