C++ - Bibliotečka podrška za algoritme i strukture podataka

STL – brzi uvod

Dodatni resursi
http://www.cplusplus.com/reference/stl/
https://en.cppreference.com/w/cpp/container
http://www.sgi.com/tech/stl/

VEKTOR

Vektor je osnovni kontejner iz STL-a i zamenjuje obican niz. Kad mu je potreban dodatni prostor u memoriji, vektor se sam povecava, tako sto se ceo vektor prekopira na dvostruko veci prostor. Buduci da se vektor svaki put povecava za dvostruko, ukupno se povecava log n puta. Time je svako novo kopiranje veće od svih ranije kreiranih memorijskih polja vektora, tako da je ukupno vreme potrebno za sva prethodna kopiranja jednako veličini vektora. **Dakle, ne moramo se uopšte brinuti o veličini vektora, jer se STL sam o tome brine.**

Možemo reći da se ubacivanje na kraj vektora odvija konstantno (O(1))

Pristupanje svakom elementu vektora je konstantno, kao i izmena nekog elementa vektora.

Vremenska složenost funkcije size() je O(1), jer ***vector*** pamti svoju trenutnu veličinu.
http://www.cplusplus.com/reference/vector/vector/size/

Ipak, ubacivanje na početak ili sredinu vektora je linearno tj. O(n).

Vektor je vrlo jednostavan kontejner (klasa koja služi za čuvanje podataka). Nalazi se u biblioteci (zaglavlju) **vector**.

Primer 01

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{   vector <int> v;
    v.push_back(3); v.push_back(5);
    v.push_back(7);
    for(int i=10;i<14;i++)
      v.push_back(i);
    v[0]--;
    v[5]=-200;
    for(unsigned i=0;i<v.size();i++)
     cout << v[i] << ' ';
    cout << endl;
    return 0;
}
```

IZLAZ

```
 2 5 7 10 11 -200 13
```

U ovom zadatku smo koristili metod push_back() koji dodaje argument metoda na kraj vektora.
U 11. liniji programskog koda, pristupamo vektoru preko preopterećenog (overload) operatora [], kao da radimo sa nizovima (v[0]--;).
U 15. liniji programskog koda, koristimo metod size() koji vraća veličinu vektora.

Primer 02: vektor može čuvati bilo koji tip podataka

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

int main()
{   vector <string> vs;
    vs.push_back("ja"); vs.push_back("volim");
    vs.push_back("programiranje");

    for(unsigned i=0;i<vs.size();i++)
     cout << vs[i] << ' ';
    cout << endl;

    vector < vector<int> > matrica;
    for(int i=0;i<5;i++)
    {
        vector <int> v;
        matrica.push_back(v);
        for (int j=0;j<=i;j++)
        matrica[i].push_back(j);
    }

    for(unsigned i=0;i<matrica.size();i++)
    {
        for(unsigned j=0;j<matrica[i].size();j++)
         cout << matrica[i][j] << ' ';
        cout << endl;
    }


    return 0;
}
```

U 7.liniji programskog koda
   vector <string> vs;
kreiran je prazan vektor vs koji u sebi čuva stringove.

U 8. i 9. liniji smo u vektor ubacili 3 stringa koristeći metod push_back. Stringove smo ubacili na kraj vektora i time povećali veličinu vektora.

U 15. liniji definisali smo vektor koji u sebi sadrži vektor celih brojeva.
    vector < vector<int> > matrica;
Na taj način smo kreirali strukturu sličnu dvodimenzionom nizu celih brojeva.
Pri deklaraciji vektora matrica, morali smo voditi računa da ne izostavimo blanko karakter između karaktera > >,
jer bi deklaracija
 vector < vector<int>> matrica;

izazvala sintaksnu grešku.

U 27. liniji ispisujemo vrednost vektora pristupajući im pomoću operatora []. Moramo voditi računa da operator [] ne koristimo za mesta u vektoru koja nisu zauzeta (zauzeta su ona koja smo ubacili pomoću metoda push_back!!!).

Ako bismo iz našeg programskog koda izbacili 18. i 19. liniju programskog koda
vector <int> v;
matrica.push_back(v);
programski kod bi se uspešno kompajlirao, ali bi se verovatno srušio prilikom izvršavanja. Na takve greške moramo sami pazitii često ih je teško uočiti.

IZLAZ
ja volim programiranje
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4

Primer 03: vektor i iteratori

U STL-u postoje iteratori kako bi se pomoću njih moglo iterirati kroz neke strukture podataka u kojima se ne može definisati koji je element po redu, nego samo koji je element pre ili nakon nekog elementa. U vektorima se može direktno pristupiti nekom elementu što nije slučaj sa svim strukturama podataka koje ćemo kasnije obraditi.

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{   vector <int> v;
    for(unsigned i=0;i<15;i++)
     v.push_back(i);
    vector<int>::iterator it;
    for(it=v.begin(); it!=v.end();it++)
     cout << *it << ' ';
    cout << endl;
    return 0;
}
```

IZLAZ
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

U 10. liniji programskog koda kroz vektor v umesto brojačem, iteriramo iteratorom it. Deklaracija iteratora it je vector<int>::iterator it;
gde iterator je naziv tipa, a koristimo notaciju :: da odredimo kojoj klasi pripada.
U 10. liniji inicijalizujemo iterator na početak vektora, tj. dodeljujuemo it=v.begin();

Metod begin() vraća iterator na 1. element vektora, a metod end() vraća iterator na element iza poslednjeg.
Pomoću operatora ++ menjamo iterator tako da pokazuje na sledeći element.
Pomoću operatora != gledamo da li je it različit od nekog iteratora.

U 11. liniji programskog koda ispisujemo elemente vektora pomoću iteratora koristeći preopterećen operator * kako bismo pristupili elementu na koji iterator pokazuje.

Primer 04: još neki metodi klase vector

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{   vector <int> v(10); //v.size() je 10
    v[0]=-1; v[9]=99;
    v[4]=v[5]=100;
    v.push_back(2345);
    for(unsigned i=0;i<v.size();i++)
    cout << v[i] << ' ';
    v.pop_back();
    cout << v.front() << ' ' << v.back() << endl;
    v.clear(); //sve brise i size postaju 0
   if (v.empty()) cout << "v je prazan\n";
    return 0;
}
```

U 6. liniji programskog koda stvaramo vector<int> koji ima 10 elemenata postavljenih na 0. Time v.size() vraca 10, a
v.push_back(nesto); ubacuje na lokaciji v[10].
U 12. liniji metoda pop_back izbacuje poslednji element (tj. v[v.size()-1]) iz vektora u vremenskoj slozenosti O(1) i smanjuje v.size() za 1.
U 13. liniji programskog koda v.front() vraca prvi element vektora, tj. v[0], dok v.back() vraca poslednje element vektora tj. v[v.size()-1].
U 14.liniji pozivamo metod v.clear(); da bi se obrisali svi elementi vektora v i onda ce v.size() postati 0.
U 15. liniji pozivamo metod v.empty() koji ce vratiti true ako je v.size()==0

