maturski rad iz

Informatike

DINAMIČKO PROGRAMIRANJE

Šta je dinamičko programiranje?

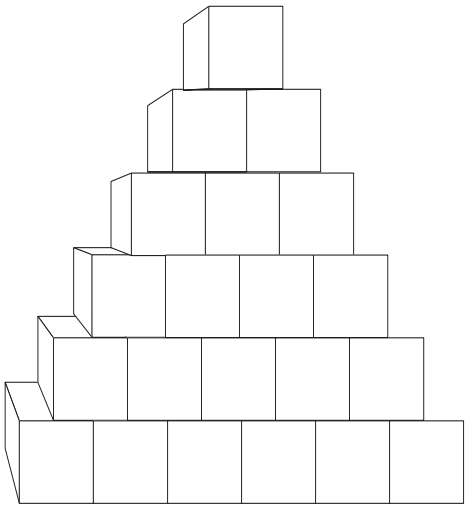
Programiranje kao oblast matematike je veoma opširna oblast koja se razvila tek sredinom ovog veka, nalazeći veliku primenu na kompjuterima i kompjuterskoj tehnologiji. Programiranje kao metod može se definisati kao niz postupaka ili procedura koje vode do željenog rezultata. Pojam algoritma možemo isto ovako definisati. To je spisak uputstava koji opisuje izvršavanje nekog problema, tako da svako uputstvo iz spiska, koje nazivamo i korak algoritma, mora da bude neka dobro poznata operacija. Algoritam može biti bilo koja radnja koju čovek svakodnevno izvršava. To na primer može biti: kuvanje ručka, opravljanje kola, pakovanje stvari itd.

Primer algoritma za zamenu točka na automobilu:

1. podigni auto
2. skini točak
3. ako je rezervni točak pripremljen idi na korak 5.
4. pripremi rezervni točak
5. namesti rezervni točak
6. spusti auto

Što se tiče dinamičkog programiranja njega možemo definisati kao skup svih algoritama tj. rešenja problema koji imaju jednu zajedničku osobinu. Ta osobina je da do krajnjeg (željenog) rezultata dolazimo pomoću pomoćnih koja imaju zajedničke podprobleme, čije rezultate upisujemo u memoriju, izbegavajući rešavanje podproblema više puta. Da bi dobili rešenje našeg problema potrebno je naći rešenja svih podproblema. Naravno da i podproblemi imaju svoje podprobleme koji se mogu preklapati. Na primer podproblemi X i Y mogu imati zajednički podproblem Z, tako da su podproblemi X i Y uslovljeni podproblemom Z, tj. da bi rešili podprobleme X i Y neophodno je prethodno rešiti podproblem Z.

Sve ovo možemo ilustrovati primerom. Potrebno je sagraditi zid u obliku trougla. Zid gradimo iz kocki jednake veličine tako što u prvi red (najniži red na samoj zemlji) naslažemo neki broj kocki jednu pored druge u jednoj liniji, tako da se svake dve susedne kocke dodiruju svojim bočnim stranama. U svaki sledeći red stavljamo jednu kocku manje, tako da se svake dve u tom redu takođe dodiruju i da svaka kocka stoji na dve iz prethodnog reda. Završavamo sa zidanjem kada u poslednji red tj. najviši red stavimo poslednju kocku. Naš zid bi onda izgledao ovako:



Neka se naš problem sastoji u tome da izgradimo zid odnosno da možemo postaviti poslednju kocku na vrh. Uslovi koji se podrazumevaju jesu da je sila gravitacije prisutna, odnosno da za svaku kocku moramo imati po dve donje na kojoj ona stoji.

Analogija između ovog zida i dinamičkog programiranja je u sledećem. Neka se naš problem sastoji u tome da postavimo poslednju kocku na zid, to jest da sagradimo ceo zid. Da bi mogli da stavimo poslednju kocku neophodno je staviti one dve kocke koje su neposredno ispod nje. Da bi stavili te dve kocke potrebno je staviti one tri kocke koje su ispod njih itd. Možemo zaključiti da za svaku kocku važi pravilo da njeno ugrađivanje u zid zahteva prethodno stavljanje kocki koje su neposredno ispod nje, a pošto svaka kocka koja je neposredno ispod, zahteva to isto od onih u nižem redu, itd, zaključujemo da postavljanje određene kocke u zid zahteva stavljanje čitavog niza kocki. To naravno ne važi za one kocke koje su pri samom dnu.

