Ovdje su dani opisi algoritama te izvorni kôdovi rješenja zadataka s XII. Međunarodne informatičke olimpijade održane u Pekingu 2000. godine. Iako su rješenja namijenjena svim zainteresiranima za rješavanje složenijih informatičkih zadataka ona ipak podrazumijevaju neko poznavanje osnovnih algoritama. Izvorni kôdovi dani su samo u programskom jeziku C, ali su pisani takvim stilom da se mogu potpuno razumjeti samo uz poznavanje minimuma sintakse C-a.

Izvorni kôdovi namijenjeni su kompajliranju sa **gcc** kompajlerom (za bilo koju platformu). Da bi ispravno radili s Borlandovim C kompajlerom potrebno je na nekim mjestima napraviti manje preinake (npr. tip long umjesto int, alokacija memorije itd.).

Ante Đerek, djerek@student.math.hr

Palindrome

U ovom zadatku je potrebno za zadanu riječ odrediti koliko joj je najmanje znakova potrebno umetnuti tako da dobivena riječ bude palindrom tj. da se čita jednako sprijeda i straga.

Najprije se pozabavimo palindromima. Palidrom koji ima paran broj slova zapravo je građen od dvije riječi koje su međusobno obrnute (npr. **abccba** se sastoji od međusobno obrnutih riječi **abc** i **cba**). Ukoliko palindrom ima neparan broj slova onda se on sastoji od dvije međusobno obrnute riječi i jednog znaka između (npr. **abcdcba** se sastoji od riječi **abc** i **cba** između kojih se nalazi znak **d**). U prvom slučaju reći ćemo da je *centar* palindroma između trećeg i četvrtog slova, a u drugom slučaju ćemo reći da je centar palindroma četvrto slovo.

Zadatak možemo riješiti razmatrajući sve moguće slučajeve za centar palindroma. Neka je zadana riječ **A** koja se sastoji od **N** slova. Označimo slova riječi **A** s **A(1), A(2), …, A(N)**. Najkraći palindrom koji sadrži riječ **A**, a kojem je centar između **i**-tog i **i+1**-og slova riječi **A** možemo naći tako da odredimo najkraću riječ koja sadrži riječi **A(1)A(2)…A(i)** i **A(N)A(N-1)…A(i+1)**. Na primjer, ako je zadana riječ **A=mgebemc** i zanima nas najkraći palindrom koji ju sadrži, a kojem je centar negdje između trećeg i četvrtog slova riječi **A**, mi tražimo najkraću riječ koja sadrži **mge** i **cmeb**. To je očito riječ **cmgeb**, a traženi palindrom je **cmgebbegmc**. Slično, ako nas zanima najkraći palindrom koji sadrži riječ **A**, a kojem je centar **i**-to slovo riječi **A** možemo ga odrediti tražeći najkraću riječ koja sadži riječi **A(1)A(2)…A(i-1)** i **A(N)A(N-1)…A(i+1)**.

Pokušajmo sada vidjeti kako ćemo za dvije zadane riječi odrediti najkraću riječ koja ih obje sadrži. Primjetimo da se u samom zadatku ne traži najkraći palindrom nego samo njegova duljina, pa shodno tome nije potrebno određivati najkraću riječ koja sadrži dvije zadane riječi nego samo njenu duljinu. Lako se vidi da njena duljina ovisi o najvećem zajedničkom podnizu riječi tj. vrijedi da je duljina najkraće riječi koja sadrži dvije zadane riječi jednaka sumi njihovih duljina umanjenoj za duljinu najvećeg zajedničkog podniza tih riječi. Primjerice, za riječi **mge** i **cmeb** najveći zajednični podniz je **me**, a najkraću riječ koja ih obje sadrži dobijemo tako da u ovaj najkraći zajednični podniz umetnemo ostala slova iz riječi odgovarajućim redosljedom (**cmgeb**).

Dakle, da bi riješili zadatak potrebno je za svaki **i** odrediti duljinu najduljeg zajedničkog podniza riječi **A(1)A(2)…A(i)** i **A(N)A(N-1)…A(i+1)**, te također riječi **A(1)A(2)…A(i-1)** i **A(N)A(N-1)…A(i+1).** Ovo najefikasnije možemo riješiti dinamičkim programiranjem:

- Neka **Zaj(i, j)** označava duljinu najdužeg zajedničkog podniza riječi **A(1)A(2)…A(i)** i **A(N)A(N-1)…A(j)**
- Vrijedi rekurzivna relacija:

  Zaj(i, j) = Zaj(i-1, j+1) + 1, ako je **A(i) = A(j)**

  Zaj(i, j) = Max (Zaj(i-1, j), Zaj (i, j+1)) inače
- Na temelju ove relacije možemo konstruirati tablicu **Zaj** računajući redak po redak, svaki redak od zadnjeg stupca prema prvome
- Rješenje zadatka možemo iščitati iz tablice **Zaj**, to je najmanji od svih brojeva **N-2*Zaj(i, i+1)**, **N-2*Zaj(i, i+2)+1**
- Primjetimo da nije potrebno pamtiti u memoriji cijelu tablicu **Zaj** nego svaki redak možemo konstruirati samo na temelju prethodnog.

Vremenska složenost predloženog algoritma je **O(N²)**, dok je prostorna složenost **O(N)**. Korisno je primjetiti da ovako efikasnim algoritmom ne možemo izračunati sam palindrom, nego samo njegovu duljinu. Naime, za rekonstrukciju polinoma potrebno je pamtiti čitavo polje **Zaj**, te bi takav algoritam imao prostornu složenost **O(N²).**

PALIN.C

```
/* ---- palin.c ---- */
#include <stdio.h>

#define MAXSLOVA  5001
#define SIZE      2
```

```
#define ZAJ(I,J)  (zaj[(I+SIZE)%(SIZE)][(J)])
#define MAX(A,B)  ((A)>(B)?(A):(B))

char rijec[MAXSLOVA];
int n;
int sol;
int zaj[SIZE][MAXSLOVA];

void rijesi (void)
{
    int i, j;

    sol=n-1;

    for (j=0; j<=n; j++)
        ZAJ(-1, j)=0;

    for (i=0; i<=n-2; i++)
    {
        for (j=n-1; j>i; j--)
            if (rijec[i]==rijec[j]) ZAJ(i, j)=ZAJ(i-1, j+1)+1;
            else ZAJ(i, j)=MAX(ZAJ(i-1, j), ZAJ(i, j+1));
        if (n-2*ZAJ(i, i+1)<sol)
           sol=n-2*ZAJ(i, i+1);
        if (i<n-2 && n-2*ZAJ(i, i+2)-1<sol)
           sol=n-2*ZAJ(i, i+2)-1;
    }
}

int main (void)
{
    FILE *input, *output;

    input=fopen("palin.in", "r");
    output=fopen("palin.out", "w");

    fscanf(input, "%d", &n);
    fscanf(input, "%s", rijec);

    rijesi();

    fprintf(output, "%d\n", sol);

    fclose(input);
    fclose(output);
    return 0;
}
```

