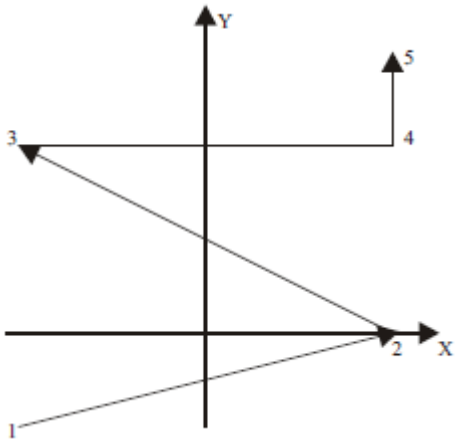
Priprema pred kolokvijum

1. Gradonačelnik jednog grada doneo je obavezujuću odluku da se na dan održavanja Rolerijade vozačima koji dolaze sa magistralnog pravca na svakoj raskrsnici zabrani skretanje u levo uveren da bi takvim skretanjima mogli naići na učesnike rolerijade. Za svako takvo skretanje saobraćajci će naplaćivati kazne. Ako su za jedno vozilo date koordinate za N raskrsnica koje je vozač prošao (u redosledu prolaska), odrediti koliko puta je platio kaznu.
Na primer, ako je na mršuti vozača bilo N=5 raskrsnica sa koordinatama: 1(-2, -1), 2(2, 0), 3(-2, 2), 4(2, 2), 5(2, 3) napravljena su dva skretanja u levo i prema tome plaćene dve kazne

ULAZ	IZLAZ
5	2
-2 -1	
2 0	
-2 2	
2 2	
2 3	



Resenje:
Trajektorija kretanja automobila se moze predstaviti kao izlomljena linija, koja se sastoji od usmerenih odsecaka (vektora).
Skretanje je u levo ako pravac tekuceg odsecka trajektorije a2 u odnosu na pravac prethodnog odsecka a1 po manjem uglu predstavlja kretanje u levo (suprotno kretanju kazaljke na satu). A to znaci, da je vektorski proizvod axb>0

```
int Kazne(int n, Point p[])
{int s=0;
for (int i=0;i<=n-3;i++)
s+=VectMul(Vector(p[i],p[i+1]),Vector(p[i+1],p[i+2]))>0;
return s;
}
```

2. Sortirajte n celih brojeva koji su iz intervala -5 do 5. Broj n (n<10^8) se unosi sa standardnog ulaza kao i niz od n celih brojeva.

3. Unosi se n1 (broj kapetana u avionu), n2 (broj prvih oficira) i m opisa veza koje postoje samo između kapetana i prvih oficira. Veze su opisane sa po 2 čvora a, i b (0<=a<n1, 0<=b<n2). Čvor a iz skupa kapetana je povezan sa čvorom b iz skupa prvih oficira ako se njihove efikasnosti sličnog intenziteta. U prvom redu standardnog ulaza dati su brojevi n1, n2 i m (broj parova (a,b)). Potom su dati opisi grana u dva reda sa po m članova razdvojenih blanko karakterom gde svaka dva clana reda predstavljaju jedan par (a,b). Ispisi koliko najviše parova posada (kapetan, prvi oficir) se može formirati tako da je svaki kapetan povezan sa najviše jednim prvim oficirom.

ULAZ
6 5 10

0 0 0 1 0 3 1 1 1 4
2 2 3 0 4 2 4 4 5 4

IZLAZ
5

Podesećanje:

1.
Graham scan algoritam

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX 100
struct Point {int x,y;};
struct Point P[MAX], Q[MAX];
int n,k;

void form(Point P[], int n) // tacke zadaje na slucajan nacin
{
for (int i=0; i<n; i++)
{
   P[i].x=(int)((float)rand()/RAND_MAX *10+0.5);
   P[i].y=(int)((float)rand()/RAND_MAX *10+0.5);
}
}

void print(Point P[], int n)
{
for (int i=0; i<n; i++) printf("(%d,%d) ",P[i].x,P[i].y);
}