Ovo bi se moglo nazvati analizom problema. Posle ovoga možemo preći na implementaciju. Da bi napravili zid, zaključujemo prema prethodnoj analizi, da je to ostvarivo ako budemoređali red po red počev od najnižeg do najvišeg. Svaki red je uslovljen prethodnim redom. Pošto prvi red nije uslovljen ni jednim, krećemo od njega. Posle njega postavljamo red koji je neposredno iznad njega, zatim onaj sledeći i tako dalje dok ne stavimo najviši red odnosno poslednju kocku na zid.

Analogno sa stavljanjem kocki na zid jeste rešavanje podproblema u dinamičkom programiranju. Svakoj kocki odgovara jedan podproblem ili ceo problem ako je u pitanju kocka na samom vrhu. Analogno prvom redu kocki jeste skup onih podproblema koji nisu uslovljeni ni jednim podproblemom odnosno oni podproblemi koji ne zahtevaju nikakve prethodne podprobleme.

Posle njih rešavamo one podprobleme koji su uslovljeni ovim prvim podproblemima. Posle njih rešavamo one probleme koji su uslovljeni ovim prethodnim i tako dalje dok ne dođemo do rešenja krajnjeg problema. To je analogno slaganju red po red kocki na zid dok ne dođemo do one poslednje. Zbog ovoga se rešavanje problema dinamičkim programiranjem zove i *bottom-up* (odozdo-nagore), pošto krećemo da rešavamo prvo one najniže probleme, pa onda one sve više (ako naravno zamislimo hijerarhiju problema kao što je kod zida od kocki).

Šta ovo praktično znači? Ilustrujmo to na sledećem primeru. Neka naš problem bude da nađemo član A(1,1) kvadratne matrice dimenzije N, ako su nam dati članovi A(N,I) (za svako $1 \leq I \leq N$, a N je veličina matrice), i zakonitost F tako da je A(I,K)=F(A(I+1,K),A(I+1,K+1)), ($1 \leq K \leq I < N$) to jest svaki broj u matrici koja nam je delom data zavisi od dva člana: od člana matrice koji ima broj vrste za jedan veći i isti broj kolone i člana koji ima i broj vrste i broj kolone za jedan veći od traženog člana matrice. Možemo reći da nalaženje nekog člana matrice zahteva prethodno nalaženje članova koji su ispod njega. Već sada vidimo veliku analogiju ovoga sa zidom od kocki. Za član A(I,K), članovi ispod njega su znači A(I+1,K) i A(I+1,K+1). Kao i kod zida od kocki, ovde zaključujemo da treba izračunavati red po red. Pošto nam je poslednji red N-ti već dat, krećemo od N-1 reda naniže. Znači izračunamo prvo sve elemente N-1 reda po datoj formuli. Posle njih izračunavamo elemente reda N-2 itd dok ne dođemo do rešenja glavnog problema. Jasno je da svaki element matrice izračunavamo po formuli A(I,K)=F(A(I+1,K),A(I+1,K+1)) koja nam je data. Možda najbolji primer za ovaj slučaj dinamičkog programiranja je nalaženje puta sa maksimalnom sumom u trouglu koji ćemo kasnije navesti.

Navedenih nekoliko primera predstavljaju dosta jednostavne primere dinamičkog programiranja. U prethodno navedenom primeru svaki element matrice zavisi samo od dva druga elementa matrice, što u opštem slučaju dinamičkog programiranja nije slučaj. Takođe, u opštem slučaju dinamičkog programiranja podaci ne moraju biti smešteni u dvodimenzionalnoj matrici. Dok sa jedne strane to mogu biti samo nizovi, sa druge strane to mogu biti i matrice sa nekoliko dimenzija (najčešće ne više od 3). U svim tim matricama postoje relacije među elementima, kao što je slučaj sa navedenim primerom: A(I,K)=F(A(I+1,K),A(I+1,K+1)).