Car Parking

U ovom zadatku je zadan niz od **N** brojeva kojeg želimo sortirati. Sortiramo tako da u jednoj *rundi* možemo odabrati bilo kojih **W** brojeva te zamijeniti njihova mjesta na proizvoljan način. Potrebno je napraviti plan za sortiranje koji će zadani niz brojeva sortirati u najviše $\mathbf{R}=\lceil \mathbf{N}/(\mathbf{W}-1) \rceil$ rundi ($\lceil \mathbf{a} \rceil$ označava najmanji prirodan broj veći ili jednak od **a**).

Neka **A(1), A(2), …, A(N)** označava zadani niz brojeva, te neka **B(1), B(2), …, B(N)** označava isti niz brojeva ali sortiran uzlazno. Dakle, želimo od niza **A** dobiti niz **B** u najviše **Q** rundi tako da u svakoj rundi zamijenimo mjesta nekih **W** elemenata niza. Reći ćemo da je **i**-ti element na *dobrom mjestu* ako je **A(i) = B(i)**. Da bi niz sortirali u najviše **R** rundi dovoljno je osmisliti algoritam koji će u svakoj rundi (osim možda u zadnjoj) broj elemenata na dobrom mjestu povećati za barem **W-1.**

Konstruirajmo sada takav algoritam. Algoritam će elemente koje su na dobrom mjestu ostavljati na miru, i u svakoj rundi će barem **W-1** elemenata koji nisu na dobrom mjestu staviti na dobro mjesto.

- Nađimo prvi element koji nije na dobrom mjestu, neka je to npr. element s indeksom **K(1)**.
- Nađimo sada neki element različit od **A(K(1))** koji se nalazi na mjestu predviđenom za **A(K(1))** (takav mora postojati). Neka je to npr. element s indeksom **K(2)**.
- Nađimo sada neki element različit od **A(K(2))** koji se nalazi na mjestu predviđenom za **A(K(2)).** Neka je to npr. element s indeksom **K(3)** itd.
- Ovaj postupak ponavljamo sve dok ne dođemo opet do elementa s indeksom **K(1)** (to se mora prije ili kasnije dogoditi). Pretpostavimo da ovaj postupak staje nakon **M** koraka.

- Ukoliko je **M** veći ili jednak od **W** u toj rundi zamijenimo elemente tako da element s indeksom **K(1)** stavimo na mjesto **K(2)**, element s indeksom **K(2)** stavimo na mjesto **K(3)**, …, element s indeksom **K(W)** stavimo na mjesto **K(1)**.
- Ukoliko je **M** manji od **W** tih **M** elemenata ćemo u trenutnoj rundi razmještati kao gore te tražimo sljedeći cikus kojim ćemo popuniti ostale moguće poteze u rundi.

Algoritam koji rješava ovaj problem samo ponavnja gornji proces dok niz nije potpuno sortiran. Kako je garantirano da će se u svakoj rundi broj elemenata na dobrom mjestu povećati za **W-1**, očito je da će algoritam generirati rješenje od najviše **R** rundi.

Analizirajmo složenost ovog algoritma. Osnovni korak ovog algoritma za neki broj **C** traži element u nizu **A** koji je različit od **C**, a nalazi se na mjestu na kojem se treba nalaziti **C**. Ovaj korak će se izvoditi najviše jednom za svaki element u nizu **A** koji nije na svom mjestu na početku, dakle najviše **N** puta. Ukoliko ovo traženje implementiramo tako da niz **A** pretražujemo od početka, tada je složenost izvođenja tog koraka jednaka **O(N)** pa je složenost čitavog algoritma **O(N²)** što je nedovoljno efikasno. Jedno moguće poboljšanje je da za svaki element unaprijed izračunamo pozicije na kojima se nalazi taj element u sortiranom nizu. Iako asimptotska složenost ostaje ista, ovakav algoritam je dovoljno efikasan da riješi sve test primjere koji zadovoljavaju ograničenja zadatka.

CAR.C

```
/* ---- car.c ---- */
#include <stdio.h>
#include <stdlib.h>

#define MAXCARS    20001
#define MAXTYPES   51
#define MAXWORKERS 51
#define NO         -1
#define FIRST      -2

int n, m, w, q;
int tip[MAXCARS];
int dobar[MAXCARS];
int start[MAXTYPES+1];
int q;
int nrundi;
int npoteza;
int prije[MAXWORKERS];
int poslije[MAXWORKERS];

FILE *input, *output;

int cmp (const void *a, const void *b)
{
    if ( * (int *) a < * (int *) b ) return -1;
    if ( * (int *) a > * (int *) b ) return 1;
    return 0;
}

void citaj_ulaz (void)
{
    int i, j;

    fscanf(input, "%d %d %d", &n, &m, &w);
    q=ceil( (double) n / (double) (w-1) );
    for (i=0; i<n; i++)
    {
        fscanf(input, "%d", &tip[i]);
        tip[i]--;
        dobar[i]=tip[i];
    }
    qsort(dobar, n, sizeof(dobar[0]), cmp);
    i=0;
    for (j=0; j<n; j++)
        while (dobar[j]>=i && i<m)
        {
           start[i]=j;
           i++;
        }
    for ( ; i<=m; i++)
        start[i]=n;
```

```
}

int prvi_krivi (void)
{
    int i;

    for (i=0; i<n; i++)
        if (tip[i]!=dobar[i])
           return i;

    return NO;
}

void dump (void)
{
    int i;

    fprintf(output, "%d", npoteza);
    for (i=0; i<npoteza; i++)
        fprintf(output, " %d %d", prije[i]+1, poslije[i]+1);
    fprintf(output, "\n");
}

int na_mjestu (int tp)
{
    int i;

    for (i=start[tp]; i<start[tp+1] ; i++)
        if (tip[i]!=tp) return i;

    return NO;
}

void rijesi (void)
{
    int i, j;
    int a, tipa, b;
    int tmp;
    int prvi;

    nrundi=0;
    npoteza=0;
    a=prvi_krivi();
    while (a!=NO)
    {
        prvi=a;
        tipa=tip[a];
        tip[a]=FIRST;
        b=na_mjestu(tipa);
        while (b!=NO)
        {
            prije[npoteza]=a;
            poslije[npoteza]=b;
            npoteza++;
            if (npoteza==w)
            {
                nrundi++;
                if (poslije[npoteza-1]==prvi)
                {
                    dump();
                    npoteza=0;
                }
                else
                {
                    poslije[npoteza-1]=prvi;
                    dump();
                    prije[0]=prvi;
                    poslije[0]=b;
                    npoteza=1;
                }
            }
            tmp=tip[b]; tip[b]=tipa; tipa=tmp;
            if (tipa==FIRST) break;
            a=b;
```

```
            b=na_mjestu(tipa);
        }
        a=prvi_krivi();
    }
    if (npoteza!=0)
    {
        dump();
        nrundi++;
    }
    for ( ;nrundi<q; nrundi++)
        fprintf(output, "1 1 1\n");
}

int main (void)
{
    input=fopen("car.in", "r");
    output=fopen("car.out", "w");

    citaj_ulaz();

    q=n/(w-1);
    if (n%(w-1)!=0) q++;
    fprintf(output, "%d\n", q);

    rijesi();

    fclose(input);
    fclose(output);
    return 0;
}
```