Primer 05: još neki metodi klase vector

```
#include <iostream>
#include <vector>
using namespace std;

inline void ispisi(const vector<int>& x)
{  for(int i=0;i<x.size();i++) cout << x[i] << ' ';
   cout << endl;
}

int main()
{   vector <int> v,q;
    v.insert(v.end(),7,1);
    v.insert(v.begin()+2,3);
    ispisi(v);
    q.insert(q.begin(),10,8);
    q.insert(q.end(),9);
    v.insert(v.begin()+4, q.end()-3, q.end());
    ispisi(v);ispisi(q);
```

```
    v.erase(v.begin()+2);
    v.erase(v.begin()+3,v.begin()+6);
    ispisi(v);
    vector <int> ve(3,4);
    ispisi(ve);
    v.swap(ve);
    ispisi(v);ispisi(ve);
    return 0;
}
```

IZLAZ

```
1 1 3 1 1 1 1 1
1 1 3 1 8 8 9 1 1 1 1
8 8 8 8 8 8 8 8 8 8 9
1 1 1 1 1 1 1
4 4 4
4 4 4
1 1 1 1 1 1 1
```

Napomene:

U 5. liniji programskog koda implementirana je funkcija koja ispisuje sadrzaj vektora. Kako bismo ubrzali rad funkcije, opredelili smo se da kao argument funkcije primamo referencu na vektor i zbog toga ga ne kopiramo. Kako bismo se zastitili od slucajne izmene vektora, argument je konstantna referenca.

U 12. liniji programskog koda

```
  v.insert(v.end(),7,1);
```

dodajemo sedam 1ca ispred iteratora v.end(), gde v.end() pokazuje ne element iza poslednjeg.

U 13.liniji programskog koda

```
  v.insert(v.begin()+2,3);
```

dodajemo broj 3 ispred iteratora v.begin()+2. To je iterator na 3. element vektora. Metod insert i erase imaju slozenost O(n).

Iteratori nisu jednaki za sve kontejnere iz STL-a, tako da iterator na vektor moze imati vecu funkcionalnost nego iterator na neki drugi kontejner.
Iterator na vektor moze se opteretiti + ili – ili pomeriti nekoliko elemenata unapred ili unazad (jer vektor interno podatke cuva u obliku niza), dok neki drugi iteratori to nece moci.

U 15. liniji

```
q.insert(q.begin(),10,8);
```

dodajemo 10 puta broj 8 na pocetak vektora q.

16. linija

```
 q.insert(q.end(),9);
```

moze da se napise i kao

```
  q.push_back(9)
```

17. linija

```
  v.insert(v.begin()+4, q.end()-3, q.end());
```

sadrzi metod insert koji prima tri iteratora.
  insert(it, begin, end) dodaje sve elemente između iteratora begin, end i to ispred iteratora it.

U 19. liniji

```
v.erase(v.begin()+2);
```

brišemo 3. element vektora v.

U 20. liniji
v.erase(v.begin()+3,v.begin()+6);
brišemo 4., 5., 6. element vektora v.

U 22. liniji
vector <int> ve(3,4);
kreiramo vektor ve koji sadrži tri četvorke.

Vektori se u lineranoj složenosti mogu pridruživati jedni drugima.
Na primer.
vector <int> s;
s=ve;
ispisi(s);

Vektori se u lineranoj složenosti mogu upoređivati operatorima == != <= < >= >.

U 24. liniji
v.swap(ve);
zamenjuju se vektori v i ve u vremenskoj slozenosti O(1).

Nakon trampe vektora metodom swap, svi iteratori koji pokazu na bilo koji od dva zamenjena vektora postaju nevazeci (moraju se ponovo izracunati).

Primer 06

Vremensko ograničenje: **1 s**
Memorijsko ograničenje: **64 mb**

Napiši program koji određuje koja bi od nekoliko datih reči trebalo da bude navedena prva u rečniku ako se ne pravi razlika između velikih i malih slova (za tu reč kažemo da je leksikografski ispred ostalih reči, tj. da je leksikografski najmanja među svim rečima). Sa standardnog ulaza se učitava linija teksta koja sadrži više reči razdvojenih sa po tačno jednim razmakom (u liniji postoji bar jedna reč, a iza poslednje reči nema razmaka). Na standardni izlaz ispisati reč iz učitane linije koja je leksikografski ispred ostalih reči.

Primer

Ulaz

Ko rano rani dve srece grabi

Izlaz

dve

Rešenje

Zadatak zahteva da se odredi minimum niza učitanih reči u odnosu na relaciju leksikografskog poretka. Da bi se napisao traženi program potrebno je učitavati reč po reč iz date linije, porediti dve reči leksikografski i pronaći minimum serije reči u odnosu na leksikografski poredak.

Krenimo od pitanja kako učitati reči. Ovo zavisi od programskog jezika. U jeziku C++ jednu reč možemo učitavati pomoću cin >> rec, a sve reči možemo učitati tako što ćemo ovo ponavljati sve dok uspeva tj. do EOF.

```
string rec;
while (cin >> rec)
  ...
```

Drugi način je da se upotrebi iterator ulaznog toka.

```
istream_iterator pocetak(cin), kraj;
for (auto it = pocetak; it != kraj; it++)
  ...
```

Ovaj način je naročito pogodan ako se minimum određuje pomoću bibliotečkih STL funkcija kojima je dovoljno samo iterator na početak i na kraj serije, jer tada nije čak potrebno pisati petlju.

DIGRESIJA

Ključno pitanje je kako organizovati ciklus u kojem se učitavaju brojevi sve dok se ne dostigne kraj ulaza. Način prepoznavanja kraja ulaza vrši se na različit način u različitim jezicima.

U jeziku C++ operator >> vraća referencu na ulazni tok (u našem slučaju to je cin) i konverziju toka u logičku vrednost (vrednost tipa bool) koja će imati vrednost true ako je prethodno čitanje uspelo, tj. false ako nije. Zato je najjednostavniji način da učitaju svi brojevi sa ulaza da se upotrebi petlja oblika

```
while (cin >> x)
  ...
```

Još jedan način da se u jeziku C++ učitaju svi elementi sa standardnog ulaza je da se koristi istream_iterator (definisan u zaglavlju <iterator>).

