Dinamičko programiranje - zadaci

Dinamičko programiranje (DP) je nastalo kao način rešavanja jedne klase algoritamskih problema, u kojima se traži optimalno rešenje, odnosno rešenje koje maksimizira ili minimizira neku zadatu veličinu. Kasnije se naziv preneo na širu klasu problema koji se rešavaju u osnovi istom idejom, a to je rešavanje manjih problema istog tipa i njihovo kombinovanje radi dobijanja rešenja polaznog, većeg problema.

DP je tehnika oblikovanja rešenja koja rastavlja složen problem na manje složene potprobleme/korake.

Za DP važno je usvojiti specifičan način razmišljanja pomoću kog treba:
1. pronaći sva stanja
Pri rešavanju probleme upotrebom DP-a, potrebno je rastaviti problem na **stanja**. Stanje je rešenje nekog malog dela problema.
2. Na temelju prethodno izračunatih stanja, izračunati novo stanje (moramo znati relaciju)
3. Dinamike (algoritmi koji koriste DP) izračunavaju stanja takvim redom, tako da se niti jedno stanje ne izračunava dva puta.
4. Poznavati početna stanja

Sama reč programiranje u terminu dinamičko programiranje se kao i u linearnom programiranju odnosi na popunjavanje tabele pri rešavanju problema, a ne na upotrebu programskog jezika ili računara. Tehnike optimizacije koje koriste dinamičko programiranje bile su poznate i pre II svetskog rata, ali se tvorcem metode smtra profesor Richard Belman, koji je sredinom pedesetih godina proučavao problem tako što je proučavao hijerarhiju podproblema sadržanih u glavnom problemu i rešavanje počinjao od najjednostavnijih.
Algoritmi iz prakse koji koriste dinamičko programiranje

- Cocke-Younger-Kasami (CYK) algoritam - određuje da li i kako zadati string se može generisati iz zadate kontekst-slobodne gramatike
- transpoziciona tabela za igranje šaha na računaru, računanje pozicija premeštanja i napada
- Viterbi algoritam, nalaženje najverovatnije sekvence za neko skriveno stanje, npr. šema za korekciju grešaka u digitalnim komunikacionim linijama u kojima se pojavljuju i šumovi;
- Floyd algoritam za nalaženje najkraćih puteva u grafu
-

Pomoću algoritama dinamičkog programiranja, potproblemi jednog problema postaju međusobno zavisni, pa je dovoljno da ih samo jednom rešite, da bi Vaš rad bio lakši i brži.
Postoji velika klasa problema koje je moguće rešiti tako što se polazni problem podeli na potprobleme, a oni dalje na pod-pod-probleme i sve tako dok se ne stigne do najjednostavnijih problema, koje se ne moraju dalje deliti i koji se odmah rešavaju.
Divide-and-conquer je jedan od metoda koji "radi" na ovom principu (*Quick sort*).

Realno je očekivati da postoje problemi kod kojih se dešava da prilikom deobe na potprobleme se neke pojave više puta. Ukoliko bi svaki bio rešavan onog trenutka kada se na njega naiđe, ne vodeći računa o tome da li je već jednom rešen, moglo bi se desiti da isti posao rađen više puta, što može rezultirati katastrofalno sporim algoritmom.

Osnovna ideja na koju se oslanjaju algoritmi dinamičkog programiranja je da se svaki dobijeni međurezultat (rešenje nekog potproblema) sačuva i potom, kada se sledeći put naiđe na taj isti potproblem, izbegne njegovo ponovno rešavanje. Već sada se uočava bitna razlika između dinamičkog programiranja i *divide-and-conquer metoda*: kod ovog drugog potproblemi su međusobno nezavisni, dok kod dinamičkog programiranja nisu.

1. Formirati algoritam koji za dve reči dužine n (prva reč), m (druga reč) nad azbukom slova i cifara pronalazi niz operacija editovanja najmanje cene kojim se prva reč transformiše u drugu reč. Dozvoljene su tri vrste operacija editovanja i svaka ima cenu jedan:

- insert = umetanje jednog karaktera
- delete = brisanje 1 karaktera
- replace = zamena jednog karaktera drugim karakterom

Primer za A=maak, B=amaa
A(1)=amaak /*insert slovo a u reč A*/
A(2)=amaa /*delete slovo k iz reči A*/
CENA: 1+1=2

A(1)=aak /*delete slovo m iz reči A*/
A(2)=amak /*insert slovo m u reči A*/
A(3)=amaa /*replace slovo k slovom a u reči A */
CENA: 1+1+1=3

Operacija editovanja najmanje cene zapravo predstavlja Levenshtein ili edit rastojanje dve niske. Nazvana je po ruskom naučniku (Vladimir Levenshtein) koji se 1965. bavio ovim rastojanjem.

Koristi se u aplikacijama u kojima je potrebno znati koliko su slične dve niske, na primer:

1. **spell checker-i (provera pravopisa)**

Spell checkeri porede reč sa ulaza sa jedinicom rečnika. Ako reč nije pronađena u rečniku, smatra se da nije korektno uneta i izlistavaju se moguće reči iz rečnika, počev od onih sa najmanjom Levenshtein udaljenosti od ulazne reči.