Median Strength

U ovom zadatku je za zadanu permutaciju **N** prirodnih brojeva potrebno odrediti indeks mediana tj. srednjeg elementa. Pri određivanju možemo se samo služiti funkcijom **Med3** koja za tri zadana indeksa vreća indeks srednjeg od ta tri elementa. Potrebno je napraviti rješenje koje će što je moguće manje puta pozvati funkciju **Med3**.

Najprije primjetimo da samo koristeći funkciju **Med3** nikako ne možemo odrediti većeg ili manjeg od neka dva elementa permutacije, međutim to nam niti ne treba jer tražimo median koji je veći od točno **(N-1)/2** elemenata permutacije i manji od točno **(N-1)/2** elemenata permutacije. Kako nam je za ovaj predloženi algoritam ipak korisno uspoređivati brojeve po veličini koristimo sljedeći trik: Izaberimo bilo koja dva indeksa **A** i **B** te jednostavno proglasimo element sa indeksom **A** većim. Za neki drugi indeks **C** reći ćemo da je

- *veći od **A** i **B*** ako je **Med3(A, B, C)=A**
- *veći od **B** i manji od **A*** ako je **Med3(A, B, C)=C**
- *manji od **A** i **B*** ako je **Med3(A, B, C)=B**

Ovako smo možda potpuno okrenuli uređaj, ali to je svejedno jer je median opet **(N+1)/2**-ti najmanji element.

Konstruirati ćemo "divide and conquer" algoritam koji će za zadanih **M** brojeva i broj **K** naći **K**-ti po redu najmanji broj koristeći pritom samo funkciju **Med3**. Algoritam radi tako da:

- Izaberemo bilo koja dva indeksa **C** i **D**, te uspoređujući ih s početno odabranim indeksima **A** i **B** odredimo većeg i manjeg među njima, pretpostavimo, na primjer, da je **C** veći, a **D** manji
- Uspoređujući ostalih **M-2** indeksa sa **C** i **D**, te indekse grupiramo u tri kategorije, one veće od **C**, one između **C** i **D**, te one manje od **D**
- Prebrojavajući elemente u svakoj kategoriji ili smo odredili rješenje (**C** ili **D**) ili rekurzivno primjenjujemo algoritam na onu kategoriju u kojoj se traženi element nalazi.

Na primjer ako su nepoznati brojevi (6,3,4,1,2,5,7) indeksirani od 1 do 7 i u njoj tražimo indeks petog najmanjeg elementa, početni korak algoritma bi bio ovakav:

- Izaberemo indekse 1 i 2 te neka je, na primjer, indeks 1 veći
- Uspoređujući sve ostale indekse s 1 i 2 grupiramo ih u tri kategorije
  - indeks veći od 1 je samo 7
  - indeksi između 1 i 2 su 3 i 6
  - indeksi manji od 2 su 4 i 5
- Traženi element je drugi najmanji u skupu indeksa 3 i 6

Analizirajmo složenost ovog algoritma, tj. pokušajmo odrediti koliko je najmanje potrebno poziva funkcije **Med3** da bi se odredio median. Intuitivno, u svakom koraku "divide and conquer" algoritma se postavi onoliko pitanja koliko ima kandidata, te se broj kandidata otprilike smanji na trećinu. Dakle algoritam je u općenitom slučaju dovoljno

efikasan da median od 1499 brojeva odredi sa manje od 7777 pitanja. Međutim, u najgorem slučaju u svakom koraku algoritma se eliminiraju samo dva kandidata (na primjer, nemamo sreće pa uvijek izaberemo dva najmanja od preostalih kandidata). Dakle, složenost je u najgorem slučaju kvadratna što nije dovoljno dobro. Primjetimo da efikasnost čitavog algoritma počiva na funkciji koja će od preostalih kandidata odabrati dva pomoću kojih će vršiti raspodjelu u kategorije. Ukoliko ta funkcija jednostavno bira prva dva kandidata onda će algoritam loše raditi na degeneriranim test primjerima, npr. na sortiranim permutacijama. Rješenje je u tome da se indeksi za raspodjelu biraju *slučajno* među preostalim kandidatima. Ovako će za svaku zadanu permutaciju algoritam sa jednakom vjerojatnošću raditi dovoljno efikasno.

Ovaj algoritam radi garantirano točno, ali nema garancije da radi dovoljno efikasno. Međutim lako je se uvjeriti (bilo intuicijom ili matematičkim metodama) da je vjerojatnost da algoritam ne radi dovoljno brzo na proizvoljnom test primjeru zanemariva. Ovakva vrsta algoritma zove se *probabilistički algoritam*.