Potrebno je spomenuti i sledeće. Poželjno je da se sva rešenja podproblema stavljaju u takve strukture u kojima je moguće u svakom trenutku lako manipulisati podacima. Tome najviše odgovara *array* struktura u Pascal-u, ili u C-u inicijalizacija niza sa na primer *int[n]* (niz od 0 do n-1). Nije preporučljivo stavljati podatke u dinamičke strukture, iako naziv upućuje na dinamičko programiranje što dovodi do čestog mešanja pojmova *Dinamičko Programiranje* i *Dinamičke promenljive*. Dinamičkim promenljivama nije lako

pristupiti u svakom trenutku i one nisu preporučljiva struktura podataka za dinamičko programiranje.

Rekurzija

Rekurzija je metod u kome programske procedure ili funkcije pozivaju same sebe. Odmah se nameće pitanje: Zar to nije beskonačna petlja? (petlja koja se izvršava beskonačno dugo i iz koje nema izlaza). Međutim, to ovde nije slučaj, zato što u samoj proceduri, odnosno funkciji postavljamo ograničenje tj. uslov koji ako je ispunjen, procedura poziva samu sebe.

Primer: treba ispisati sve brojeve od 1 do zadatog broja n. To možemo jednostavno uraditi petljom *for* tako što uzimamo neku promenljivu *i* tako da ona menja sve vrednosti redom od 1 do n. To bi izgledalo ovako:

```
Program Ispis_od_1_do_n;
var n,i:integer;
begin
        readln(n);
        for i:=1 to n do
             writeln(i);
        readln;
end.
```

Rekurzivni postupak bi bio malo drugačiji. Analizirajmo "problem"! Neka se naš problem zove *Ispis(1,n)* pošto treba da odštampamo sve brojeve od 1 do n. Pošto znamo da štampamo broj komandom *writeln*, naš problem možemo razložiti na:

Ispis(1,n) ⇔ *Ispis(1,n-1)* + *writeln(n).*

Tj. problem se svodi na ispis svih brojeva od 1 do n-1 i ispis broja n. Tako dalje, se problem *Ispis(1,n-1)* svodi na ispis brojeva od 1 do n-2 i broja n-1, itd. Da rekurzija ne bi bila beskonačna moramo postaviti neko ograničenje, a to je:

Ispis(1,1) ⇔ *writeln(1)*

To jest: ispis brojeva od 1 do 1 ekvivalentan je ispisu broja 1. Po svemu ovome naša rekurzivna procedura bi izgledala ovako:

```
Procedure Ispis(n: integer);
begin
      if n=1 then writeln(1)
```

```
        else begin
                Ispis(n-1);
                writeln(n);
            end;
    end;
```

Dosta dobar primer rekurzije je nalaženje N-tog elementa Fibonačijevog niza. Fibonačijev niz je niz prirodnih brojeva u kome je svaki element jednak zbiru prethodna dva elementa niza, osim prva dva elementa koja su jednaka jedinici. To radimo na sledeći način. Za nalaženje N-tog broja pozivamo funkciju *Fib(N)*. Za N=1 i N=2 funkcija nam bez ikakvih daljih računanja daje odgovor *Fib(N)=1*, zato što su prva dva elementa Fibonačijevog niza jedinice. Ako je N>2 onda funkcija *Fib(N)* poziva dva puta samu sebe i to *Fib(N-1)* i *Fib(N-2)*, pošto je *Fib(N)=Fib(N-1)+Fib(N-2)*. Naša rekurzivna funkcija bi onda izgledala ovako:

```
function Fib(k:integer):longint;
begin
    if k>2 then Fib:=Fib(k-1)+Fib(k-2)
           else Fib:=1;
end;
```