// vraca teme kome je X najmanje, i u slucaju vise takvih max Y
int LeftP(Point P[], int n)
{
int Ind=0;
for (int i=1; i<n; i++)
   if ((P[i].x<P[Ind].x) || (P[i].x==P[Ind].x && P[i].y>P[Ind].y))   Ind=i;
return Ind;
}

// vraca cosinus ugla izmedju vektora kojima su krajevi tacke P i Q
double cosVekt(Point P, Point Q)
{ return (P.x*Q.x+P.y*Q.y)/(sqrt(P.x*P.x+P.y*P.y)*sqrt(Q.x*Q.x+Q.y*Q.y)); }

// vraca vektor kome je pocetak tacka a, a kraj tacka b
Point Vector(Point a, Point b)
{
Point pom;
  pom.x=b.x-a.x;
  pom.y=b.y-a.y;
return pom;
}

double Moduo(Point a)// vraca moduo vektora a
{ return sqrt(a.x*a.x+a.y*a.y); }
```

```
// vraca teme koje je kraj vektora sa pocetkom u tekT koji zaklapa
// najmanji ugao sa pravcem vektora lastV, ako je vise takvih temena uzima
// ono ciji je vektor sa najvecim modulom
int minUgao(Point P[], int n, int tekT, Point lastV)
{
int NextT=(tekT+1)%n;
for (int i=0; i <n; i++)
   if (i!=tekT && i!=NextT)
       if (cosVekt(lastV,Vector(P[tekT],P[i]))>cosVekt(lastV,Vector(P[tekT],P[NextT]))
||(cosVekt(lastV,Vector(P[tekT],P[i]))==cosVekt(lastV,Vector(P[tekT],P[NextT])) &&
Moduo(Vector(P[tekT],P[i]))>Moduo(Vector(P[tekT],P[NextT]))))
NextT=i;

return NextT;
}

// iz niza P[] u niz Q[] kopira temena koja obrazuju konveksni omotac
void Convex(Point P[], int n, Point Q[], int *k)
{
 int startT=LeftP(P,n), tekT, NextT;
 struct Point lastV={0,1};
 tekT=startT;
 Q[0]=P[tekT];
 *k=0;

do
{
  NextT=minUgao(P,n,tekT,lastV);
  Q[++(*k)]=P[NextT];
  lastV=Vector(P[tekT],P[NextT]);
  tekT=NextT;
}
while (tekT!=startT);
}

int main()
{
   scanf("%d",&n);
   form(P,n);
   print(P,n);
   Convex(P,n,Q,&k);
   printf("\n");

 for (int i=0; i<k; i++)
    printf("(%d,%d) ", Q[i].x,Q[i].y);

 return 0;
}
```

2. DFS

/* DFS implementacija u programskom jeziku C za graf dat matricom povezanosti */ **#include <stdio.h>** **#define MAX 10**	// DFS, ali predstavljanja grafa preko niza listi //suseda svakog od čvorova grafa. #include <stdlib.h> #include <stdio.h>

```
/*maksimalan broj cvorova*/

void poseti(int x, int a[][MAX], int n, int m[]);
/*obilazak grafa sa n cvorova zadatog matricom a
pocev od neposecenog cvora x;
pri obilasku u nizu markiranih cvorova m, vrednost
clana m[x] ce postati 1, kad se poseti cvor x*/

void DFS(int a[][MAX], int n); /*DFS obilazak grafa
G sa n cvorova zadatog matricom povezanosti a*/

void ucitaj_matricu(int a[][MAX], int n); //ucitavanje
grana grafa sa stdin

int main()