Ova šablonska klasa zahteva navođenja tipa podataka koji će se učitavati. Ako se u konstruktoru navede ulazni tok (u našem slučaju će to biti cin jer brojeve čitamo sa standardnog ulaza) iterator će ukazivati na tekuću poziciju u tom toku, a ako se u konstruktoru ne navedu argumenti dobiće se iterator koji ukazuje na kraj toka. Tako bi se učitavanje brojeva moglo izvršiti pomoću:

```
for (auto it = istream_iterator(cin); it != istream_iterator(); it++)
  ...
```

Korišćenje iteratora je naročito korisno u kombinaciji sa bibliotečkim STL funkcijama deklarisanim u zaglavljualgorithm o kojima će više biti reči u nastavku zbirke. Na primer, fukcija distance određuje broj elemenata između dva iteratora, tako da se ceo program može realizovati na sledeći način.

```
cout << distance(istream_iterator(cin), istream_iterator()) << endl;
```

Na kraju, obradu reči je moguće uraditi i tako što se cela linija učita u jednu nisku, a onda se iz te niske izdvaja jedna po jedna reč. Tokom tog postupka održavamo promenljivu start u kojoj se čuva početak naredne reči i koju možemo inicijalizovati na nulu. Svaku narednu reč pronalazimo tako što pronađemo poziciju narednog razmaka u liniji i izdvojimo karaktere između pozicije start i pozicije poz(uključujući prvu, ali ne i drugu). Nakon svake izdvojene reči, pozicija start se pomera na poziciju poz+1). Poslednja reč je specifična po tome jer iza nje nema razmaka, međutim, i ona se može izdvojiti na uniforman način kao i sve ostale, ako se nakon učitavanja linije na njen kraj doda jedan razmak. Za pronalaženje prvog pojavljivanja nekog karaktera (ili podniske) c počevši od neke pozicije p u niski s u jeziku C++ možemo koristiti metod s.find(c, p), a u jeziku C# s.IndexOf(c, p). Obe vraćaju indeks prvog pojavljivanja ako se cpronađe. Ako ne, onda find vraća specijalnu vrednost string::npos dok IndexOf vraća vrednost -1. Izdvajanje dela niske s između pozicija a i b uključujući karakter na poziciji a, a ne uključujući onaj na poziciji b u jeziku C++ možemo izvršiti pomoću s.substr(a, b - a), a u jeziku C# pomoću s.Substring(a, b - a) (argumenti su u oba slučaja početna pozicija i broj karaktera koji se izdvajaju). Izdvajanje reči može trajati sve dok se nakon pozicije start može pronaći razmak ili dok pozicija start ne pređe dužinu linije iz koje se reči izdvajaju.

Dakle, izdvajanje reč po reč se može realizovati na neki od narednih načina

```
string str;
getline(cin, str); str += ' ';
int start = 0, poz;
while ((poz = str.find(' ')) != string::npos) {
  string rec = str.substr(start, poz - start);
  ...
  start = poz + 1;
}
```

ili

```
string str;
getline(cin, str); str += ' ';
int start = 0;
while (start < str.size()) {
  int poz = str.find(' ', start);
  string rec = str.substr(start, poz - start);
  ...
  start = poz + 1;
}
```

Drugo pitanje koje treba razrešiti je kako izvršiti leksikografsko poređenje dve niske, ignorišući pri tom razliku između malih i velikih slova

Funkciju leksikografskog poređenja koja nam je potrebna možemo implementirati i samostalno. U petlji prolazimo kroz obe niske (dok ne dođemo do kraja bar jedne od njih) i poredimo karaktere na istim pozicijama. Da bismo zanemarili veličinu slova, pre poređenja karaktera oba pretvaramo u velika slova (ili mala, potpuno je svejedno), najbolje bibliotečkom funkcijom toupper koju smo uveli u zadatku transformacija_karaktera. Ako su karakteri na tekućoj poziciji različiti njihov međusobni odnos određuje i odnos celih niski, i funkcija može da vrati rezultat (prekidajući time ujedno i petlju). Na kraju, kada se utvrdi da je kraća niska prefiks duže (ili da su eventualno iste dužine), poređenje se svodi na poređenje dužina niski (kraća je po definiciji leksikografski ispred druge).

Umesto pisanja funkcije leksikografskog poređenja, mogli smo potegnuti i za malo jednostavnijim (ali neefikasnijim) rešenjem. Naime, pre poređenja niski mogli smo obe pretvoriti u mala (ili velika slova). Nažalost, ni za to ne postoji bibliotečka funkcija, ali jednostavno ju je implementirati (na primer, u petlji je moguće proći kroz sve elemente niske i na svaki primeniti funkciju toupper ili je moguće pozvati algoritam transform(str.begin(), str.end(), ::toupper)).
Na kraju, određivanje najmanje reči vršimo korišćenjem uobičajenog algoritma za određivanje minimuma serije (koji smo videli, na primer, u zadatku Najmanja temperatura). Najmanju reč inicijalizujemo na prvu reč koju pročitamo (ona sigurno postoji, po pretpostavci zadatka), a zatim u petlji čitamo ostale reči, poredimo ih sa minimalnom (korišćenjem opisanog leksikografskog poretka koji zanemaruje veličinu slova) i ako je tekuća reč manja od do tada minimalne, ažuriramo minimalnu.

Resenje 01

```
/*
poredi dve reci leksikografski, ne praveci razliku izmedju velikih i malih slova i vraca true ako i samo ako je prva rec leksikografski manja od druge

*/
```

```
#include <iostream>
#include <string>
#include <cctype>

using namespace std;

bool manja(const string& str1, const string& str2) {
  // dok u obe reci ima karaktera
  for (int i = 0; i < str1.size() && i < str2.size(); i++) {

    // poredimo tekuci karakter, transformisuci ga u veliko slovo
      char c1 = toupper(str1[i]), c2 = toupper(str2[i]);

    // ako su karakteri razliciti, leksikografski odnos niski je odredjen njihovim odnosom
    if (c1 != c2)      return c1 < c2;
}

  // procitali smo kracu rec (ili obe), a nismo naisli na karakter koji se razlikuje - odnos je odredjen duzinama
reci        return str1.size() < str2.size();
}


int main() {
  // ucitavamo liniju teksta koja sadrzi reci (do znaka za prelazak u novi red)
  string minRec;
  cin >> minRec;

  // ucitavamo rec po rec, dok ih ima
  string tekucaRec;
  while (cin >> tekucaRec)

    // ako je tekuca rec leksikografski ispred najmanje, azuriramo najmanju rec
    if (manja(tekucaRec, minRec))      minRec = tekucaRec;

  // prijavljujemo najmanju rec
  cout << minRec << endl;
}
```

Resenje 02

/*poredi dve reci leksikografski, ne praveci razliku izmedju velikih i malih slova i vraca true ako i samo ako je prva rec leksikografski manja od druge */

U jeziku C++ (u standardima, još uvek) ne postoji standardna bibliotečka podrška za leksikografsko poređenje koje ne pravi razliku između velikih i malih slova. Podaci tipa string se mogu porediti relacijskim operatorima <, <= i slično, poređenje će biti leksikografsko, ali će se razlikovati velika i mala slova.

Funkcije poput strcasecmp i strcmpi su nasleđene iz jezika C i nisu standardizovane, tako da nije sigurno da će na svim platformama raditi.

Umesto toga, postoji bibliotečka funkcija lexicographical_compare (#include <algorithm>) koja vrši leksikografsko poređenje dva segmenta (nizova, vektora, stringova) koji se zadaju iteratorima na početak i iza kraja. Ovoj funkciji je moguće proslediti i funkciju (bilo imenovanu, bilo anonimnu) koja se koristi da

uporedi dva tekuća elementa tih segmenata i koja treba da vrati true samo ako je tekući element prvog raspona manji od tekućeg elementa drugog raspona. Kada se porede niske, ta funkcija prima dva karaktera, prevodi eventualna velika slova u mala (pozivom funkcije toupper) i onda ih poredi operatorom <.