Rekurzivne procedure i funkcije zaista nije teško napisati, ali one imaju jedan veliki nedostatak. To je vreme izvršavanja programa. Uzmimo za primer ovaj program za nalaženje N-tog Fibonačijevog broja. Za nalaženje prva dva elementa niza trošimo minimalno vreme, vreme potrebno da se broj upiše na dato mesto. Za nalaženje trećeg člana takođe trošimo minimalno vremena zato što je to najmanje moguće vreme za izračunavanje tog broja, to je sumiranje prva dva elementa niza. Već kod nalaženja četvrtog elementa nastaje višak računanja. Kako? Da bi izračunao četvrti element, program poziva prvo funkciju *Fib(4)*, a ona poziva *Fib(2)* i *Fib(3)*. Broj *Fib(2)* se odmah nalazi jer je on inicijalno dat, dok sam poziv *Fib(3)* poziva *Fib(2)* i *Fib(1)*. Vidimo da ovde već dolazi do preklapanja poziva što dovodi do usporavanja rada programa. Ako bi razmatrali pozive *Fib(5)* i *Fib(6)* videli bi da ima sve više i više preklapanja poziva što dovodi do sve većeg usporavanja rada programa. Vreme potrebno da se izračuna N-ti Fibonačijev broj, ovim postupkom, proporcionalno je jednako broju K^N, gde je K približno jednako Fibonačijevom broju F=1.618… To jest da bi program došao do ovog broja potrebno je približno K^N poziva funkcije *Fib*. Da bi se definitivno ubedili da je ovo veoma sporo, možemo izračunati vreme potrebno da se izračuna 100-ti Fibonačijev broj. Potrebno je, po prethodno datoj formuli K^{100} poziva. Uzmimo da je K=F, i dobija se da je broj poziva funkcije približno jednak 10^{20}. Pošto u jednoj sekundi imamo odprilike oko 10^6 poziva, dobijamo da je vreme približno jednako 10^{14} sekundi što je nekoliko miliona godina.

Pošto smo se upoznali sa rekurzijom možemo objasniti **BackTrack algoritam**. To je takozvano *pretraživanje sa vraćanjem*. Kod ovog algoritma ispitujemo sve moguće kombinacije (sva moguća rešenja problema) i odabiramo ono pravo, tj. optimalno rešenje. BackTrack-om možemo rešiti skoro svaki problem, ali on ima jedan veliki nedostatak, a to je vreme izvršavanja programa. Vreme izvršavanja algoritma koji se bazira na BackTrack-u se smatra eksponencijalnim, a to nije dobro zato što je vreme izvršavanja takvih algoritama eksponencijalno veliko za neke veće test primere.

Na primer, BackTrack-om možemo rešavati problem lavirinta, tako što ćemo tražiti sve moguće puteve i od njih odabrati onaj pravi ili najkraći, u zavisnosti šta tražimo. To radimo tako što od svakog polja lavirinta isprobavamo sva susedna polja koja su prazna, na koja možemo stati. Ako je polje prazno, idemo na njega, i na njemu primenjujemo isti postupak. Moramo postaviti jedno ograničenje, a to je da ne idemo na ona polja na kojima smo već bili. To možemo uraditi jednostavno markiranjem polja na kojima smo bili. Znači svako polje koje posetimo obojimo (na primer u crno ako su sva ostala neposećena polja bela).

Sličnost i prednost dinamičkog programiranja u odnosu na rekurziju i backtrack

Već smo videli primer rekurzivnog programa za nalaženje N-tog Fibonačijevog elementa i njegovu brzinu. Dinamičko rešenje ovog problema bilo bi traženje svih elemenata od prvog do N-tog po redu. Znači to bi bila najobičnija *for* petlja sa kojom bi tražili redom brojeve od, tačnije rečeno, trećeg do N-tog Fibonačijevog broja pošto su nam prva dva inicijalno data. Program bi onda izgledao ovako:

```
const maxn=100;
var f:array[1..maxn]:longint;
  i,n:byte;
begin
        f[1]:=1;
        f[2]:=1;
        readln(n);
        for i:=3 to n do
            f[i]:=f[i-1]+f[i-2];
         writeln(f[n]);
         readln;
end.
```