MEDIAN.C

```
/* ---- median.c ---- */
#include <stdio.h>
#include <stdlib.h>
#include "device.h"

#define GOTOVO     0
#define NOVO       1
#define GORE       2
#define SREDINA    3
#define DOLJE      4
#define MAX        1500

int status[MAX];
int n;

int veci, manji;
int index;
int sol;

int ask (int a, int b, int c)
{
    return Med3(a+1, b+1, c+1)-1;
}

/* Slucajno biram novi indeks, nije savrseno ali je dovoljno dobro */
int izaberi (void)
{
    int i;

    i=rand()%n;
    while (status[i]!=NOVO)
          i=(i+1)%n;
    return i;
}

/*
   Medju elementima sa statusom NOVO, te elementima veci i manji trazim
   'index'-ti najmanji element tako da usporedim elemente veci i manji sa
   svima ostalima, te ili nadjem rjesenje ili problem svedem na manji
*/

void nadji (void)
{
    int i;
    int tmp;
    int ngore, nsredina, ndolje;
    int nmanji, nveci;

    ngore=nsredina=ndolje=0;

    /* Postavljam pitanja */
    for (i=0; i<n; i++)
        if (status[i]==NOVO)
        {
            tmp=ask(manji, veci, i);
            if (tmp==manji)
            {
```

```
            status[i]=DOLJE;
            ndolje++;
        }
        if (tmp==i)
        {
            status[i]=SREDINA;
            nsredina++;
        }
        if (tmp==veci)
        {
            status[i]=GORE;
            ngore++;
        }
    }

/* Mozda je 'manji' rjesenje */
if (index==ndolje+1)
{
    sol=manji;
    return ;
}

/* Mozda je 'veci' rjesenje */
if (index==ndolje+1+nsredina+1)
{
    sol=veci;
    return ;
}

/* Mozda je rjesenje dolje */
if (index<=ndolje)
{
    for (i=0; i<n; i++)
        if (status[i]!=DOLJE) status[i]=GOTOVO;
        else status[i]=NOVO;

    if (ndolje==1)
    {
        sol=izaberi();
        return ;
    }

    nveci=izaberi(); status[nveci]=GOTOVO;
    nmanji=izaberi(); status[nmanji]=GOTOVO;
    if (ask(manji, nveci, nmanji)==nmanji)
    {
        tmp=nveci; nveci=nmanji; nmanji=tmp;
    }

    veci=nveci; manji=nmanji;
}

/* Mozda je rjesenje u sredini */
if (index>ndolje+1 && index<=ndolje+1+nsredina)
{
    for (i=0; i<n; i++)
        if (status[i]!=SREDINA) status[i]=GOTOVO;
        else status[i]=NOVO;

    if (nsredina==1)
    {
        sol=izaberi();
        return ;
    }

    nveci=izaberi(); status[nveci]=GOTOVO;
    nmanji=izaberi(); status[nmanji]=GOTOVO;
    if (ask(veci, nveci, nmanji)==nmanji)
    {
        tmp=nveci; nveci=nmanji; nmanji=tmp;
    }

    veci=nveci; manji=nmanji;
    index=index-ndolje-1;
```

```
    }

    /* Mozda je rjesenje gore */
    if (index>ndolje+1+nsredina+1)
    {
        for (i=0; i<n; i++)
            if (status[i]!=GORE) status[i]=GOTOVO;
            else status[i]=NOVO;

        if (ngore==1)
        {
            sol=izaberi();
            return ;
        }

        nveci=izaberi(); status[nveci]=GOTOVO;
        nmanji=izaberi(); status[nmanji]=GOTOVO;
        if (ask(veci, nveci, nmanji)==nveci)
        {
            tmp=nveci; nveci=nmanji; nmanji=tmp;
        }

        veci=nveci; manji=nmanji;
        index=index-ndolje-1-nsredina-1;
    }
}

int main (void)
{
    int i;

    n=GetN();
    for (i=0; i<n; i++)
        status[i]=NOVO;

    veci=izaberi(); status[veci]=GOTOVO;
    manji=izaberi(); status[manji]=GOTOVO;

    index=n/2+1;
    sol=-1;

    while (sol<0)
          nadji();

    Answer(sol+1);
    return 0;
}
```

Post Office

U ovom zadatku zadane su koordinate niza sela poredanih uz ravnu cestu te je potrebno odabrati određeni broj sela u kojima će se sagraditi pošte tako da ukupna suma udaljenosti od svakog sela do najbliže pošte bude najmanja moguća. U ostatku teksta ćemo za zadani raspored pošti po selima ukupnu sumu udaljenosti od svakog sela do najbliže pošte nazivati *cijena* rasporeda. Dakle, u zadatku je potrebno odrediti raspored pošti s minimalnom cijenom.

Promotrimo najprije slučaj kada je potrebno izgraditi samo jednu poštu. Neka je, na primjer, zadano **V** sela sa koordinatama **D(1), D(2), …, D(V)** u uzlaznom redosljedu. Neka se pošta, na primjer. nalazi u **i**-tom selu, te promatrajmo promjenu cijene kada poštu premjestimo u susjedno (**i+1**) selo. Za svaki grad lijevo od **i**-tog doprinos cijeni poveća se za **D(i+1)-D(i)**, dok se za svaki grad desno od **i**-tog doprinos cijeni smanji za istu svotu. Dakle, cijena se promijeni za **(L-D)*(D(i+1)-D(i))** gdje **L** označava broj sela lijevo od **i**-tog sela (**L=i-1)**, a **D** označava broj sela desno od **i**-tog sela (**D=V-i**). Iz ovog rezultata sada lako slijedi činjenica da je ukupna cijena minalna kada se pošta nalazi u selu sa srednjim indeksom tj. ako je **V** neparan, minimalna cijena se postiže kada je pošta u selu (**V+1)/2**, a ako je **V** paran onda je optimalan izbor za poštu selo s indeksom **V/2** ili **V/2+1**.

Sada se koncentrirajmo na opći problem. Dakle želimo **P** pošta smjestiti u **V** sela tako da ukupna cijena bude minimalna. Neka je zadan jedan raspored s minimalnom cijenom. Promotrimo sva sela koja gravitiraju zadnjoj (najdesnijoj) pošti tj. kojima je ona najbliža. Na primjer, uzmimo da zadnjih **N** sela gravitira zadnjoj pošti dok ostalih **V-N** sela gravitiraju nekima od prvih **P-1** pošta. Zaboravimo nakratko tih zadnjih **N** sela i zadnju poštu te promatrajmo samo prvih **V-N** sela i prvih **P-1** pošta. Očigledno je da su tih **P-1** pošta tako raspoređene da je cijena za tih **V-N** sela minimalna, naime, inače bi bolji raspored smanjio cijenu za ovih **V-N** sela, ali bi i smanjio ukupnu cijenu u početnom

problemu jer zadnjih **N** sela gravitiraju zadnjoj pošti te drugačiji raspored prvih **P-1** pošti ne mijenja njihov doprinos ukupnoj cijeni.