{int a[MAX][MAX];
int n; //broj cvorova grafa

printf("\nUnesite broj cvorova grafa: ");
scanf("%d", &n);

ucitaj_matricu(a,n);
DFS(a, n); return 0;
}

void poseti(int x, int a[][MAX], int n, int m[])
{
int y; //cvor koji je u grafu potencijalni sused cvora x

printf(" %d ", x); //stampa se neposeceni cvor od kog
//krece nova poseta DFSom

m[x]=1; //markira se cvor x kao posecen

/*ako postoji susedni cvor y koji nije
markiran(m[y]=0), rekurzivno se poziva poseti za y
*/

for (y=0; y<n; y++)
  if( m[y]==0 && a[x][y]==1) poseti(y, a, n, m);
}

void DFS(int a[][MAX], int n)
{
int x, m[MAX]; //cvor grafa x, niz markiranih
//(posecenih) cvorova m

for (x=0; x<n; x++) m[x]=0; //na pocetku su svi
//cvorovi neposeceni

for (x=0; x<n; x++)
```

```
/* Cvor liste suseda */
typedef struct _cvor_liste
{ int broj; /* indeks suseda */
struct _cvor_liste* sledeci;
} cvor_liste;

/* Graf predstavlja niz pokazivaca na pocetke listi
suseda */
#define MAX_BROJ_CVOROVA 100

cvor_liste* graf[MAX_BROJ_CVOROVA];

int broj_cvorova;

int posecen[MAX_BROJ_CVOROVA]; /* niz
markiranih čvorova u DFS algoritmu */

/* Ubacivanje na pocetak liste */
cvor_liste* ubaci_u_listu(cvor_liste* lista, int broj)
{ cvor_liste* novi=malloc(sizeof(cvor_liste));
novi->broj=broj;
novi->sledeci=lista;
return novi;
}

void obrisi_listu(cvor_liste* lista)
{ if (lista) { obrisi_listu(lista->sledeci); free(lista);
}
}

void ispisi_listu(cvor_liste* lista)
{ if (lista) { printf("%d ",lista->broj);
ispisi_listu(lista->sledeci); }
}

/* Rekurzivna implementacija DFS algoritma */
void poseti(int i)
{ cvor_liste* sused;
printf("Posecujem cvor %d\n",i);
posecen[i]=1;
for( sused=graf[i]; sused!=NULL;
 sused=sused->sledeci)
   if (!posecen[sused->broj]) poseti(sused->broj);
}

int main()
{ int i; /* brojac u ciklusu */

printf("Unesi broj cvorova grafa : ");
scanf("%d",&broj_cvorova);

for (i=0; i<broj_cvorova; i++)
{ int br_suseda,j;
graf[i]=NULL;
printf("Koliko cvor %d ima suseda : ",i);
```

```
  if (m[x]==0) poseti(x, a, n, m); //ako x nije posecen,
//pokrenuti posetu iz x
}

void ucitaj_matricu(int a[][MAX], int n)
{
int i, j;
for (i=0; i< n; i++)
   for(j=0; j<n; j++) a[i][j]=0;
printf("\nUnesite grane grafa u obliku x,y. Zavrsite
sa EOF\n");

while (scanf("%d,%d", &i, &j) != EOF) a[i][j]=1;
}
```

```
scanf("%d",&br_suseda);

for (j=0; j<br_suseda; j++)
{ int sused;
do{
printf("Unesi broj %d. suseda cvora %d : ",j+1,i);
scanf("%d",&sused);
} while (sused<0 || sused>=broj_cvorova);

graf[i]=ubaci_u_listu(graf[i],sused);
}
}

for (i=0; i<broj_cvorova; i++)
{ printf("%d - ",i); ispisi_listu(graf[i]);
printf("\n");}

poseti(0); return 0;
}
```

3. BFS

```
/* BFS algoritam za obilazak grafa G=(V,E) koji je zadat matricom povezanosti (a[i][j]=1, ako postoji grana
(i,j), inace a[i][j]=0) */

void BFS(int a[][50], int n)
/*a = matrica susedstva, n=broj cvorova grafa G */
{
int x; /* cvor grafa */
int m[50]; /* m= niz markiranih cvorova garaf, vrednost clana m[x] ce postati 1, kad se poseti cvor x, a inace 0 */


/* inicijalizacija niza markiranih cvorova na 0, jer su na pocetku svi cvorovi grafa neposeceni */
for (x=0; x<n; x++) m[x]=0;