Složenost bibliotečke funkcije
lexicographical_compare(v1.begin(), v1.end(), v2.begin(), v2.end()) je barem
2*min(N1, N2) poređenja
gde N1 rastojanje prvog raspona std::distance(v1.begin(), v1.end()) i N2 = std::distance(v2.begin(), v2.end()).

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <iterator>
#include <algorithm>
using namespace std;

bool manja(const string& str1, const string& str2) {
  return lexicographical_compare(str1.begin(), str1.end(),
                                 str2.begin(), str2.end(),
                                        [](char ca, char cb) {
                                          return toupper(ca) < toupper(cb);
                                        });
}

int main() {

  // odredjujemo leksikografski najmanju rec ucitanu sa ulaza
  // zanemarujuci razliku izmedju velikih i malih slova
     cout << *min_element(istream_iterator<string>(cin),
                          istream_iterator<string>(),
                          manja) << endl;
}
```

Dakle, minimum (leksikografski najmanju reč) možemo određivati i bibliotečkim funkcijama. Ako su u jeziku C++ reči predstavljene u kolekciji kroz koju se može prolaziti iteratorom, onda najmanji element možemo odrediti funkcijom min_element koja vraća iterator na najmanji element. To može biti slučaj ili ako učitamo sve reči u vektor ili ako koristimo iterator ulaznog toka istream_iterator.

```
vector reci;
...
cout << *min_element(reci.begin(), reci.end(), poredi) << endl;
```

ili

```
cout << *min_element(istream_iterator(cin),
                     istream_iterator(),
                     poredi);
```

Funkciji min_element se kao treći parametar prosleđuje funkcija poređenja koja poredi dva elementa i vraća logičku vrednost true ako i samo ako je njen prvi parametar manji od drugog.

Rešenje 03

```
  // ucitavamo rec po rec u vektor i odredjujemo leksikografski najmanju rec u vektoru

#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <algorithm>
using namespace std;

int main() {
  // ucitavamo rec po rec u vektor, dok ne dodjemo do kraja ulaza
  vector<string> reci;
  string rec;
  while (cin >> rec)
    reci.push_back(rec);

  // ucitavanje u niz se moze ostvariti i sa:
  // copy(istream_iterator<string>(cin), istream_iterator<string>(),
  //      back_inserter(reci));

  auto min = min_element(reci.begin(), reci.end(),
                    [](const string& str1, const string& str2) {
                              return lexicographical_compare(str1.begin(), str1.end(),
                            str2.begin(), str2.end(),  [](char a, char b) {
                                    return toupper(a) < toupper(b);
                                });
                             });
  cout << *min << endl;
}
```

Rešenje 04

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

/* Poredi dve reci leksikografski, ne praveci razliku izmedju velikih i malih slova.
Za rezultat važi zakon trihotomije:  Vraca pozitivan broj ako je prva rec leksikografski veca od druge, nulu
ako su recu jednake  i negativan broj ako je prva rec manja od druge */

int poredi(const string& str1, const string& str2) {
  // dok u obe reci ima karaktera
  for (int i = 0; i < str1.size() && i < str2.size(); i++)
    // poredimo tekuci karakter, transformisuci ga u veliko slovo
    // ako su karakteri razliciti, leksikografski odnos niski je
    // odredjen njihovim odnosom
    if (toupper(str1[i]) != toupper(str2[i]))
      return toupper(str1[i]) - toupper(str2[i]);
  // procitali smo kracu rec (ili obe), a nismo naisli na karakter
```

```
  // koji se razlikuje - odnos je odredjen duzinama reci
  return str1.size() - str2.size();
}

int main() {
  // ucitavamo liniju teksta koja sadrzi reci (do znaka za prelazak u novi red)
  string str;
  getline(cin, str);
  // linija se prosiruje razmakom da bi se sve reci izdvajale na uniforman nacin
  str += ' ';

  // odredjuje poziciju razmaka iza prve reci
  size_t pos = str.find(' ');
  // izdvajamo prvu rec koja je za sada leksikografski najmanja
  string minRec = str.substr(0, pos);

  // pozicija iza razmaka iza prve reci
  size_t start = pos + 1;
  // radi dok ne stignes do kraja stringa str
  while (start < str.size()) {
    // pronalazimo narednu prazninu krenuvsi od pozicije start
    size_t pos = str.find(' ', start);
    // izdvajamo rec od pozicije start do praznine (ne ukljucujuci nju)
    string tekucaRec = str.substr(start, pos - start);
    // ako je minRec leksikografski iza tekucaRec, azuriramo minRec
    if (poredi(tekucaRec, minRec) < 0)
      minRec = tekucaRec;
    // pozicija iza razmaka iza pozicije start
    start = pos + 1;
  }
  // prijavljujemo najmanju rec
  cout << minRec << endl;
}
```

Rešenje 05

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

// u stringu str pretvara sva mala slova u velika
string uVelikaSlova(string str) {
  // svako slovo u reci trasnformisemo u veliko
  for (int i = 0; i < str.length(); i++)
    str[i] = toupper(str[i]);
  // ovo moze lakse da se uradi i na sledeci nacin
  // transform(str.begin(), str.end(), ::toupper);
  return str;
}

int main() {
  // ucitavamo liniju teksta koja sadrzi reci (do znaka za prelazak u novi red)
  string str;
  getline(cin, str);
```

```
  // linija se prosiruje razmakom da bi se sve reci
  // izdvajale na uniforman nacin
  str += ' ';

  // odredjuje poziciju razmaka iza prve reci
  size_t pos = str.find(' ');
  // izdvajamo prvu rec koja je za sada leksikografski najmanja
  string minRec = str.substr(0, pos);
  // pozicija iza razmaka iza prve reci
  size_t start = pos + 1;
  // radi dok nalazi razmak u stringu str krenuvsi od pozicije start
  // pri tom pamteci poziciju razmaka u promenljivu pos
  while ((pos = str.find(' ', start)) != string::npos) {
    // izdvajamo rec od pozicije start do praznine (ne ukljucujuci nju)
    string tekucaRec = str.substr(start, pos - start);
    // ako je minRec leksikografski iza tekucaRec, azuriramo minRec
    // da bi se zanemarali razlika slova prilikom poredjenja, obe reci se
    // prvo transformisu u velika slova
    if (uVelikaSlova(tekucaRec) < uVelikaSlova(minRec))
      minRec = tekucaRec;
    // startna pozicija sledece pretrage - pozicija iza pronadjenog razmaka
    start = pos+1;
  }
  // prijavljujemo najmanju rec
  cout << minRec << endl;
}
```

LISTE

Klasa list iz STL-a definisana je u biblioteci (zaglavlju) list i predstavlja impementaciju dvostruko povezane liste.
Svaki element u takvoj listi pokazuje na element ispred i iza sebe i na taj nacin omogucuje ubacivanje i izbacivanje elemenata u vremenskoj slozenosti O(1).
Dohvatanje n-tog elementa u takvoj listi ima vremensku slozenost O(n).
PORedjENJE SA kontejnerom vector: Vremenska slozenost izbacivanja u vector je O(n), ali dohvatanje n-tog elementa slozenosti O(1).

Primer 07: uvod u rad sa listom