Iz prethodnog rasuđivanja slijedi dinamički algoritam za rješenje problema. Za zadanih **V** sela i **P** pošta, minimalnu cijenu i optimalan raspored ćemo odrediti tako da isprobamo sve moguće vrijednosti za **N** - broj sela koje gravitiraju zadnjoj pošti te tako svedemo problem na manje potprobleme istog tipa.

- Označimo sa **Min(i, j)** najmanju moguću cijenu za raspored prvih **i** pošta među prvih **j** sela.
- Označimo sa **Cij(a, b)** najmanju moguću cijenu za raspored jedne pošte među selima od **a** do **b** (uključujući i **a** i **b**), to znamo izračunati prema gornjim razmatranjima.
- Vrijedi rekurzivna relacija:
  - **Min(i, j)** = najmanji od brojeva **Min(i-1, k) + Cij(k+1, j),** gdje k ide od **i-1** do **j-1**, **k** označava indeks zadnjeg (najdesnijeg) sela koja gravitira nekoj od prvih **i-1** pošti, gorespomenuti **N** je zapravo **j-k**.
- Na temelju ove relacije možemo konstruirati tablicu **Min** računajući redak po redak
- Prilikom računanja tablice **Min** konstruiramo i tablicu **How** koja pamti **k** u kojem se postiže minimum
- Minimalna cijena je broj **Min(P, V)**, dok sam raspored konstruiramo na temelju tablice **How**
- U stvarnoj implementaciji ne računamo stalno vrijednost **Cij(k+1, j)** već je dinamički konstruiramo

Vremenska složenost ovog algoritma je $\mathbf{O(V*P^2)}$, dok je prostorna složenost **O(V*P)**.

POST.C

```
/* ---- post.c ---- */
#include <stdio.h>

#define MAXSELA   301
#define MAXPOSTA  31
#define NO        -1

int nsela, nposta;
int dist[MAXSELA];

int min[MAXPOSTA][MAXSELA];
int how[MAXPOSTA][MAXSELA];

void citaj_ulaz (void)
{
    int i;
    FILE *input;

    input=fopen("post.in", "r");

    fscanf(input, "%d %d", &nsela, &nposta);
    for (i=0; i<nsela; i++)
        fscanf(input, "%d", &dist[i]);
}

void rijesi (void)
{
    int i, j, k;
    int cijena;
    int tmp;

    min[0][0]=0;
    how[0][0]=NO;
    for (i=1; i<nsela; i++)
    {
        min[0][i]=min[0][i-1]+dist[i]-dist[i/2];
        how[0][i]=NO;
    }

    for (i=1; i<nposta; i++)
        for (j=i; j<nsela; j++)
        {
            min[i][j]=NO;
            cijena=0;
            for (k=j-1; k>=i-1; k--)
            {
                cijena+=dist[(k+1+j+1)/2]-dist[k+1];
                tmp=min[i-1][k]+cijena;
```

```
                if (min[i][j]==NO || tmp<min[i][j])
                {
                    min[i][j]=tmp;
                    how[i][j]=k;
                }
            }
        }
}

void ispisi_rjesenje (void)
{
    int i, j;
    int sol[MAXPOSTA];
    FILE *output;

    output=fopen("post.out", "w");

    j=nsela-1;
    for (i=nposta-1; i>=0; i--)
    {
        sol[i]=(how[i][j]+1+j)/2;
        j=how[i][j];
    }

    fprintf(output, "%d\n", min[nposta-1][nsela-1]);
    for (i=0; i<nposta; i++)
        fprintf(output, "%d ", dist[sol[i]]);
    fprintf(output, "\n");

    fclose(output);
}

int main (void)
{
    citaj_ulaz();
    rijesi();
    ispisi_rjesenje();
    return 0;
}
```

Walls

U ovom zadatku nalazimo se u zemlji sastavljenoj od gradova od kojih su neki povezani zidovima te regija koje ti zidovi omeđuju. Nekoliko ljudi, svi iz različitih gradova žele se sastati na jednom mjestu pa je potrebno odrediti regiju takvu da ukupna suma broja zidova koje svaki čovjek prijeđe putem na sastanak bude najmanja moguća.

Zaboravimo na trenutak gradove, zidove i regije i probajmo riješiti problem u terminima teorije grafova. Dakle, neka je zadan neusmjereni graf, tj. skup vrhova od kojih su neki povezani bridom. Neka su na grafu zadani neki istaknuti vrhovi, potrebno je odrediti vrh takav da je suma odaljenosti od tog vrha do svih istaknutih vrhova najmanja moguća. Pod terminom *udaljenost* dva vrhova podrazumjevamo duljinu najkraćeg puta između njih tj. najmanji broj bridova koje je potrebno prijeći da dođemo od jednog do drugog. Jedno jednostavno rješenje koje je, s obzirom na zadane veličine ulaza, dovoljno efikasno za primjenu u ovom zadatku je sljedeće:

- Najprije za svaki par vrhova izračunamo udaljenost između njih, na primjer Floydovim algoritmom (vidi npr. R. Sedgewick: Algorithms, Addison-Wesley 1988).
- Za svaki vrh izračunamo sumu udaljenosti do istaknutih vrhova te odaberemo najpovoljniji vrh.

Rješenje zadatka se sada samo nameće, ako regije predstavljaju vrhove grafa, a zid između regija predstavlja brid koji povezuje dva vrha, onda nam prethodno opisani algoritam daje rješenje s tim da za svaki zadani grad isprobavamo iz koje je njemu susjedne regije najbolje krenuti na sastanak.

Najteži dio rješenja je način predstavljanja gradova, zidova i regija u memoriji računala. Jedan od dobrih načina je sljedeći:

- Za svaki par gradova **a** i **b** neka **Regija(a, b)** i **Regija(b, a)** označavaju regije sa dvije strane zida koji povezuje gradove **a** i **b.**
- Za svaki par regija **c** i **d** neka **Susjedni(c, d)** bude 1 ako regije **c** i **d** dijele zid, a 0 inače
- Kako su ulazni podaci takvi da su regije dane kao niz gradova u konzistentnom poredku (unutrašnje u smjeru kazaljke na satu, a vanjska regija obratno) tablice **Regija** i **Susjedni** se vrlo jednostavno ispune na temelju ulaznih podataka.