2. **aplikacije molekularne biologije (DNK analiza; što je veći stepen sličnosti dve sekvence, bilo DNK, RNK ili amino-kiselina, to im je sličnija i funkcija ili struktura, npr. kod multiplskleroze koja to bakterijska ili virusna sekvenca iz ranije infekcije je slična sekvenci proteinske mijelinske ovojnice)**
3. **detekcija plagijata**
4. **prepoznavanje govora**

Što je Levenštajnovo rastojanje dve niske veće, to se one više razlikuju među sobom.

Procena sličnosti dve niske može se utvrditi na osnovu Levenštajnovog rastojanja ili LCS algoritmom koji pronalazi najdužu zajedničku podsekvencu (podsekvenca se dobija iz niske brisanjem 0 ili više karaktera). Oba algoritma će biti navedena kao ilustracija tehnike dinamičkog programiranja.

Inače, današnje softverske alatke koriste više metrika, a često i njihovu kombinaciju. Neke metrike su pogodnije za oblast molekularne biologije, neke za oblast text mining-a.

Oznake: Neka je A(k) niz koji se sastoji od prvih k elemenata niza A. Neka je B(k) niz koji se sastoji od prvih k elemenata niza B.

Nizovi edit operacija koji transformišu A(i) u B(j) zavisno od poslednje operacije u nizu edit operacija
Postoje tri ključne situacije:

1. ako je A(i-1) transformisan u B(j), onda se obavlja delete znaka a[i]
2. ako je A(i) transformisan u B(j-1), onda se obavlja insert znaka b[j]
3. ako je A(i-1) transformisan u B(j-1), onda se obavlja replace znaka a[i] (istim ili različitim) znakom b[j]

Dakle, najmanja cena transformacije A(i) u B(j), tj. Cena[i,j] jednaka je Cena[i,j]=Min_3_broja (c1, c2, c3), gde

- **c1 - slučaj prethodnog delete a[i], c1= Cena[i-1,j]+1**
- **c2 - slučaj prethodnog insert b[j], c2= Cena[i,j-1]+1**
- **c3 - slučaj prethodnog replace a[i] sa b[j], c3= Cena[i-1,j-1]+ (a[i]==b[j])**

pseudo-jezik:
Algoritam NajjeftinijeEditovanje (a, b, n, m)
ulaz: a, n, b, m
izlaz: Cena /* tabela (n+1)*(m+1) */

{

/* i, j su lokalne promenljive i imaju ulogu brojača u petljama
Potez je matrica koju pregledamo unazad (počev od Potez[n,m]), ne bi li dobili konačan niz operacija editovanja.
Potez[i,j] -poslednja operacija (delete ili insert ili replace) editovanja koja se obavlja da bi Cena[i,j] bila minimalna

```
c1 - slučaj prethodnog delete a[i]
c2 - slučaj prethodnog insert b[j]
c3 - slučaj prethodnog replace a[i] sa b[j]
*/

/* transformacija prefiksa A(i) u praznu nisku B(0) */
for (i=0; i <= n; i++)
{
   Cena[i,0]=i;
   Potez[i,0]= delete;
}

/* transformacija praznog prefiksa A(0) u prefiks B(j) */
for (j=0; j <= m; j++)
{
   Cena[0,j]=j;
   Potez[0,j]= insert;
}

/* i!=0, j!=0 i najjeftinija transformacija A(i) u B(j) */
for (i=1; i <= n; i++)
 for (j=1; j <= m; j++)
  {
    c1= Cena[i-1,j]+1;
    c2= Cena[i,j-1]+1;
   if (a[i]==b[j])
     c3= Cena[i-1,j-1];
    else
      c3= Cena[i-1,j-1]+1;
  Cena[i,j]=Min_3_broja (c1, c2, c3);
   Potez[i,j] = izabrana_operacija_editovanja_u_Min_3_broja;
  } /*for */
} /*KRAJ*/
```

Procena složenosti
Svaki element matrice Cena se i izračunava za konstantno vreme, pa je vremenska složenost algoritma *O(mn)*, dok je i prostorna složenost *O(mn)*. Može se postići prostorna složenost reda *O(m)*, ali ako se matrica Cena popunjava po vrstama (jer jedna vrsta zavisi samo od prethodne vrste).

Primer
Primer formiranja matrice cena za niske "SAMBA" i "SIMBAD".

Koraci 1, 2

		S	A	M	B	A
	0	1	2	3	4	5
S	1					
I	2					
M	3					
B	4					
A	5					
D	6					

Koraci od 3 do 6 kada i = 1

		S	A	M	B	A
	0	1	2	3	4	5
S	1	0				
I	2	1				
M	3	2				
B	4	3				
A	5	4				
D	6	5				

Koraci od 3 do 6 kada i = 2

		S	A	M	B	A
	0	1	2	3	4	5
S	1	0	1			
I	2	1	1			
M	3	2	2			
B	4	3	3			
A	5	4	3			
D	6	5	4			

Koraci od 3 do 6 kada i = 3

		S	A	M	B	A
	0	1	2	3	4	5
S	1	0	1	2		
I	2	1	1	2		
M	3	2	2	1		
B	4	3	3	2		
A	5	4	3	3		
D	6	5	4	4		

Koraci od 3 do 6 kada i = 4

		S	A	M	B	A
	0	1	2	3	4	5
S	1	0	1	2	3	
I	2	1	1	2	3	
M	3	2	2	1	2	
B	4	3	3	2	1	
A	5	4	3	3	2	
D	6	5	4	4	3	

Koraci od 3 do 6 kada i = 5

		S	A	M	B	A

	0	1	2	3	4	5
S	1	**0**	1	2	3	4
I	2	1	**1**	2	3	4
M	3	2	2	**1**	2	3
B	4	3	3	2	**1**	2
A	5	4	3	3	2	**1**
D	6	5	4	4	3	**2**

Rastojanje je u donjoj desnoj ćeliji, tj. 2, što odgovara i inuitivnoj realizaciji da se niska "SAMBA" može transformisati u nisku "SIMBAD" zamenom slova "A" slovom "I" i dodavanjem slova "D" (1 zamena i 1 umetanje = 2 operacije).