/* poseta grafa pocev od prvog neposecenog cvora */
for (x=0 ;x<n ;x++ )
if ( !m[x]) poseti (x, a, n, m)

}

void poseti(int s, int a[][50], int n, int m[])
/*s = polazni cvor pretrage po sirini, a = matrica susedstva, n = broj cvorova grafa, m = niz posecenih cvorova*/
{
int i,k; /*brojaci u ciklusu /
int v; /* cvor grafa */
int red [50]; /* niz cvorova grafa poredjanih u poretku BFS pretrage */

/*inicijalizacije niza m, red u odnosu na polazni cvor pretrage s*/
m[s]=1; red[0]=s; k=1;

/*obilazak neposrednih suseda cvora s, razlika od DFSa*/
for(i=0;i<k;i++)
{
s=red[i];
for(v=0;v<n;v++)
```

```
if( !m[v] && a[s][v])
{m[v]=1; red[k++]=v; }
}

/*ispis BFS poretka*/
for(i=0;i<k;i++) printf(" %d ", red[i]);
}
```

4. Ford Fulkerson algoritam
Za dati orijentisani graf sa datim čvorom s=0 (izvor) i t=n-1 (ponor) i pridruženim kapacitetima za grane, odrediti optimalni tok od izvora do ponora. Za svaku granu, tok ne sme da bude veći od kapaciteta grane. Za svaki čvor, ulazni tok mora da bude jednak izlaznom toku. U prvoj liniji standardnog ulaza se zadaje broj cvorova n (n<=1000), broj grana e. U narednih e linija se zadaju: polazni cvor grane, dolazni cvor grane, kapaciter. Svi brojcani podaci su razdvojeni blanko karakterom. Napisati C program koji ce na standardni izlaz ispisati optimalni protok kroz dati graf.

ULAZ (6 cvorova, 10 grana, kapacitet grane (0,1) je 16)

```
6 10
0 1 16
0 2 13
1 2 10
2 1 4
3 2 9
1 3 12
2 4 14
4 3 7
3 5 20
4 5 4
```

IZLAZ (optimalni protok od cvora 0 do cvora 5)
23

```
#include <stdio.h>

#define BELI 0
#define SIVI 1
#define CRNI 2
#define MAX_CVOROVA 1000
#define MAX_TOK 1000000000

int n;  // broj čvorova
int e;  // broj grana
int kapacitet[MAX_CVOROVA][MAX_CVOROVA]; // matrica kapaciteta
int tok[MAX_CVOROVA][MAX_CVOROVA];     // matrica toka
int boja[MAX_CVOROVA]; // niz koji se koristi za BFS obilazak grafa
int pred[MAX_CVOROVA];  // niz koji cuva povecavajuci put

int min (int x, int y) {
    return x<y ? x : y;
}
//Implementacija reda potrebnog za BFS
int glava,rep;
int red[MAX_CVOROVA+2];
void stavi (int x) {
    red[rep] = x;
    rep++;
    boja[x] = SIVI;
}
int skini () {
    int x = red[glava];
    glava++;
```

```
    boja[x] = CRNI;
    return x;
}
//BFS za povecavajuci put
int bfs (int pocetak, int cilj) {
    int u,v;
    for (u=0; u<n; u++) boja[u] = BELI;