```
#include <iostream>
#include <list>
using namespace std;

int main()
{ list<int> l;
  l.push_back(3);
  l.push_front(1);
  l.insert(++l.begin(),2);
  list<int>::iterator it;
  for(it=l.begin();it!=l.end();it++)
   cout << * it << ' ';
    return 0;
}
```

IZLAZ
1 2 3

U 6. liniji
  list<int> l;
kreiramo praznu listu celih brojeva koju nazovemo l.

Mogli smo kreirati i listu sa 10 praznih elemenata na sledeci nacin (slicno kao kod vektora)
list<int> l(10);

Mogli smo kreirati i listu sa 5 elemenata jednakih 7 na sledeci nacin (slicno kao kod vektora)
list<int> l(5,7);

U 7. liniji
 l.push_back(3);
dodaje se 3 na kraj liste. Vremenska slozenost ove operacije je O(1).

U 8. liniji
 l.push_front(1);
dodaje se 1 na pocetak liste. Vremenska slozenost ove operacije je O(1).

Takodje, kao i za vektor postoje metodi pop_front(), pop_back(). Postoje i metodi front() i back() koje u vremenskoj slozenosti O(1) vracaju prvi i poslednji element liste.
Pomocu navedenih metoda se lista moze koristiti kao queue ili stack sa neogranicenim kapacitetom. O kapacitetu se brine lista.

U 9. liniji
  l.insert(++l.begin(),2);
je ubacen broj 2 ispred 2. elementa.

Naredbu l.insert(++l.begin(),2);
nismo smeli napisati kao
  l.insert(l.begin()+1,2);
jer iteratori liste nisu isti kao i iteratori vektora (slabiji su iteratori liste) i nemaju preopterecen operator sabiranja i oduzimanja, nego samo operatore ++ i --. To je i logicno, jer bi se operatori + i – morali izvoditi u slozenosti O(n).

Metod ***insert(iterator, vrednost)*** se izvrsava u vremenskoj slozenosti O(1), za razliku od vektora koji tu operaciju obavlja u vremenskoj slozenosti O(n). Lista takodje ima metod ***insert*** i ***erase*** kao i vektor, ali su kod liste brži.

Liste se mogu, kao i vektori, uporedjivati u slozenosti O(n) pomocu operatora == != <= < > >=, te se takodje mogu pridruzivati u O(n).

Liste, za razliku od vektora, nemaju preopterecen operator [] i nijma **se ne moze pristupati kao** da se radi o nizu.

Zato 10. liniju programskog koda NISMO mogli ispisati u obliku
for (int i=0;i<l.size();i++)   cout << l[i] << ' ';

Ispis liste smo obavili u 10. i 11. liniji programskog koda u formi
  for(it=l.begin();it!=l.end();it++)
   cout << * it << ' ';
Koristili smo iteratore koji su inicijalizovani kao it=l.begin().
U 11. liniji pristupamo pojedinacnom elementu sa *it.

Liste (kao i vektori) mogu sadrzavati bilo sta, tako da je moguce definisati:
list<vector<list<string> > > listaVektoraSaListomStringova;

Metod erase moze izazvati problem ako se ne koristi pazljivo. Ako napisemo naredbu
l.erase(it);
nakon toga ne smemo dalje koristiti iterator **it**, jer postaje nevazeci (tj. mora se iznova izracunati). To, takodje, vazi i za vektore. Razlog je što brišući element, gubimo element, te ne možemo iz njega pročitati koji element je sledeći. Želimo li nakon brisanja elementa imati i dalje važeći iterator, onda to možemo učiniti ovako:
it=l.erase(it);

Metod *erase* vraca iterator na element nakon obrisanog.
Takodje, mozemo napisati
l.erase(it++);
To je moguce zbog nacina na koji je operator ++ preopterecen.

Ako napisemo
*it=2;
l.insert(it,1);
onda se element 1 ubacuje pre elementa 2, a iterator *it* ce i dalje pokazivati na element s vrednoscu 2.

Lista takodje ima metodu *swap* kao i vektor, koja je O(1), kao i metodi *clear* i *empty*.

Lista ima dodatni metod *reverse* koji okrene listu u O(n) i metod *sort* koji sortira listu u slozenosti O(n log n).

Primer 08: metodi za rad sa listom

```
#include <iostream>
#include <list>
using namespace std;

void ispis (list<int> &li)
{ for(list<int>::iterator x=li.begin();x!=li.end();x++)
    cout << *x << ' ';
    cout << endl;
}

int main()
{  list<int> l;
   l.push_back(3); l.push_back(2);
   l.push_back(4); l.push_back(1);
   ispis(l);
   l.reverse();
   ispis(l);
   l.sort();
   ispis(l);
   l.reverse();
   ispis(l);
    return 0;
}
```

U 5. liniji ne mozemo pisati
const list<int> &li
umesto

list<int> &li

Zbog toga je definisan poseban iterator zvan const_iterator pomocu kog se moze iterirati (znaci nije nepromenjiv), ali

PAIR

Klasa *pair* (tj. par) nalazi se u biblioteci *utility*, ali je često ne moramo uključiti (include-ovati), jer je uključe razne druge biblioteke. Klasa *pair* je veoma jednostavna i koristi se kada želimo da čuvamo dva različita tipa podataka pod istim nazivom.
Par ne podržava iteriranje, te nije kontejner. Često se dešava da funkcije koje moraju vratiti dve vrednosti, vrate par vrednosti.

Primer 09

```
#include <iostream>
#include <utility>
using namespace std;

int main()
{   pair<int,double> a;
    pair<int,int> b(3,4);
    pair<int,int> c(6,2);
a=make_pair(2,3.14);
cout << a.first << ' ' << a.second << endl;
cout << (b<c) << ' ' << (b==c) << endl;
b=c;
cout << (b==c)  << endl;

pair<pair<int,int>,int> trojka;
trojka=make_pair(make_pair(1,2),3);
cout << trojka.first.first << ' ';
cout << trojka.first.second << ' ';
cout << trojka.second << endl;