Rješenje zadaka se sada jednostavno dobije primjenom gore opisamog algoritma na ovako orgranizirane podatke. Vremenska složenost tog algoritma je **O(M³+L*N*M)**, gdje **M** označava broj regija, **L** označava broj članova kluba, a **N** broj gradova. Prvi sumand u ocjeni je složenost Floydovog algoritma, dok je drugi složenost određivanja najoptimalnije regije za sastanak nakon što su poznati svi parovi najkraćih puteva. Prostorna složenost je **O(M²+N²)**, prvi sumand je veličina tablice **Regije**, dok je drugi veličina tablice **Susjedni**.

WALLS.C

```
/* ---- walls.c ---- */
#include <stdio.h>

#define MAXREGIONS  200
#define MAXTOWNS    250
#define NO          -1

int ngradova, nregija;
int regija[MAXTOWNS][MAXTOWNS];
int susjedni[MAXREGIONS][MAXREGIONS];
int npocetnih;
int pocetni[MAXTOWNS];
int sol, minput;

void citaj_ulaz (void)
{
    int i, j;
    int a;
    int novi, stari, prvi;
    FILE *input;

    input=fopen("walls.in", "r");
    fscanf(input, "%d", &nregija);
    fscanf(input, "%d", &ngradova);
    fscanf(input, "%d", &npocetnih);

    for (i=0; i<npocetnih; i++)
    {
        fscanf(input, "%d", &pocetni[i]);
        pocetni[i]--;
    }

    for (i=0; i<ngradova; i++)
        for (j=0; j<ngradova; j++)
            regija[i][j]=NO;
    for (i=0; i<nregija; i++)
        for (j=0; j<nregija; j++)
            if (i==j) susjedni[i][j]=0;
            else susjedni[i][j]=NO;

    for (i=0; i<nregija; i++)
    {
        fscanf(input, "%d", &a);
        fscanf(input, "%d", &stari);
        stari--;
        prvi=stari;
        for (j=0; j<a; j++)
        {
            if (j<a-1)
            {
               fscanf(input, "%d", &novi);
               novi--;
            }
            else
                novi=prvi;
            regija[stari][novi]=i;
            if (regija[novi][stari]!=NO)
            {
                susjedni[i][regija[novi][stari]]=1;
                susjedni[regija[novi][stari]][i]=1;
            }
            stari=novi;
        }
```

```
    }

    fclose(input);
}

int cijena (int grad)
{
    int i, j, c;
    int min;

    c=0;
    for (i=0; i<npocetnih; i++)
    {
        min=NO;
        /*
           Odredjujem minimalnu udaljenost od grada 'pocetni[i]'
           do grada 'grad'
        */
        for (j=0; j<ngradova; j++)
            if (regija[pocetni[i]][j]!=NO)
               if (min==NO || susjedni[regija[pocetni[i]][j]][grad]<min)
                  min=susjedni[regija[pocetni[i]][j]][grad];
        c=c+min;
    }
    return c;
}

void rijesi (void)
{
    int i, j, k;
    int tmp;

    /*
      Floydovim algoritnom odredjujem minimalnu udaljenost izmedju svakog
      para regija
    */
    for (k=0; k<nregija; k++)
        for (i=0; i<nregija; i++) if (susjedni[i][k]!=NO)
            for (j=0; j<nregija; j++) if (susjedni[k][j]!=NO)
                if (susjedni[i][j]==NO || susjedni[i][j]>susjedni[i][k]+susjedni[k][j])
                   susjedni[i][j]=susjedni[i][k]+susjedni[k][j];

    /* Trazim najpovoljniji grad za sastanak */
    minput=NO;
    for (i=0; i<nregija; i++)
    {
        tmp=cijena(i);
        if (minput==NO || tmp<minput)
        {
            minput=tmp;
            sol=i;
        }
    }
}

void ispisi_rjesenje (void)
{
    FILE *output;

    output=fopen("walls.out", "w");
    fprintf(output, "%d\n", minput);
    fprintf(output, "%d\n", sol+1);
}

int main (void)
{
    citaj_ulaz();
    rijesi();
    ispisi_rjesenje();
    return 0;
}
```

Building with Blocks

U ovom zadatku je potrebno jedan trodimenzionalni objekt izgraditi od minimalnog broja blokova unaprijed zadanog oblika. Prilikom izgradnje objekta, blokove možemo rotirati i translatirati.

Ovaj zadatak možemo riješiti jedino tako da rekurzivno ispunjavamo zadani objekt blokovima tj. da isprobamo sva moguća popunjavanja. Pritom koristimo takozvanu *branch and bound* tehniku, drugim riječima, prekidamo rekurziju kada više ne možemo dostići do sada najbolje rješenje.

Postavlja se pitanje kako riješiti problem rotacije blokova. Postoje dva različita pristupa: možemo unutar same rekurzije rotirati zadani blok ili možemo prethodno rotirati zadane blokove na sve moguće načine te izgraditi kolekciju svih neekvivalentnih blokova. Ovdje smo se odlučili na drugi pristup radi jednostavnosti implementacije.

Pozabavimo se najprije blokovima. U zadatku su blokovi zadani pomoću trodimenzionalnih koordinata kocki od kojih se sastoje. Očito je da za pojedini blok postoji više različitih prikaza, pa stoga uvodimo sljedeća pravila pomoću kojih će svaki blok imat jedinstven prikaz kao niz koordinata:

- Kocke od kojih se blok sadrži sortiramo leksikografski tj. najprije po prvoj koordinati, pa po drugoj, pa po trećoj.
- Blok sa sortiranim kockama translatiramo tako da prva kocka ima koordinate (0, 0, 0).

Ovaj postupak nazivamo *normalizacija*. Prvi kocku u normaliziranom bloku ćemo zvati *sidro* bloka. Na primjer, normalizacijom bloka zadanog sa koordinatama kocaka (1, 1, 1), (1, 2, 1), (1, 1, 2) dobivamo koordinate kocaka redom (0, 0, 0), (0, 0, 1), (0, 1, 0).