Java implementacija

```
        public class Distance {
//*****************************
// minimum tri vrednosti a, b, c
//*****************************
private int Minimum (int a, int b, int c) {
int mi;
  mi = a;
  if (b < mi) {
    mi = b;
  }
  if (c < mi) {
    mi = c;
  }
  return mi;
}
//*****************************
// Izracunavanje Levenshtein rastojanja
//*****************************
public int LD (String s, String t) {
int d[][]; // matrica
int n; // duzina za s
int m; // duzona za  t
int i; // indeks niza s
int j; // indeks niza t
char s_i; // i-ti karakter za s
char t_j; // j-ti karakter za  t
int cena; // cena
  // Korak 1
  n = s.length ();
  m = t.length ();
  if (n == 0) {
    return m;
  }
  if (m == 0) {
    return n;
  }
  d = new int[n+1][m+1];

  // Korak 2
  for (i = 0; i <= n; i++) {
```

```
    d[i][0] = i;
   }

   for (j = 0; j <= m; j++) {
    d[0][j] = j;
   }

   // Korak 3
   for (i = 1; i <= n; i++) {
    s_i = s.charAt (i - 1);
    // Korak 4
    for (j = 1; j <= m; j++) {
     t_j = t.charAt (j - 1);
     // Korak 5
     if (s_i == t_j) {
      cena = 0;
     }
     else {
      cena = 1;
     }

     // Korak 6
     d[i][j] = Minimum (d[i-1][j]+1, d[i][j-1]+1, d[i-1][j-1] + cena);
    }
   }
   // Korak 7
   return d[n][m];
  }
}
```

Domaći

A=aabccbba, B=baacbabacc, Cena=?

2.

Napisati rekurzivnu funkciju koja računa n-ti element u Fibonačijevom nizu. Popraviti funkciju tako da se problemi manje dimenzije rešavaju samo jedanput. (dinamičko programiranje).

Analiza rešenja

U slučaju da se rekurzijom problem svodi na više manjih podproblema koji se mogu preklapati, postoji opasnost da se pojedini podproblemi manjih dimenzija rešavaju veći broj puta. Na primer,

fibonacci(20) = fibonacci(19) + fibonacci(18)
fibonacci(19) = fibonacci(18) + fibonacci(17)
tj. problem fibonacci(18) se resava dva puta. Problemi manjih dimenzija će se rešavati još veći broj puta.

Rešenje za ovaj problem je kombinacija rekurzije sa tzv. "dinamičkim programiranjem" tj.
podproblemi se rešavaju samo jednom, a njihova rešenja se pamte u memoriji (obično u nizovima ili matricama), odakle se koriste ako tokom rešavanja ponovo budu potrebni.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 50
/* Niz koji cuva resenja podproblema. */
int f[MAX];

/* Funkcija izracunava n-ti fibonacijev broj */
int fibonacci(int n)
```

```
{
/* Ako je podproblem vec resen, uzimamo gotovo resenje! */
if(f[n] != 0)
return f[n];

/* Izlaz iz rekurzije */
if(n < 2)
return f[n] = 1;

else
/* Rekurzivni pozivi */
return f[n] = fibonacci(n - 1) + fibonacci(n - 2);
}

/* Test program */
int main()
{
int n, i;
/*Inicijalizuje se niz*/
for(i=0; i<MAX; i++)
f[i] = 0;
scanf("%d", &n);
printf("%d\n", fibonacci(n));
return 0; }
```

3. Programer Vasa radi za tajnu organizaciju i mora uočiti pravilnost između dva niza podataka. Cilj mu je da za dva niza brojeva pronađe dužinu najdužeg (ne obavezno povezanog) podniza koji se pojavljuju u oba niza. Ulazni podaci su n i m (1<=n,m<=1000), koji redom predstavljaju dužine oba niza brojeva. Nakon toga sledi n celih brojeva i zatim m celih brojeva. Ispisati dužinu najdužeg zajedničkog podniza.

ULAZ	IZLAZ
10 9	5
3 2 4 1 5 8 7 3 6 9	
2 7 5 1 7 3 4 2 6	

Napomena: Najduži zajednički podniz (LCS, longest common subsequence) je 2 1 7 3 6 ili 2 5 7 3 6.

Šta je stanje u ovom DP problemu? Nazovimo nizove redom A, B.
Stanje DP[x][y] definišemo kao duzinu LCS u prvih x+1 elemenata prvog niza A i prvih y+1 elemenata niza B.

Šta je relacija u ovom DP problemu?
Jednostavno dizajniramo DP algoritama koristeci induktivni pristup, tj. pretpostavimo da smo rešili potproblem za prefiks niza A dužine x-1 i prefiks niza B dužine y-1:

Ako A[x]!=B[y], onda: DP[x][y]=max{DP[x][y-1], DP[x-1][y] }
Novo stanje mozemo izracunati prosirujuci pre izracunato stanje po x ili po y.

Ako A[x]=B[y], onda: DP[x][y]=max{DP[x][y-1], DP[x-1][y], DP[x-1][y-1]+1 }
Ako su elementi A[x], B[y] jednaki, mozemo prosiriti stanje DP[x-1][y-1] i dodati 1, jer LCS ce biti produzen za jos jedan clan tj. A[x] (ili B[x]).

Resenje

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

int main()
```

```
{  int n,m;
   cin>>n>>m;
   vector<int> A(n), B(m,0);
   vector<vector<int> > DP(n,B); //inicijalna stanja u resenju

   for(int i=0;i<n;i++) cin >>A[i];
   for(int i=0;i<m;i++) cin >>B[i];

   for(int i=0;i<n;i++)
   for(int j=0;j<m;j++) {
     if (i) DP[i][j]=max(DP[i][j], DP[i-1][j]);
     if (j) DP[i][j]=max(DP[i][j], DP[i][j-1]);
     if (A[i]==B[j])
      DP[i][j]=max(DP[i][j], ((i&&j)?DP[i-1][j-1]:0)+1);
   }

    cout << DP[n-1][m-1] << endl;
    return 0;
}
```