    glava = rep = 0;
    stavi(pocetak);
    pred[pocetak] = -1;
    while (glava!=rep) {
        u = skini();
        // pretraziti sve susedne bele cvorove v.
        // Ako je kapacitet grane (u,v) pozitivan u rezidualnom grafu,
        // smestiti v u red.
        for (v=0; v<n; v++) {
            if (boja[v]==BELI && kapacitet[u][v]-tok[u][v]>0) {
                stavi(v);
                pred[v] = u;
            }
        }
    }
    // Ako je trenutno boja dolaznog cvora crna,
    // onda je taj cvor posecen
    return boja[cilj]==CRNI;
}
//Ford-Fulkerson algoritam
int max_flow (int izvor, int ponor) {
    int i,j,u;
    // Inicijalizacija: prazan tok
    int max_flow = 0;
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++) {
            tok[i][j] = 0;
        }
    }
    // Dok postoji povecavajuci put,
    // uvecavati tok duz tog puta
    while (bfs(izvor,ponor)) {
        // oderediti kolicinu za koju treba uvecati tok.
        int uvecanje = MAX_TOK;
        for (u=n-1; pred[u]>=0; u=pred[u]) {
            uvecanje = min(uvecanje,kapacitet[pred[u]][u]-tok[pred[u]][u]);
        }
        // Povecati tok.
        for (u=n-1; pred[u]>=0; u=pred[u]) {
            tok[pred[u]][u] += uvecanje;
            tok[u][pred[u]] -= uvecanje;
        }
        max_flow += uvecanje;
    }
    // Kraj, jer nema vise povecavajuceg puta
    return max_flow;
}
//citanje ulazne datoteke
void ucitavanje() {
    int a,b,c,i;

    // ucitavanje broja cvorova i broja grana
    scanf("%d %d",&n,&e);

    // ucitavanje kapaciteta grana
    for (i=0; i<e; i++) {
        scanf("%d %d %d",&a,&b,&c);
        kapacitet[a][b] = c;
    }

}
```

```
int main () {
    ucitavanje();
    printf("%d\n",max_flow(0,n-1));
    return 0;
}
```

5. STL podsecanje

```
#include <iostream>

#include <string>

#include <vector>

#include <deque>

#include <queue>

#include <stack>

#include <list>


using namespace std;


int main()
{

  // Klasa string (zaglavlje <string>) ------------------------------

  // Konstruktori:

  // string s = "Hello";

  // string s1; // Prazan string

  // string s2 = s; // Copy konstrukcija

  // Operator dodele:

  // s = "Hello C++";

  // s = s1;

  // Aritmeticki i relacioni operatori:

  // s + s1 (konkatenacija)

  // s += s1 (dopisivanje na s)

  // s + " world" (konkatenacija sa char *)

  // s == s1, s != s1 (poredjenje na jednakost i razlicitost)

  // s < s1, s <= s1, s > s1, s >= s1 (leksikografsko poredjenje)

  // s.size() (trenutna duzina stringa).

  // s.c_str() (vraca char * -- adresu niza karaktera koji sadrzi

  //                             string)

  // cin >> s; cout << s; (operatori ulaza i izlaza)

  // s[i] (indeksni pristup karakterima)

  // podrzava iteratore, resize(), capacity() i  sl. kao i u vektoru.
```

```
string str;

// vector<T> (zaglavlje <vector>) -------------------------------------

// Konstruktori:
// vector<int> v; (prazan vektor, size() == 0)
// vector<int> v1 = v; (copy konstrukcija)
// vector<int> v2(n); (vektor duzine n, svi elementi su podrazumevane
//                     vrednosti, u slucaju int-a -- nule)
// vector<int> v3(n, x) (vektor duzine n, svi elementi su inicijalizovani
//                       vrednoscu x)
// v2 = v1; -- operator dodele, kopira sadrzaj vektora v1 u v2 (nakon
//             dodele vazi v1 == v2)
// v1 == v2, v1 != v2 (poredjenje na jednakost i razlicitost)
// v1 < v2, v1 > v2, v1 <= v2, v1 >= v2 (leksikografsko poredjenje)
// v.size()  --  velicina
// v.capacity() -- kapacitet ( >= size())
// v.resize(n) -- menja velicinu (uz odsecanje elemenata na
//                kraju ako je n < v.size(), ili dopisivanje podrazumevanih
//                vrednosti za dati tip u slucaju da je n > v.size())
// v.reserve(n) -- menja kapacitet (najcesce nema potrebe za tim)
// v.push_back(x) -- dodaje na kraj (uz eventualnu realokaciju)
// v.pop_back() -- skida poslednji element
// v.back() -- referenca na poslednji element
// v.front() -- referenca na prvi element
// v[i]      -- indeksni pristup (indeksi idu od 0 do size() - 1)
// v.clear()  -- brise sadrzaj vektora (prazni ga i vraca size() na nulu)
// indeksni pristup ne vrsi automatsku realokaciju!! Ukoliko je indeks
// i >= v.size(), tada je v[i] neispravna upotreba vektora. Potrebno je
// najpre resize() funkcijom uvecati velicinu niza (npr. v.resize(i + 1),
// kako  bi u vektoru v postojao element sa indeksom i. Sa druge strane,
// push_back() automatski uvecava size() za jedan, vrseci realokaciju
// ako je potrebno.