Općenito se svaki blok može rotirati na 24 različita načina, što je previše da bi mogli ručno pobrojati sve moguće rotacije pa primjenjujemo sljedeću taktiku: Fiksirajmo dvije rotacije, npr. rotacija oko x osi za 90 stupnjeva (x→x, y→-z, z→y) i rotacija oko y osi za 90 stupnjeva (x→z, y→y, z→-x) te primjetimo da svaku rotaciju možemo dobiti kao nekoliko uzastopnih primjena ove dvije. Sada znamo kako ćemo generirati sve blokove:

- Učitamo zadane blokove iz ulazne datoteke i ubacimo ih u kolekciju.
- Svaki blok iz kolekcije rotiramo oko x osi i u kolekciju ubacimo nove blokove.
- Svaki blok iz kolekcije rotiramo oko y osi i u kolekciju ubacimo nove blokove.
- Zadnja dva koraka ponavljano sve dok ne dobijemo niti jedan novi blok.

Prilikom ubacivanja blokova u kolekciju vršimo normalizaciju pa je jednostavno provjeriti da li su dva bloka ekvivalentna.

Opišimo sada samu rekurziju. Prije nego što počinjemo blokove rekurzivno slagati u objekt, sortiramo kocke od kojih se objekt sadrži leksikografski. Sada možemo primjeniti sljedeću taktiku.

- Nađemo prvu slobodnu kocku u objektu, ona je leksikografski manja od svih ostalih pa se točno na njoj mora nalaziti sidro nekog bloka.
- Isprobamo sve blokove postaviti na objekt tako da se sidro nalazi na tom slobodnom mjestu, te kada neki blok stavimo rekurzivno nastavljamo dalje.

Ostaje još jedimo odrediti uvjet za prekid rekurzije. Ako smo do sada objekt djelomično popunili sa **N** blokova i ako je nepopunjeni volumen jednak **V** očito je da će nam za popunjavanje biti potrebno barem $\lceil N+V/4 \rceil$ (zato što je volumen pojedinog bloka najviše 4). Dakle, ako je ovaj broj veći ili jednak od trenutno nađenog minimuma prekidamo rekurziju.

Vremenska složenost ovog algoritma je eksponencijalna, međutim ako popunjavanje vršimo na gore navedeni način te ako rekurziju prekidamo kada nije više moguće postići bolje rješenje, algoritam je dovoljno efikasan da u danom vremenskom ograničenju riješi sve ulaze sa zadanom veličinom.

Kako je izvorni kod prilično dug, ovdje smo ga , radi veće preglednosti, podijelili u više datoteka. U datoteci "**gener.c**" se nalaze funkcije za izgradnju kolekcije svih blokova, dok se funkcije za rješavanje samog problema nalaze u datoteci "**block.c**". Zajedničke definicije nalaze se u datoteci "**block.h**".

BLOCK.H

```
/* ---- block.h ---- */
#define BLOCKVOL    4
#define DIM         3
#define SOLIDVOL    50
#define NTYPES      12
#define MAXD        7
#define NO          -1
#define MAXBLOCKS   (NTYPES*24)
#define INFILE      "block.in"
#define OUTFILE     "block.out"
#define TYPEFILE    "types.in"
#define DUMPFILE    "all.in"
```

```
typedef int cube[DIM];

typedef struct
{
    int type;
    int vol;
    cube part[BLOCKVOL];
} block;
```

GENER.C

```
/* ---- gener.c ---- */
#include <stdio.h>
#include "block.h"

int ntypes=NTYPES;

extern int nblocks;
extern block blocks[MAXBLOCKS];

/* Da li je kocka a leksikografski manja od kocke b */
int less (cube a, cube b)
{
    int i;

    for (i=0; i<DIM; i++)
        if (a[i]<b[i])
           return 1;
        else
        if (a[i]>b[i])
           return 0;
    return 0;
}

/* Sortira kocke leksikografski i normalizira po koordinatama prve */
void normalize (block *b)
{
    int i, j, k;
    int tmp;

    for (i=0; i<b->vol-1; i++)
        for (j=i+1; j<b->vol; j++)
            if (less(b->part[j], b->part[i]))
                for (k=0; k<DIM; k++)
                {
                    tmp=b->part[i][k];
                    b->part[i][k]=b->part[j][k];
                    b->part[j][k]=tmp;
                }

    for (i=b->vol-1; i>=0; i--)
        for (j=0; j<DIM; j++)
            b->part[i][j]-=b->part[0][j];
}

/* Da li su normalizirani blokovi a i b jednaki */
int equal (block *a, block *b)
{
    int i, j;

    if (a->vol!=b->vol)
       return 0;

    for (i=0; i<a->vol; i++)
        for (j=0; j<DIM; j++)
            if (a->part[i][j]!=b->part[i][j])
               return 0;
    return 1;
}

/* Dodaje blok u koleciju */
void dodaj (block *b)
{
```

```
    int i;

    for (i=0; i<nblocks; i++)
        if (equal(&blocks[i], b))
           return ;

    blocks[nblocks]=*b;
    nblocks++;
}

/* Ucitava blokove iz datoteke "types.in" */
void ucitaj_tipove (void)
{
    int i, j, k;
    FILE *input;
    block b;

    nblocks=0;
    input=fopen(TYPEFILE, "r");
    for (i=0; i<ntypes; i++)
    {
        fscanf(input, "%d", &b.type);
        fscanf(input, "%d", &b.vol);
        for (j=0; j<b.vol; j++)
            for (k=0; k<DIM; k++)
                fscanf(input, "%d", &b.part[j][k]);
        normalize(&b);
        dodaj(&b);
    }
}

/* Debug */
void ispisi_blokove (void)
{
    int i, j, k;
    FILE *output;
    int nt[NTYPES];

    output=fopen(DUMPFILE , "w");
    fprintf(output, "nblocks = %d\n", nblocks);
    for (i=0; i<nblocks; i++)
    {
        fprintf(output, "Block %d:\n", i+1);
        fprintf(output, "\tType = %d\n", blocks[i].type);
        fprintf(output, "\tVol  = %d\n", blocks[i].vol);
        for (j=0; j<blocks[i].vol; j++)
        {
            fprintf(output, "\t\t");
            for (k=0; k<DIM; k++)
                fprintf(output, "%d ", blocks[i].part[j][k]);
                fprintf(output, "\n");
        }
    }

    for (i=0; i<ntypes; i++)
        nt[i]=0;
    for (i=0; i<nblocks; i++)
        nt[blocks[i].type-1]++;
    for (i=0; i<ntypes; i++)
        fprintf(output, "\tType %d - %d pieces\n", i+1, nt[i]);

    fclose(output);
}

/* Rotira kocku oko x osi za 90 stupnjeva: x -> x, y -> -z, z -> y */
void rotx (cube c)
{
    int x=c[0];
    int y=c[1];
    int z=c[2];

    c[0]=x;
    c[1]=z;
    c[2]=-y;
```

```
}

/* Rotira kocku oko y osi za 90 stupnjeva: x -> z, y -> y, z -> -x */
void roty (cube c)
{
    int x=c[0];
    int y=c[1];
    int z=c[2];

    c[0]=-z;
    c[1]=y;
    c[2]=x;
}

/* Rotira blokove na sve moguce nacine i ubacuje dobiveno u kolekciju */
void rotiraj (void)
{
    int i, j, k;
    int nb;
    block nova;

    nb=0;
    while (nb!=nblocks)
    {
        nb=nblocks;
        for (k=0; k<nblocks; k++)
        {
            nova=blocks[k];
            for (i=0; i<nova.vol; i++)
                rotx(nova.part[i]);
            normalize(&nova);
            dodaj(&nova);
            nova=blocks[k];
            for (i=0; i<nova.vol; i++)
                roty(nova.part[i]);
            normalize(&nova);
            dodaj(&nova);
        }
    }
}

void gener (void)
{
    int i;

    ucitaj_tipove();
    rotiraj();
    ispisi_blokove();

    return ;
}
```