    return 0;
}
```

U liniji
pair<int,double> a;
kreiran je par celog i realnog broja

U naredne dve linije kreirani su parovi <int,int> kojima su pridružene vrednosti.

Naredbom a=make_pair(2,3.14); pridružene su vrednosti paru a.

Paru se može pristupiti pomoću članova *first* i *second*.
Parovi se mogu i porediti i to tako što se najpre porede prvi elementi iz para, a ao su oni jednaki porede se i drugi elementi. Zato je par b(3,4) < c(6,2)

Par se cesto koristi za kreiranje ovakvih struktura:
vector <pair<int,int> > v1;
vector <pair< vector<int>, list<int> > > v2;

Primer 10

1. Data je pravougaona mrežu ulica predgrađa Beograda poznatog kao Bisergrad. Postoji 50000 vertikalnih ulica u smeru sever-jug (označenih x-koordinatama od 1 do 50000) i 50000 horizontalnih ulica u smeru istok-zapad (označenih y-koordinatama od 1 do 50000). Sve ulice su dvosmerne. Presek horiznotalne i vertiklane ulice naziva se biserraskršće. Stanovnici Bisergrada mogi biti vrlo neodgovorni i nesaavesni vozači, te je iz zbog bezbednosnih potreba, gradonačelnik Bisergrada postavio semafore na **N** biserraskršća. **Put** između dva biserraskršće je **opasan** ako na njemu negde postoji **skretanje bez semafora**, a inače je bezopasan. Nije moguće osigurati da svi putevi budu bezopasni, ali gradonačelniku je dovoljno da **između svaka dva semafora** bar **jedan od najkraćih puteva** bude **bezopasan**. Na žalost, trenutni raspored semafora to ne osigurava. Vaš je zadak postaviti **još neke semafore** (manje od 700 000) tako da skup semafora (koji sadrži i stare i nove semafore!) zadovoljava traženi zahtev. Razmislite i pomozite Bisergradu!

ULAZ

U prvoj liniji standardnog ulaza nalazi se prirodan broj **N** ($2 \leq \mathbf{N} \leq 50000$), broj postavljenih semafora. U svakoj od sledećih **N** linija nalazi se lokacija jednog semafora, predstavljena prirodnim brojevima **X** i **Y** ($1 \leq \mathbf{X}, \mathbf{Y} \leq 50000$), koordinatama vertikalne i horizontalne ulice koje se seku u tom biserraskršću. Svi semafori biće jedinstveni.

IZLAZ

Ispišite lokacije novih semafora, svaku u posebnoj liniji. Dozvoljeno je postavljanje više semafora na istu lokaciju.

Broj novih semafora mora biti **manji od 700 000**.

PRIMERI TEST PODATAKA

ULAZ	**IZLAZ**
2 1 1 3 3	1 3
3 2 5 5 2 3 3	3 5 5 3
5 1 3 2 5 3 4 4 1 5 2	1 5 3 3 3 5 4 2 4 3

Resenje:

Date semafore sortirajmo po x-koordinati. Neka je **x'** srednja (median) x koordinata u tom nizu. Neka je **A** skup datih semafora levo od **x'**, a **B** skup datih semafora desno od **x'**.

Povežimo bezopasnim putem svaki par semafora **a**, **b** takvih da je **a** iz skupa **A**, te **b** iz skupa **B**. Kako? Tako da dodamo nove semafore na lokacije (**x'**, **y**) za sve y-koordinate **y** iz skupova **A** i **B**. Sada za semafore **a** i **b** imamo bezopasan put (**xa**, **ya**) → (**x'**, **ya**) → (**x'**, **yb**) → (**xb**, **yb**).

Još je preostalo povezati međusobno semafore unutar skupa **A**, kao i semafore unutar skupa **B**. To činimo tako da rekurzivno ponovimo opisani postupak posebno za skup **A** i posebno za skup **B**. Ovaj način razmišljanja naziva se *podeli pa vladaj* (divide and conquer).

Koliki je broj dodatih semafora? Budući da delimo skup na dva dela, najveća je dubina rekurzije O(log **N**). Posmatrajmo neki početni semafor na lokaciji (**x**, **y**). U najgorem slučaju, na svakoj dubini rekurzije on će biti uključen u jedan skup i tamo će generisati jedan novi semafor (**x'**, **y**). Dakle, jedan početni semafor generise O(log **N**) novih semafora, što daje ukupno O(**N** log **N**) novih semafora.
Tačan broj uz pažljivu implementaciju ispada manji od 700000.

```
#include <algorithm>
#include <cstdio>
using namespace std;

const int NN = 100005;
pair<int, int> t[NN];

void divcon(int lo, int hi) {
  if (lo + 2 == hi) {
    printf("%d %d\n", t[lo].first, t[lo + 1].second);
    return;
  }
  if (lo + 2 > hi) return;
  int mid = (lo + hi) / 2;
  int x = t[mid].first;
  for (int i = lo; i < hi; ++i)
    if (i != mid)
      printf("%d %d\n", x, t[i].second);
  divcon(lo, mid);
  divcon(mid + 1, hi);
}

int main () {
  int n;
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) {
    scanf("%d%d", &t[i].first, &t[i].second);
  }
  sort(t, t + n);
  divcon(0, n);
}
```

SKUP (klasa set)

Klasa set definisana je u biblioteci set.

SET je kontejner koji implementira veoma korisnu strukturu podataka, eng. red-black tree koja u sebi održava elemente poređane po veličini (ili po nekom našem poretku).

SET je sličan matematičkom pojmu skup. Svaki element je jedinstven. Ubacivanje i uklanjanje elementa sprovodi se za vreme O(log n). Svim elementima može se pristupati (za razliku od priority_queue) i može se iterirati kroz njih pomoću iteratora koji podržavaju ++ i --.

Primer 11

```
#include <iostream>
#include <set>
using namespace std;
inline void ispis(const set<int> &s)
{  set<int>::const_iterator it;
     for(it=s.begin();it!=s.end();it++)
       cout << *it << ' ';
   cout << endl;
}

int main()
{ set <int> s;
```

```
  s.insert(3); s.insert(5); s.insert(2);
  s.insert(6); s.insert(1); s.insert(-3);
  ispis(s);
  for(int i=4;i<10;i++) s.insert(i);
  ispis(s);
  s.erase(3);
  s.erase(s.find(7));
  ispis(s);
  s.erase(s.lower_bound(-2));
  ispis(s);
  s.erase(s.upper_bound(5));
  ispis(s);
  cout << "Ima ih: " << s.size() << endl;
  s.clear();
  if (s.empty()) cout << "prazan" << endl;
    return 0;
}
```

MAPA

Klasa map je struktura set<pair<kljuc, vrednost> >

Struktura map koju pretrazujemo pomocu kljuca kojim pristupamo polju vrednost. Za svaki kljuc postoji jedinstvena vrednost. Mapa ima preopterecen operator [] pomocu kog preko kljuca pristupamo polju vrednost. Mapa se nalazi u biblioteci map.

Primer 12 – mapa

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

int main()
{   map<string,int> m;
    m["dvadeset"]=20;
    m[string("pet")]=5;
    m.insert(pair<string,int>("dva",2));
    m.insert(make_pair(string("sto"),100));
    m.insert(make_pair("pedeset",50));
    map<string,int>::iterator it;
    for(it=m.begin(); it!=m.end();it++)
     cout << it->first << ' ' << it->second << endl;
     return 0;
}
```

IZLAZ

```
dva 2
dvadeset 20
pedeset 50
pet 5
sto 100
```

Primer 13

Pekar Joca prodaje burek u **tri oblike** – **četvrtina, polovina**, **tri četvrtine.** Za potrebe proslave skole koja se nalazi u blizini pekare, dobio je porudzbine. Skolska deca su mala niko od njih ne može pojesti ceo burek ali svako je prijavio svojoj uciteljici koliko parce Jocinog bureka može pojesti. Uciteljice instiraju da pekar Joca postuje njihov spisak tako da svako dete dobije **tačnu kolicinu zeljenog bureka.** Napišite program koji će pomoći Joci da odredi koliki je **minimalni** broj bureka koje mora napraviti kako bi svako dete dobilo tačno onoliki komad koliki želi.