vector<int> v;

cout << "size(): " << v.size() << endl;
cout << "capacity(): " << v.capacity() << endl;
```

```
for(int i = 0; i < 50; i++)
  {
    v.push_back(i);
    cout << "size(): " << v.size() << endl;
    cout << "capacity(): " << v.capacity() << endl;
    cout << "back(): " << v.back() << endl;
  }

// Iteracija pomocu indeksa

for(int i = 0; i < v.size(); i++)
  {
    cout << v[i] << endl;
  }


// Iteracija pomocu iteratora

for(vector<int>::iterator it = v.begin() ; it != v.end(); it++)
  {
    cout  << *it << endl;
  }


v.clear();
cout << "size(): " << v.size() << endl;
cout << "capacity(): " << v.capacity() << endl;


// deque<T> -- dek (vektor koji se moze efikasno prosirivati sa oba kraja,

//              zaglavlje <deque>)


// Osim push_back() i pop_back(), dek sadrzi i push_front() i pop_front(),

// koji omogucavaju efikasno umetanje elementa na pocetak deka. U svemu

// ostalom se dek ponasa kao i vektor, ima efikasni indeksni pristup

// elementima. Dodavanje u sredinu je, kao i kod vektora, neefikasno!!


// Adapterske klase queue i stack: ove dve klase interno koriste neki od

// sekvencijalnih kontejnera (deque ili list, u slucaju steka moze i vector)

// ali korisniku nude jednostavniji i prirodniji interfejs.


// zaglavlje <queue> ----------------------------------------------

// queue<T> -- red (podrazumevano implementiran kao deque<T>)

// queue<T, list<T> >  -- red implementiran pomocu liste

// Konstruktor:

// queue<int> q; // prazan red

// q.push(x) // dodaje  x na kraj reda

// q.pop() // skida sa pocetka reda

// q.front() // vraca element sa pocetka reda

// q.back() // vraca element sa kraja reda
```

```
queue<int> q;

// zaglavlje <stack> ----------------------------------------

// stack<T> -- stek (podrazumevano implementiran kao deque<T>)

// stack<T, vector<T> > -- stek implementiran pomocu vektora

// Konstruktor:

// stack<int> s;

// s.push(x); // postavlja x na stek

// s.pop();  // skida element sa vrha steka

// s.top(); // vraca element trenutno na vrhu steka bez skidanja

//

stack<int> s;

// list<T> (zaglavlje <list>, dvostruko povezana lista)

// Omogucava efikasno umetanje i brisanje bilo gde u kontejneru

// Ne omugucava efikasno pristupanje proizvoljnom elementu, ne

// poseduje indeksni operator []. Elementima se mora pristupati

// redom.

// Konstruktori:

// list<int> l;  (prazna lista)

// list<int> l(n); (lista od n int-ova inicijalizovanih podrazumevanim

//                  vrednostima -- nulama)

// list<int> l(n, x); (lista od n int-ova inicijalizovanih vrednoscu x)

// l.size(); // velicina liste

// l.push_back(), l.pop_back(), l.push_front(), l.pop_front(), l.back(),

// l.front() -- isto znacenje kao i kod vektora i deka

// NAPOMENA: Jedini nacin prolaska kroz listu je pomocu iteratora, buduci

// da ne postoji indeksni pristup!!