BLOCK.C

```
/* ---- block.c ---- */
/* Kompajliraj sa: gcc -o block.exe block.c gener.c */
#include <stdio.h>
#include "block.h"

#define SWAP(A,B,C) { (C)=(A); (A)=(B); (B)=(C); }

int nblocks;
block blocks[MAXBLOCKS];

int volume;
cube solid[SOLIDVOL];
int index[MAXD][MAXD][MAXD];

int volused;
int used[SOLIDVOL];

int min;
int sol[SOLIDVOL];
```

```
int current;
int parts[SOLIDVOL];

void citaj_ulaz (void)
{
    int i, j, k;
    FILE *input;

    for (i=0; i<MAXD; i++)
        for (j=0; j<MAXD; j++)
            for (k=0; k<MAXD; k++)
                index[i][j][k]=NO;

    input=fopen(INFILE, "r");
    fscanf(input, "%d", &volume);
    for (i=0; i<volume; i++)
    {
        for (j=0; j<DIM; j++)
        {
            fscanf(input, "%d", &solid[i][j]);
            solid[i][j]--;
        }
        index[solid[i][0]][solid[i][1]][solid[i][2]]=i;
    }
}

/* Da li je pozicija x, y, z slobodna */
int valid (int x, int y, int z)
{
    int i;

    if (x<0 || x>=MAXD) return 0;
    if (y<0 || y>=MAXD) return 0;
    if (z<0 || z>=MAXD) return 0;
    if (index[x][y][z]==NO) return 0;
    if (used[index[x][y][z]]) return 0;
    return 1;
}


/* Da li blok b moze stati na poziciju x, y, z */
int moze (block *b, cube c)
{
    int i;

    for (i=0; i<b->vol; i++)
        if (!valid(c[0]+b->part[i][0], c[1]+b->part[i][1], c[2]+b->part[i][2]))
           return 0;
    return 1;
}

void stavi (block *b, cube c)
{
    int i;

    for (i=0; i<b->vol; i++)
        used[index[c[0]+b->part[i][0]][c[1]+b->part[i][1]][c[2]+b->part[i][2]]]=1;
}

void makni (block *b, cube c)
{
    int i;

    for (i=0; i<b->vol; i++)
        used[index[c[0]+b->part[i][0]][c[1]+b->part[i][1]][c[2]+b->part[i][2]]]=0;
}

/* Sortira blokove po volumenu silazno, te sortira kocke u solidu leksikogr. */
void sort (void)
{
    int i, j, k;
    int tmpi;
    block tmpb;
```

```
    for (i=0; i<nblocks-1; i++)
        for (j=i+1; j<nblocks; j++)
            if (blocks[i].vol<blocks[j].vol)
               SWAP(blocks[i], blocks[j], tmpb);

    for (i=0; i<volume-1; i++)
        for (j=i+1; j<volume; j++)
            if (less(solid[j], solid[i]))
            {
                SWAP(index[solid[i][0]][solid[i][1]][solid[i][2]],
                     index[solid[j][0]][solid[j][1]][solid[j][2]],tmpi);
                for (k=0; k<DIM; k++)
                    SWAP(solid[i][k], solid[j][k], tmpi);
            }

}

void rijesi (int fcube)
{
    int i, j;

    /* Ako smo popunili cijeli lik */
    if (volused==volume)
    {
        if (current<min)
        {
            min=current;
            for (i=0; i<min; i++)
                sol[i]=parts[i];
        }
        return ;
    }
    /* Optimizacija */
    if ((volume-volused-1)/BLOCKVOL+1+current>=min) return ;

    /* Na sljedece prazno mjesto isprobavam staviti sve blokove */
    i=fcube;
    while (used[i]) i++;
    for (j=0; j<nblocks; j++)
        if (moze(&blocks[j], solid[i]))
           {
                stavi(&blocks[j], solid[i]);
                parts[current]=j;
                current++;
                volused+=blocks[j].vol;
                rijesi(i+1);
                volused-=blocks[j].vol;
                current--;
                makni(&blocks[j], solid[i]);
           }
}

void ispisi_rjesenje (void)
{
    int i;
    FILE *output;

    output=fopen(OUTFILE, "w");
    fprintf(output, "%d\n", min);
    for (i=0; i<min; i++)
        fprintf(output, "%d ", blocks[sol[i]].type);
    fprintf(output, "\n");
    fclose(output);
}

int main (void)
{
    int i;

    /*
       Pozivam funkciju gener() iz datoteke gener.c koja, ucitava zadane
       tipove, rotira ih, te u kolekciju blocks spremi sve neekvivalentne
       blokove
```

```
    */
    gener();
    citaj_ulaz();

    volused=0;
    for (i=0; i<volume; i++)
        used[i]=0;
    min=volume+1;
    current=0;

    sort();
    rijesi(0);

    ispisi_rjesenje();

    return 0;
}
```