4. Srbija je ,nekada bila bogata šumama, te se radi zaštite životne sredine pokušava na svaki način sprečiti nesavesna seča drveća. Lokalni bogataš, kupio je veliko kvadratno parče zemlje i na njemu želi da izgradi što je veće moguću kuću i to tako da i njena osnova bude kvadratnog oblika sa stranama paralelnim stranama placa. Mapa imanja je sastavljena iz kvadratića dimenzija 1x1. Jedno drvo zauzima tačno jedan takav kvadratić. Pošto je imanje veliko, vrlo je teško naći najveći kvadrat sa stranama paralelnim stranama placa na kome nema drveća. Vaš zadatak je da za zadatu mapu imanja na kojoj se nalaze pozicije drveća, nađete kvadrat čija je strana najduža na kome nema stabala. Dužina strane je broj kvadratica 1x1 duž njegove strane.

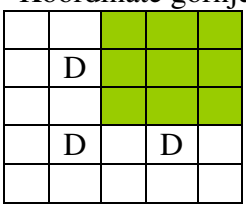
Sa standardnog ulaza se učitava sledeće: U prvom redu broj n (n <= 200) koji je dužina strane imanja. Sledeći red sadrži broj p, broj stabala na imanju. Narednih p redova sadrže po dva broja koji su koordinate stabala. Na standardni izlaz treba ispisati samo jedan broj, dužina stranice najvećeg kvadrata koji ne sadrži ni jedno drvo.

Primer

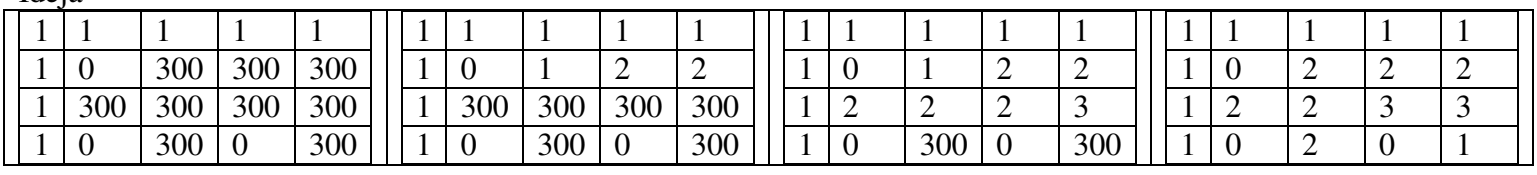
Ulaz:
5
3
2 2
4 2
4 4

Izlaz:
3

Koordinate gornjeg levog temena traženog kvadrata su 1 3, a donjeg desnog 3 5.



Ideja



1	300	300	300	300		1	300	300	300	300		1	300	300	300	300		1	300	300	300	300

Rešenje :

```
#include <iostream>
#include <cstdio>
#include <cstring>

using namespace std;
#define MaxN  200
#define BLA 300
int a[MaxN+1][MaxN+1];

int min3(int x, int y, int z)
{   int min=x;
    if (y<min) min=y;
    if (z<min) min=z;
    return min;
}

int main()
{ int i, j, p, n, X, Y, Max1=1;
    cin >> n >> p;

  for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)
   {   if ((i==1) || (j==1)) a[i][j]=1;
       else
       a[i][j]=BLA;
   }

  for(i=1;i<=p;i++)
  {
     cin >>X>>Y;
     a[X][Y]=0;
  }

  for(i=2;i<=n;i++)
  for(j=2;j<=n;j++)
       if ( a[i][j]>0)
     {
       a[i][j]=1+min3(a[i-1][j-1],a[i-1][j],a[i][j-1]);
       if(a[i][j]>Max1) Max1=a[i][j];
    }
 cout<<Max1<<endl;

    return 0;
}
```