list<int> lista;

lista.push_back(1);
lista.push_back(2);
lista.push_back(8);

list<int>::iterator it = lista.begin();
it++;
it++;

lista.insert(it, 5);
```

```
for(list<int>::iterator iter = lista.begin(); iter != lista.end();
    iter++)
  {
    cout << *iter << endl;
  }

// Iteratori:

// Svaka kontejnerska klasa ima ugnjezdeni tip iterator koji predstavlja

// klasni tip koji simulira "pametni pokazivac". Svaki iterator je interno

// implementiran na nacin koji odgovara kontejneru u kome je definisan,

// a spolja se svi iteratori ponasaju isto, tj. imaju isti interfejs, nalik

// pokazivackoj sintaksi:

// list<int>::iterator i1 (iterator koji pokazuje na element liste int-ova)

// vector<double>::iterator i2  (iterator koji pokazuje na element vektora

//                               double-ova).

// deque< vector<int> >::iterator i3 (iterator koji pokazuje na element deka

//                                    ciji su elementi vektori intova)

// Iteratori podrzavaju sledece operacije:

// *it -- vraca referencu na element na koji trenutno pokazuje

// it++ (++it) -- pomera se na sledeci element u kontejneru

// it-- (--it) -- pomera se na prethodni element u kontejneru

// it1 == it2 -- da li iteratori pokazuju na isti element?

// it1 != it2 -- da li iteratori pokazuju na razlicite elemente?

// Iteratori u kontejnerima sa indeksnim pristupom (vector, deque)

// podrzavaju i sledece dodatne operacije:

// it + n (iterator koji pokazuje na element n pozicija udesno u odnosu

//         na element na koji pokazuje it)

// it - n (slicno, samo ulevo)

// it += n, it -= n (kombinacija prethodna dva sa dodelom)

// it1 - it2 (ako it1 i it2 pokazuju na elemente u istom kontejneru,

//            tada je rezultat rastojanje izmedju elemenata na koje pokazuju)

// it1 < it2, it1 > it2, it1 <= it2, it1 >= it2 (prvi slucaj znaci da it1

// pokazuje na element u kontejneru pre elementa na koji pokazuje it2, ostalo

// analogno).

// it[n]  (ekvivalentno sa *(it + n))

// NAPOMENA: Svi kontejneri sadrze funkcije clanice begin() i end() koje
```

```
// vracaju iterator na prvi element u kolekciji i iterator na poziciju
// NEPOSREDNO IZA POSLEDNJEG ELEMENTA u kolekciji (prva invalidna pozicija)
// VAZNO: end() NE VRACA ITERATOR NA POSLEDNJI ELEMENT, u tu svrhu se morate
// vratiti nazad za jedno mesto (it--).

// Iteratori nude uniforman nacin za prolazak kroz kontejner, bez obzira
// na tip kontejnera!!

// Umetanje i brisanje pomocu iteratora:
// Iteratori omogucavaju da se u svim tipovima kontejnera vrsi umetanje
// i brisanje elemenata na proizvoljnoj poziciji. Zato svi kontejneri
// podrzavaju funkcije insert() i erase():
// c.insert(it, x) // umece u kontejner c element x neposredno ispred
//                    elementa na koji pokazuje it.
// Primetimo da je c.push_back(x) <=> c.insert(c.end(), x), kao i
//                 c.push_front(x) <=> c.insert(c.begin(), x).
// c.insert(it, n, x) // umece n kopija vrednosti x neposredno ispred
//                       elementa na koji pokazuje it
// c.insert(it, it_beg, it_end) // umece elemente nekog drugog kontejnera
//                                 u opsegu [it_beg, it_end) u kontejner
//                                 c neposredno pre elementa na koji pokazuje
//                                 it.
// c.erase(it)  // brise element kontejnera c na koji pokazuje it
// c.erase(it1, it2) // brise sve elemente kontejnera c u opsegu [it1, it2)

// VAZNA NAPOMENA: Iako svi kontejneri podrzavaju navedene insert() i erase()
// operacije, ove operacije se efikasno izvrsavaju na proizvoljnoj poziciji
// samo za liste. Kod vektora su operacije insert() i erase() efikasne samo
// u slucaju iteratora koji vraca end(), tj. brisanje i dodavanje na kraj.
// Kod deka su operacije insert() i erase() efikasne samo u slucaju
// iteratora koji vraca begin() ili end(), tj. brisanje i dodavanje na pocetak
// ili kraj. U ostalim situacijama se vrsi fizicko pomeranje elemenata koji
// slede nakon pozicije umetanja, sto je neefikasno u opstem slucaju.