Ulaz

U prvom redu standardnog ulaza se nalazi ceo broj N, $1 \leq N \leq 10{,}000$, broj dece koja ce jesti burek.
U svakom od sledećih redova standardnog ulaza nalazi se veličina bureka koju svako od dece želi pojesti tj. razlomak
1/4, **1/2** ili **3/4**.

Izlaz

U prvi i jedini red standardnog ulaza ispisati traženi minimalni broj bureka koje Joca treba da napravi.

PRIMERI

ulaz
3
1/2
3/4
3/4
izlaz
3

ulaz
5
1/2
1/4
3/4
1/4
1/2
izlaz
3

ulaz
6
3/4
1/2
3/4
1/2
1/4
1/2
izlaz
4

Resenje:

```
#include <cstdio>
#include <map>
#include <string>

using namespace std;

map<string, int> broj;

int main( void ) {
```

```
  int n;
  scanf( "%d", &n );
  for( int i = 0; i < n; ++i ) {
    char linija[10];
    scanf("%s", linija );
    broj[linija]++;
  }

  int burek_broj = broj["3/4"] + broj["1/2"] / 2;

  broj["1/4"]  = max( broj["1/4"] - broj["3/4"], 0 );
  broj["1/4"] += 2 * (broj["1/2"] % 2);

  burek_broj += (broj["1/4"] + 3) / 4;

  printf( "%d\n", burek_broj );

  return 0;
}
```

Multimap

Map ima iste metode kao i set. Slicno kao sto postiji multiset u biblioteci set, tako postoji i multimap u biblioteci map. Ali, multimap nema operator [], jer za vise istih kljuceva moze imati razlicite vrednosti.

Multimap kontejner može da se koristi umesto hash strukture. Svaki element je sastavljen od dva dela: ključ i vrednost. Struktura dozvoljava ponavljanje ključeva po kom se vrši pretraga .
Ključevi po kojima se pretražuju moraju biti uporedljivi, tj. mora da može da se nad njima definiše relacija <. Za standardne tipove podataka, uključujući i biblioteku string, već imaju definisne ove operatore za poređenje.

Da bi mogla da se koristi ova struktura, potrebno je na početku programa da se uključi sledeća direktiva
#include <map>

Primer

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
// Definisanje multimap-a
multimap<string,int> mymm;
// Definisanje iteratora za ovakav tip multimape
multimap<string,int>::iterator it;
// Unos elemenata u multimap-u
mymm.insert(pair<string,int>("marko",10));
mymm.insert(pair<string,int>("ana",20));
mymm.insert(pair<string,int>("ana",30));
mymm.insert(pair<string,int>("mica",40));
mymm.insert(pair<string,int>("maca",50));
mymm.insert(pair<string,int>("kuca",60));
mymm.insert(pair<string,int>("ana",60));

// Stampanje elemenata multimap-e
```

```
for (it = mymm.begin();it != mymm.end();++it)
cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;

// Stampanje svih vrednosti pri svim pojavljivanjima kljuca ana
pair<multimap<string,int>::iterator,multimap<string,int>::iterator> ret;
ret = mymm.equal_range("bliksa");
for (it=ret.first; it!=ret.second; ++it)
cout << " " << (*it).second;
cout << endl;
// Broj elemenata sa kljucem ana
cout<<endl<<"Broj elemenata sa kljucem 'ana' : "<<mymm.count("ana");
// Dodavanje elementa na pocetak multiset-a
it=mymm.begin();
mymm.insert(it,pair<string,int>("bla",123));
// Stampamo elemente multimap-e
for (it = mymm.begin();it != mymm.end();++it)
cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;
return 0;
}
```

STRINGOVI

Schlemiel the Painter's algorithm

„Neke od najvećih grešaka koje ljudi prave, čak i na najvišim nivoima arhitekture, potiču od nerazumevanja nekoliko jednostavnih stvari na najnižim nivoima“. https://www.joelonsoftware.com/2001/12/11/back-to-basics/

„I think that some of the biggest mistakes people make even at the highest architectural levels come from having a weak or broken understanding of a few simple things at the very lowest levels.“

Schlemiel the Painter's algorithm, označava neefikasnu metodologiju razvoja softvera, u kojoj programer ne uočava osnovne probleme na vrlo niskom nivou razvoja softvera. **Pojam je 2001. uveo Joel Spolsky, programer i bloger (blog: Joel on Software).**

Spolsky je problem ilustrovao korišćenjem sledeće šale: Schlemiel radi na iscrtavanju isprekidanih linija po sredini puta. Svakog dana on iscrta manje linija nego prethodnog.
Upitan zbog čega, Schlemiel se žali da je to zbog toga što je svakim danom sve dalje od kantice sa bojom.

Shlemiel gets a job as a street painter, painting the dotted lines down the middle of the road. On the first day he takes a can of paint out to the road and finishes 300 yards of the road. “That’s pretty good!” says his boss, “you’re a fast worker!” and pays him a kopeck.
The next day Shlemiel only gets 150 yards done. “Well, that’s not nearly as good as yesterday, but you’re still a fast worker. 150 yards is respectable,” and pays him a kopeck.
The next day Shlemiel paints 30 yards of the road. “Only 30!” shouts his boss. “That’s unacceptable! On the first day you did ten times that much work! What’s going on?”
“I can’t help it,” says Shlemiel. “Every day I get farther and farther away from the paint can!”

U programskom jeziku C, ovaj vid problema srećemo kad koristimo bibliotečku funkciju **strcat** za višestruko uzastopno nadovezivanje (konkatenaciju) stringova (odnosno niza karaktera koji se završava terminalnom nulom)...