II način

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

int main (){
   int n, p;
   cin >> n >> p;
   int pos[p][2];
   for (int i = 0; i < p; ++i){
      cin >> pos[i][0] >> pos[i][1];
   }
   int polje[n][n];
   memset (polje, 0, sizeof (polje));
   for (int i = 0; i < p; ++i){
      polje[pos[i][0] - 1][pos[i][1] - 1] = 1;
   }
   bool test = true;
   for (int i = n; i > 0; --i){
      for (int x = 0; x < n - i + 1; ++x){
         for (int y = 0; y < n - i + 1; ++y){
            test = true;
            for (int x2 = 0; x2 < i; ++x2){
               for (int y2 = 0; y2 < i; ++y2){
                  if (polje[x2 + x][y2 + y] == 1){test = false; break;}
               }
               if (!test){break;}
            }
            if (test){cout << i; return 0;}
         }
      }
   }
   cout << 0;
}
```

5. Dat je niz celih brojeva **a** sa **n** elemenata. Odrediti podniz niza **a** čija je suma elemenata maksimalna, a u kome nema susednih elemenata niza **a**. Smatrati da je suma elemenata praznog niza jednaka 0.

TEST PRIMERI

ULAZ(a)	**IZLAZ(podniz)**
5 2 -1 8 10 -2 -5	5 10
-1 -2 -4 3 1 -2 1	3 1
-3 4 -5 -4 1 18 12	4 18

REŠENJE: Neka je a niz od n celih brojeva.
Neka je R=R(A) jedan podniz iz formulacije zadatka koji odgovara nizu a.
Neka je A_k niz koji se sastoji od prvih k elemenata niza a.
Podniz niza A_0 je prazan niz, dok je podniz niza A_1 element a[1] ako je a[1]>0.
1. Ako element a[n] ne pripada podnizu R=R(A), onda je R
podniz iz formulacije zadatka koji odgovara nizu A_{n-1}.
2. Ako element a[n] pripada podnizu R=R(A), onda je R \ a[n]
podniz iz formulacije zadatka koji odgovara nizu A_{n-2}.

Dakle, za i=2..n, važi da $R(A_i)$ je ili $R(A_{i-1})$ ili $R(A_{i-2})$* {a[i]}, tj.
uzima se onaj od podnizova koji ima veci zbir.

Neka je niz s takav da:
s(i) je suma podniza $R(A_i)$.

Iz gore izlozenog, jasno je da:
s(0)=0,
s(1)=max(0, a[1]),
s(i)=max{ s(i-1), s(i-2) + a[i] }, i=2..n

Iz niza s će se ispisati rešenje R.

```
Algoritam MSN (a,n)
ulaz: a, n   /* niz a duzine n, BSO clanovi su a[1]...a[n] */
izlaz: podniz niza a cija suma je maksimalana,
gde podniz ne sadrzi uzastopne elemente iz a
{
  s[0]=0;        /* s[j] = suma elemenata podniza b koji je resenja zadatka za podniz A(j) */
                 /* s[0]=0, jer suma praznog niza je 0, po dogovoru */
if (a[1] > 0)  s[1]=a[1];
              else s[1]=0;


for (i=2;i <= n; i++)  /*uzimamo da indeks prvog clana niza je 1, drugog je 2,...*/

     if  (s[i-2] + a[i]  > s[i-1])
           s[i]= s[i-2] + a[i];
      else s[i]= s[i-1];

MSN_ispis (n);
}


procedure  MSN_ispis (n)
ulaz: n
izlaz: ispis clanova podniza koji sadrzi nesusedne clanove niza,
        ali tako da je suma podniza maksimalna
{
if (n > 0)
   if (s[n] = = s[n-1)  MSN_ispis(n-1);
   else
      {  MSN_ispis(n-2);
         ispisati  a[n];   /*jer je clan niza b */
      }