// Postavljamo iterator it da pokazuje na treci element u listi.

it = lista.begin();
it++;
it++;
```

```
// Neposredno pre treceg elementa ubacujemo u listu prva tri elementa
// vektora v
lista.insert(it, v.begin(), v.begin() + 3);
for(list<int>::iterator iter = lista.begin(); iter != lista.end();
    iter++)
  {
    cout << *iter << endl;
  }

// JOS O ITERATORIMA (Ovo nije bilo na casu):
// Pored klase iterator, svaki kontejner definise i klasu const_iterator.
// konstantni iteratori funkcionisu na identican nacin kao i obicni
// iteratori, jedino sto prilikom primene operatora dereferenciranja
// (operator*) vracaju referencu na konstantan element kontejnera. Time
// se sprecava modifikacija elementa na koji iterator pokazuje.
// Funkcije begin() i end() u kontejnerima vracaju iterator akko je kontejner
// nekonstantan, u suprotnom vracaju const_iterator. Dozvoljeno je
// konvertovati iterator u const_iterator, ali ne i obrnuto. To je u skladu
// sa politikom jezika o kvalifikacionim konverzijama.

for(list<int>::const_iterator iter = lista.begin(); iter != lista.end();
    iter++)
  {
    // iter je const_iterator. lista je nekonstantan objekat klase list<int>
    // zbog cega lista.begin() vraca iterator. Konverzija iteratora u
    // const_iterator je dozvoljena, tako da prevodilac ne prijavljuje
    // gresku. U obrnutoj situaciji bi se greska javila!!

    // *iter = 5; // POGRESNO!! iter je const_iterator, pa je *iter
    //               tipa const int, zato nije moguce dodeliti mu vrednost.

    cout << *iter  << endl; // Ovo je u redu!

  }
// Konstrukicja pomocu iteratora:
// Dozvoljeno je inicijalizovati jedan kontejner sekvencom elemenata
// drugog kontejnera ogranicenom parom iteratora:
vector<int> vec(lista.begin(), lista.end()); // Konstruise vektor
// koji se sastoji iz istih elemenata kao i lista.

for(vector<int>::const_iterator iter = vec.begin(); iter != vec.end();
```

```
      ++iter)
    {
      cout << *iter << endl;
    }

  return 0;
}

// Slede neke funkcije koje opisuju koriscenje kontejnera kao argumenata

// funkcija.


// Prenosenje vektora funkciji

void f1(const vector<int> & v)
{
  // Ako necemo da menjamo vektor

}

void f2(vector<int> & v)
{
  // Ako zelimo da menjamo vektor

}

void f3(vector<int> v)
{
  // Kreira se lokalna kopija (copy

  // konstrukcijom). Retko potrebno,

  // veoma neefikasno.

}

// Vracanje vektora iz funkcije

vector<int>  g1()
{
  vector<int> v;

  // Kod koji puni vektor vrednostima...


  return v; // Vraca vektor copy konstrukcijom

}

vector <int> & g2()
{
  vector<int> v;

  // ...


  return v; // Pogresno!! Vraca referencu na

            // objekat koji ce biti unisten

            // nakon zavrsetka funkcije

}



class X {
```

```
private:

  vector<double> _v;


public:

  const vector<double> & getV() const
  {
    return _v; // U redu, vraca referencu na

               // objekat koji je clan klase

               // i samim tim nastavlja da

               // zivi i nakon zavrsetka

               // funkcije.

  }

};
```