```
void strcat( char* dest, char* src )  //verzija iz knjige K&R – Kernighan, Ritchie
{
```

```
    while (*dest) dest++;
    while (*dest++ = *src++);
}
```

...gde se pozicija od koje u rezultujućem stringu treba da počne string koji se nadovezuje se ponovo izračunava svaki put od početka rezultujućeg stringa, pošto nije upamćena prilikom prethodnog nadovezivanja. (jer Schlemiel nije poneo sa sobom dužinu stringa, tj. kanticu sa bojom!!!)

Prvi korak u svakoj implementaciji standardne bibliotečke C-ovske funkcije za nadovezivanje stringova jeste utvrđivanje dužine prvog stringa ispitivanjem da li je svaki karakter niza, počev od prvog, terminalna nula. Zatim se drugi string kopira na kraj prvog, čime se efektivno vrši njihovo nadovezivanje. Na kraju se dužina rezultujućeg stringa (odnosno pozicija terminalne nule u njemu) odbacuje.

Npr. u slučaju višestrukog nadovezivanja:

```
strcat( buffer, "John" ); /* string "John" se nadovezuje na buffer */
strcat( buffer, "Paul" ); /* a onda se na to nadoveže "Paul" */
strcat( buffer, "George" ); /* na rezultat se nadoveže string "George" */
strcat( buffer, "Ringo" ); /* a onda i string "Ringo" */
```

nakon što se Paul nadoveže na John, dužina JohnPaul, je poznata unutar strcat(), ali se odbacuje nakon što se metod završi. Onda se vrši nadovezivanje George na JohnPaul, gde strcat() ponovo kreće od prvog karaktera niza ("J") da bi pronašao terminalnu nulu. Svaki naredni poziv strcat() mora iznova da izračuna dužinu bafera pre nadovezivanja drugog imena na njegov kraj.

Analogno sa Schlemiel-om, koji ne nosi kanticu boje (dužinu stringa) sa sobom, svi uzastopni pozivi strcat() moraju da „prođu“ dužinu prvog stringa kako bi utvrdili gde treba prekopirati drugi. Što se više podataka dodaje u bafer svakim pozivom strcat(), to je terminalna nula dalja od njegovog početka, što znači da su naredni pozivi strcat() sve sporiji - upravo kao što je i Schlemiel-ov put do kantice sve duži.

Problemi ilustrovani Spolsky-jevim primerom često ostaju neprimećeni od strane programera koji koriste jezik visokog nivoa, a imaju malo ili nedovoljno znanja o funkcionalnim detaljima na kojima on počiva.
Slična pojava postoji i pri nadovezovanju stringova u jeziku C++ operatorima + i =.
Naime, za nadovezivanje koristite operator +=, umesto operatora + i operatora =

U prvom slučaju (npr. s += x) se tekući string s proširuje datim delom x
U drugom slučaju (npr. s = s + x) najpre se kreira novi string u koji bi se kopirao sadržaja stringa s, zatim i stringa x i tek onda bi se on dodelio stringu s, što bi dovelo do velike neefikasnosti
Dakle, u jeziku C++ je dovoljno koristiti operator += i program će raditi kako treba.

Eksperiment 01

```
#include <iostream>
#include <string>

using namespace std;

int main() {

  string str="12345";
 // getline(cin, str);  //ucitavamo string do kraja linije

  // string u koji ce biti upisano resenje
  string strFrankenstajn = "";
  // ucitava poziciju i duzinu sve dok se ne dodje do kraja standardnog ulaza

   for (int i=1;i<=10000;i++) strFrankenstajn=strFrankenstajn+str;
```

```
  // ispisujemo konacno resenje
  cout << strFrankenstajn << endl;
}
```

Eksperiment 02

```
#include <iostream>
#include <string>

using namespace std;

int main() {

  string str="12345";
 // getline(cin, str);  //ucitavamo string do kraja linije

  // string u koji ce biti upisano resenje
  string strFrankenstajn = "";
  // ucitava poziciju i duzinu sve dok se ne dodje do kraja standardnog ulaza

   for (int i=1;i<=10000;i++)
     strFrankenstajn+=str;

  // ispisujemo konacno resenje
  cout << strFrankenstajn << endl;
}
```

1. Napišite C ili C++ program koji od unete reči pravi novu reč sastavljenu iz delova početne reči. Na primer, za unetu reč
dabrakabmra
6 2
3 3
1 1
0 3
9 2
program gradi i ispisuje reč abrakadabra (prvo idu 2 karaktera krenuvši od pozicije 6 tj. ab, zatim 3 karaktera od pozicije 3 tj. rak, zatim 1 karakter od pozicije 1 tj. a, zatim tri karaktera od pozicije 0 tj. dab i na kraju dva karaktera od pozicije 9 tj. ra).

ULAZ: U prvoj liniji standardnog ulaza je string koji predstavlja datu reč. U narednim linijama se unose parovi: pozicija u datom stringu i dužina podstringa.
IZLAZ: Reč koja se dobija navedenim postupkom.

Primer

Ulaz

maranaakkabbikopa
13 3
16 1
8 3
3 3

Izlaz

kopakabana

Ideja rešenja

Izlazni rezultat (string strFrankenstajn) koji se traži dobija se nadovezivanjem manjih delova originalnog stringa, učitanog iz prve linije teksta (getline(cin, str);).

Nakon toga u petlji učitavamo poziciju i broj karaktera svakog dela, sve dok ne dođemo do kraja ulaza (EOF).

while (cin >> poz >> n)

Za učitanu poziciju i dužinu deo originalnog stringa izdvajamo korišćenjem metode substr u jeziku C++

Ovaj metod ima 2 parametra - poziciju od koje kreće izdvajanje podstringa i broj karaktera koji će se izdvojiti.

Svaki izdvojeni podstrin nadovezuje sa na string strFrankenstajn u kojem se čuva do sada formirani deo rešenja. Njega na početku, pre petlje, inicijalizujemo na prazan string, a onda mu u telu petlje, na kraj, dopisujemo deo koji smo upravo odredili. Dopisivanje na desnu stranu stringa u oba jezika možemo vršiti operatorom +=.

PONOVIMO da je u jeziku C++ je značajno za nadovezivanje koristiti operator +=, umesto operatora + i operatora =.

Postoje programski jezici (poput Python, C#, Java) u kojima su stringovi imutabilni (nepromenjivi). Dakle, u jeziku C# svako nadovezivanje stringova, pa i ono sa += proizvodi nove stringove. Naime, svi stringovi u jeziku C# su imutabilni i ne mogu se nikada menjati (što ima prednosti prilikom dodeljivanja stringova i njihovog prosleđivanja, ali u mnogim situacijama ima i mane). Da bi se ovaj problem prevazišao, potrebno je umesto tipa string koristiti mutabilan tip StringBuilder (zapravoSystem.Text.StringBuilder) koji je upravo namenjen za kreiranje teksta koji nastaje nadovezivanjem manjih delova. On se na početku inicijalizuje na praznu nisku korišćenjem StringBuilder sb = new StringBuilder("");, dok se proširuje niskom xkorišćenjem sb.Append(x);

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  // ucitavamo znake u string str do znaka za kraj linije
  string str;
  getline(cin, str);

  // string u koji ce biti upisano resenje
  string strFrankenstajn = "";
  // ucitava poziciju i duzinu sve dok se ne dodje do kraja standardnog ulaza
  int poz, n;
  while (cin >> poz >> n)
    // dopisuje trazenom stringu podstringove stringa str od zadate
    // pozicije poz, zadate duzine n
    strFrankenstajn += str.substr(poz, n);

  // ispisujemo konacno resenje
  cout << strFrankenstajn << endl;
}
```