}
```

6. Neka **n** ljudi čeka u redu da kupi karte za predstavu, pri cemu $\mathbf{t_i}$ je vreme koje je **i**-tom kupcu potrebno da kupi kartu. Ako se po dvoje suseda u redu udruzi da kupi karte - na primer **k**-ti i **k+1**-vi kupac-onda vreme potrebno da oni kupe karte je $\mathbf{p_k}$, k=1..n-1. Udruzivanjem kupaca moze da se ubrza kupovina karata, a i ne mora. Ulazni podaci su broj kupaca **n** i nizovi **t** i **p**. Konstruisati algoritam koji odredjuje takav nacin udruzivanja da ukupno vreme potrebno da svih **n** kupaca kupi kartu bude minimalno.

Resenje: Neka je niz **a** takav da **a[k]** je usteda u vremenu kupovine karata koja nastaje udruzivanjem **k**-tog i **k+1**-vog kupca, tj:
a[k]=t[k]+t[k+1]-p[k]
Jasno da clanovi niza **a** u opstem slucaju su i pozitivni i negativni celi brojevi.
Dalje, **k**-ti covek se ne moze istovremeno udruziti sa **k+1**-vim i **k-1**-vim covekom,
tj. odatle sledi da u nizu ne mogu istovremeno i a[k-1] i a[k] biti ustede,

tj.potrebno je pronaci podniz niza **a** koji ima najvecu sumu u kojoj ne ucestvuju susedni clanovi, a to je upravo problem koji je tema 3. zadatka.

Za vezbu: zadatak http://bee.bubblecup.org/Problems/BrzaHrana

7. U kvadratu dimenzija NxN, gde je N<13, polja su popunjena ciframa od 0 do 9. Ako su vrste i kolone kvadrata označene brojevima od 0 do N-1 napisati program kojim se ispisuje maršuta kojom se iz gornjeg levog polja (0,0) stiže do donjeg desnog polja (N-1,N-1) tako da su zadovoljeni sledeći uslovi:
1) sa svakog polja je dozvoljeno preći samo na polje ispod ili na polje desno od tog polja;
2) od svih maršuta, koje zadovoljavaju prethodni uslov ispisati onu čija je suma cifara, u poljima preko kojih se ide, maksimalna.

8.
U novoj eri elektronskog poslovanja, bankomati sve više postaju svakodnevna pojava. Banke žele da se klijenti što krace zadržavaju kod bankomata, kako bi opslužili što više ljudi, i zbog toga zahtevaju da se traženi iznos novca klijentima isplaćuje sa što manje novčanica. Medutim, bankomate su programirali lenji programeri koji su upotrebili najjednostavniji algoritam tako da se u svakom njegovom koraku bira najveća novcanica koja je manja ili jednaka od preostalog iznosa. Na primer, za sistem sa novčanicama od 1, 7 i 10 dinara, iznos od 14 dinara biće isplaćen pomoću jedne novčanice od 10 i 4 novčanice od 1 dinara, ukupno 5 novčanica. Ovo ne odgovara zahtevu banke, jer se 14 dinara može isplatiti pomoću samo dve novčanice od 7 dinara.

Vaš zadatak je da napišete program koji će za dati sistem novčanica, koji uvek sadrži i apoen od jednog dinara, naći najmanji iznos koji se može isplatiti u manje novčanica od broja novčanica koji bi bankomat iskoristio.

Ulaz: U prvom redu ulaza nalazi se prirodan broj n, broj novčanica u sistemu (1 <= n <= 50). U drugom redu ulaza nalaze se n različitih prirodnih brojeva razdvojenih razmakom, vrednosti novčanica u sistemu. Svaki od njih nije veći od 500000, a jedan od njih je 1.

Izlaz: U izlaz upisati jedan prirodan broj, najmanji iznos koji bankomati ne isplaćuju u najmanjem broju novčanica. Ako za dati sistem bankomati uvek isplaćuju iznos u najmanjem broju novčanica, napisiti 0.

Primer:

ulaz
3
1 7 10

izlaz
14

```
#include <iostream>
#include <algorithm>
#include <cmath>
#include <climits>

using namespace std;

#define MaxN  50
#define MaxMonete 500000

int DP[2*MaxMonete+1], greedy[2*MaxMonete+1];
int monete[MaxN];
int main()
{ int n;

  cin >> n;
```

```
  for (int i=0;i<n;i++) cin >> monete[i];

  sort(monete, monete+n);
//max velicina moneta je monete[n-1]
  for (int i=1; i<2*monete[n-1];i++)
  {
    DP[i]=INT_MAX; //tabela za dinamicko programiranje
    for (int j=0; j<n; j++)
     if (i>=monete[j] )
    { //paralelno popunjavamo tabelu pohlepnog
      //rasitnjavanja i rasitnjavanja din.prog.
      //za sve i koji su manji od tekuce monete[j]
      DP[i]=min(DP[i],DP[i-monete[j]]+1);
      greedy[i]=greedy[i-monete[j]]+1;
    }
    //ako se rasitnjavanje pohlepnim metodom
    //i din.prog. razlikuju, stampamo najmanji takav iznos
    if (DP[i]!=greedy[i])
    { cout<<i<<endl;
      return 0;
    }
  }
  cout<<"0"<<endl;
    return 0;
}
```

9. Za cele brojeve zadate trougaono kao na slici odrediti put koji obezbeđuje najveću sumu, a ide od vrha i završava se na osnovi trougla. Dozvoljeni pravac kretanja je nadole: dijagonalno levo i dijagonalno desno.

```
                7
            3       8
        8       1       0
    2       7       4       4
4       5       2       6       5
```

Ideja:
Trougaonu tabelu D ćemo spakovati u matricu sledećeg oblika:
7 0 0 0 0
3 8 0 0 0
8 1 0 0 0
2 7 4 4 0
4 5 2 6 5
i pravilo kretanja prevesti na: nadole pravo i nadole dijagonalno desno.

10. (POJ zadatak, 2009 ACM North Western European Regional Contest, Divisible Subsequences, Time Limit: 1000MS Memory Limit: 65536K)
Dat je niz prirodnih brojeva. Potrebno je prebrojati sve uzastopne podnizove (koji se nekad zovu i podniska, jer sadrže strogo uzastopne elemente niza) čija suma je deljiva zadatim brojem. Ovi podnizovi se mogu preklapati.

Na primerr, niz
2, 1, 2, 1, 1, 2, 1, 2
sadrži šest uzastopnih podnizova koji se deljivi sa 4: podniz prvih 8 brojeva, podniz {1,2,1} od 2. do 4. člana, podniz {1,2,1,1,2,1} od 2. do 7. člana, podniz {2,1,1} od 3. člana do 5. člana, podniz {1,1,2} od 4. do 6. člana, podniz {2,1,2} od 5. do 7. člana.

ULAZ:
Prva linija standardnog ulaza sadrži ceo broj c ($1 \leq c \leq 200$), broj test primera. Potom slede dve linije za svaki test

primer.
Svaki test primer počinje redom u kom su navedena dva cela broja d ($1 \leq d \leq 1\,000\,000$) i n ($1 \leq n \leq 50\,000$), gde d je delilac sume podniza, n je dužina niza. U drugoj liniji test primera nalaze se članovi niza. Članovi niza su celi brojevi iz segmenta [1, 1 000 000 000].

IZLAZ:
Za svaki test primer, odštampajte u zasebnoj liniji jedan ceo broj, koji predstavlja broj uzastopnih podnizova deljivih sa zadatim brojem d.

Ulaz:
2
7 3
1 2 3
4 8
2 1 2 1 1 2 1 2

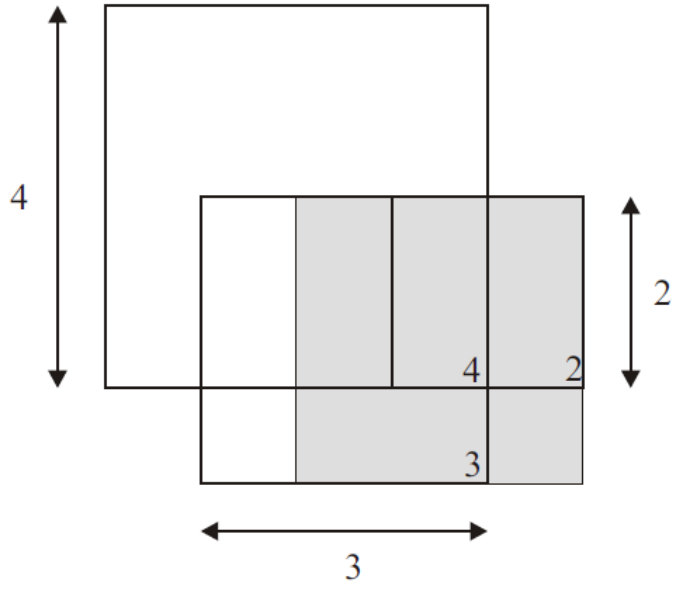
Izlaz:
0
6

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>

using namespace std;

#define MAX_NUMBER  1000000

int Prebroj[MAX_NUMBER];

int main ()
{
   int c;
   scanf("%d",&c);

   while (c)
   {
      memset(Prebroj,0,sizeof(Prebroj));
      int d,n,Sum=0,rez=0;

      scanf("%d %d",&d,&n);

      Prebroj[0]=1;
      for (int i=0;i<n;i++)
      {
         int Number;
         scanf("%d",&Number);
         Sum=(Sum+(Number%d))%d;
         rez += Prebroj[Sum]++;
      }

      printf("%d\n",rez); c--;
   }

   return 0;
}
```

11.
U matrici Anxm, popunjenoj sa 0 i 1, odrediti kvadratni blok najveće dimenzije koji se sastoji samo od nula.

0	0	0	0	0	1
1	0	0	1	0	0
0	0	0	0	0	1
1	0	0	0	1	0
1	0	0	0	1	1

1	1	1	1	1	0
0	1	2	0	1	1
1	1	2	1	1	0
0	1	2	2	0	1
0	1	2	3	0	0



Slika 1.

12.
Najduži zajednički podniz (eng. longest common subsequence): Data su dva niza **X** i **Y**. Naći niz najveće moguće dužine koji je podniz i za **X** i za **Y**.
Niz $x=(x_1, x_2, ..., x_k)$ je podniz niza $y=(y_1, y_2, ..., y_m)$, ako i samo ako postoji strogo rastući niz $(i_1, i_2, ..., i_k)$ indeksa, takav da za svako j= 1, 2, ... k važi $y_{ij}=x_j$.
Na primer, niz z=(A, N, A) je podniz niza y=(G, L, A, D, N, A) gde je odgovarajući niz indeksa (3, 5, 6). Ili (2,4,5) ako je indeks prvog elementa niza nula, a ne jedan.
Ako su data dva niza X i Y, kažemo da je niz Z zajednički podniz od X i Y, ako je Z podniz i od X i od Y.
Niz Z=(A, N, A) je zajednički podniz nizova X=(B, A, N, A, N, A) i Y=(B, R, A, N, A) i njegova dužina je 3, ali Z nije najveći zajednički podniz od X i Y, već je to niz (B, A, N, A), čija je dužina 4.

ULAZ:
Prva linija sadrži string **X**. Druga linija sadrži string **Y**. Oba stringa sadrže najmanje jedan, a najviše 1000 karaktera.

IZLAZ:
Ispisati na standardni izlaz dužinu podniza najveće moguće dužine koji je podniz i za **X** i za **Y**.

Ulaz
aleks
abcdef

Izlaz:
2

Ulaz:
abc
def

Izlaz:
0

```
#include <iostream>
#include <cstdio>
```

```
#include <cstring>
#include <algorithm>
#include <cstdlib>
#define MaxN 1000
//      Common Sequence

#define max(a,b) ((a)>(b)?(a):(b))

int Dp[MaxN+1][MaxN+1];
char X[MaxN+1],Y[MaxN+1];
int main()
{ int i, j, lenX, lenY; char pom;
  //scanf("%s", X); //scanf("%c", &pom);
  //scanf("%s", Y);
  fgets(X, MaxN+1,stdin);
  fgets(Y, MaxN+1,stdin);
    lenX=strlen(X);
  lenY=strlen(Y);
  for (i=1;i<=lenX;i++)
    for(j=1;j<=lenY;j++)
     if (X[i-1]==Y[j-1]) Dp[i][j]=Dp[i-1][j-1]+1;
     else Dp[i][j]=max(Dp[i-1][j], Dp[i][j-1]);
 printf("%d", Dp[lenX-1][lenY-1]);


   return 0;
}
```

13.
Dato je **N** celih brojevi iz intervala [1, 100 000 000] koji predstavljaju listu **A**. Odrediti najduži niz koji se sastoji od datih brojeva tako da za svaka dva susedna a_i i a_j, tako da **i = j** – 1 važi a_i deli a_j, i ispisati nađeni niz. U slučaju da postoji više rešenja sa najdužim mogućim nizom, ispisati najmanje leksikografsko.
Za dva niza X = (x_1, x_2, ..., x_n) i Y = (y_1, y_2, ..., y_n) kažemo da je X leksikografski manje od Y ako postoji **i**, tako da za sve **j < i** važi $x_j = y_j$ i $x_i < y_i$.

Ulaz:
Prvi red standardnog ulaza sadrži ceo broj **N** (2 <= **N** <= 1 000). Potom se u narednih **N** redova učitava lista brojeva **A** = {a_1, a_2, ..., a_N}, u **i** + 1-om redu ceo broj a_{**i**} (1 <= a_{**i**} <= 100 000 000).
Garantuje se da će ulaz sadržati bar dva broja a_{**i**} i a_{**j**} tako da a_{**i**} deli a_{**j**}.

OUTPUT:
Prvi red standardnog izlaza treba da sadrži prirodan broj **M** koji predstavlja dužinu nađenog niza koji zadovoljava gore opisane uslove, a sastoji se od elemenata liste **A**. Svaki elemenat liste **A** može biti upotrebljen najviše jednom. U drugom redu treba ispisati **M** brojeva odvojenih tačno jednim razmakom koji predstavljaju niz **A**. Pre prvog i nakon poslednjeg elementa ne treba da se nalazi razmak, odnsono drugi red treba da sadrži tačno **M** - 1 razmak.

Ulaz:
20
8
2
5
3
18
19
4

11
10
9
2
17
15
14
20
10
5
13
12
14

Izlaz:
5
2 2 10 10 20

Ulaz:
6
1
3
5
7
11
13

Izlaz:
2
1 3

Objašnjenje:
Postoji 5 nizova sa dužinom 2:
(1, 3), (1, 5), (1, 7), (1, 11), (1, 13).

Od njih leksikografski najmanji je (1, 3).

14.
Mali Z je rešio da se bavi menadžerskim poslom. Nakon nekoliko dana traganja za poslom, konačno se zaposlio u poznatoj kompaniji zoogle. Već prvog dana dobio je ozbiljan zadatak:

Glavni direktor firme mnogo voli da gleda filmove, međutim, za to ima mnogo malo vremena. Mali Z je zato dobio zadatak da dirkektoru snimi filmove. Direktor je malom Z-u dao spisak filmova, i za svaki od filmova: vreme kada počinje, vreme kada se završava, i koliko će Z dobiti para ako snimi taj film.

Mali Z ima na raspolaganju samo jedan video rekorder, i treba da snimi tako da zaradi što je više moguće para.

Ulaz:
Sa standardnog ulaza se u prvom redu učitava prirodni broj N (1<=N<=100000) - broj filmova na spisku. Zatim se u svakom od narednih N redova nalaze tri prirodna broja. P, K i O. Gde je 1<=P < K<=2000000000, i 1<=O<=1000. P predstavlja vreme početka filma, K predstavlja vreme kraja filma, a O predstavlja Z-ovu zaradu sa snimljen dati film.

Napomena: Ukoliko se jedan film završava u istom trenutku kada neki drugi film počinje, mali Z NE MOŽE da snimi oba filma. Drugim recima, ako je K[a] = P[b], za neke filmove a i b, tada Z ne moze snimiti i film a i film b.

Izlaz:
Na standardni izlaz ispisati samo jedan broj koji predstavlja maksimalnu Z-ovu zaradu.

Primer:

Ulaz:
3
1 5 2
2 6 3
6 8 2

Izlaz:
4

Dodatni primer:

Ulaz:
4
1 3 2
4 6 2
2 5 8
4 5 3

Izlaz:
8

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <vector>

using namespace std;
#define MaxN 100000

struct film
{
   int P, K, O;
   film ( int P1, int K1, int O1)
   {
      P = P1;K = K1;O = O1;
   }

   inline friend bool operator < (const film & X, const film &Y )
   {
      return X.K < Y.K;
   }
};

long long int DP[MaxN];
vector <film> FilmNiz;

int main()
{
   int n,i; long long int tek = 0;
   scanf ("%d",&n);
   for (i=0;i<n;i++)
```

```
    {
        int P1,K1,O1;
        scanf("%d %d %d", &P1, &K1, &O1 );
        FilmNiz.push_back(film(P1,K1,O1));
    }
    //uredjen spisak prema zavrsetku filma
    sort(FilmNiz.begin(),FilmNiz.end());

    for (i=0;i<n;i++)
    {
        tek = max(tek,0LL + FilmNiz[i].O);
        film F = FilmNiz[i];
        F.K = F.P;
        int index = lower_bound (FilmNiz.begin(), FilmNiz.begin()+i,F)-FilmNiz.begin ();
        if (index==0) DP[i]=tek;
        else
        {
            index--;

            if (FilmNiz[index].K<FilmNiz[i].P)
            tek = max(tek,FilmNiz[i].O+DP[index]);

            DP[i] = tek;
        }
    }

    printf ("%lld\n", DP[n-1]);

    return 0;
}
```