Čak i ovo rešenje ima jedan nedostatak, a to je korišćenje velike količine memorije. U ovom slučaju koristimo niz od 1 do N gde pamtimo sve Fibnačijeve brojeve do N-tog. Ako bi broj N bio dosta velik kao na primer reda

veličine 10^6, ne bismo mogli koristiti ovaj program. Najbolje rešenje ovog problema bi bilo da pamtimo samo poslednja dva Fibonačijeva broja za svako I≤N. Tada bi program izgledao ovako:

```
var f,f1,f2:longint;
i,n:integer;
begin
        readln(n);
        f1:=1;
        f2:=1;
        i:=2;
        repeat
                inc(i);
                f:=f1+f2;
                f1:=f2; f2:=f;
        until i>=n;
        writeln(f);
        readln;
end.
```

Poenta ovog poglavlja jeste uočavanje prednosti Dinamičkog programiranja u odnosu na rekurziju. BackTracking i rekurzija kao što je već rečeno, imaju veliku manu u brzini algoritma. To je posledica toga što neke podprobleme rešavaju više puta i samim tim povećavaju vreme rada programa. Treba napomenuti da je rešavanje problema kod rekurzije i backtrack-a metod koji se zove i *top-down*, zato što krećemo od vrha (od glavnog problema), pa idemo nadole. Što se tiče dinamičkog programiranja (*bottom-up*), kod njega sva rešenja podproblema nalazimo samo jednom (što je naravno i dovoljno) i smeštamo ih negde u memoriju, najčešće u matricu. **Svaki problem koji možemo rešiti Dinamičkim programiranjem, možemo rešiti i BackTracking-om, ali mnogo sporijim algoritmom.**

Memoizacija

Ostalo je još da se kaže nešto o memoizaciji. Memoizacija je sklop dinamičkog programiranja i rekurzije. Sama tehnika je *top-down*, kao kod rekurzije, ali sva rešenja pamtimo u memoriji, tako da ne dolazi do rešavanja istog problema više puta. Dinamičku matricu, tj. deo memorije u koji smeštamo rešenja problema, na početku inicijalizujemo sa nekim nemogućim brojem za rešenje (na primer sa –1, ako su rešenja svih problema pozitivna). Kada dolazimo do nekog problema, prvo pogledamo da li smo ga rešili (ako na njemu nije inicijalni broj). Ako je već rešen, ne računamo ga ponovo, već samo uzimamo rešenje za dalje rešavanje problema, a ako još nije rešen, rešavamo ga. Metod memoizacije se pokazao kao veoma povoljan, a u nekim slučajevima je i brži od dinamičkog programiranja. To je zbog toga što dinamičkim programiranjem

rešavamo sve moguće podprobleme, a memoizacijom samo one koji su nam potrebni.

Formulacija Dinamičkog programiranja

Jedino što preostaje da se kaže jeste kako formulisati problem koji se rešava dinamičkim programiranjem, to jest, rećićemo nešto o formulaciji samog problema. Svakom problemu koga možemo rešiti dinamičkim programiranjem možemo pridružiti određene parametre, kao i svakom njegovom podproblemu. Na primer, za određivanje N-tog Fibanačijevog broja *Fib(N)*, jedini parametar potreban da nađemo rešenje je N, to jest njegov redni broj u nizu. Naravno nije uvek slučaj da problem ima samo jedan parametar, što ćemo videti iz narednih primera. Znači u opštem slučaju dinamičkog programiranja svaki problem možemo formulistai kao *P(p1,p2…pk)*, gde je *P* problem, a *p1,p2…pk* parametri problema. U najvećem broju slučajeva *k* nije veće od 3, to jest problem ima najviše tri parametra. Broj parametara važi ne samo za glavni problem, nego već i za sve ostale podprobleme, tako da svaki podproblem, kao i sam problem ima jedinstvenu k-torku parametara koja jednoznačno označava taj problem. Možemo uvideti da smisao dinamičkog programiranja gubi smisao za veliko k (zbog manjka memorije), pa se onda uprkos maloj brzini algoritma pribegava metodi *top-down*.

Znači, prvi korak pri rešavanju problema jeste određivanje parametara problema i nalaženje one k-torke parametara koja odgovara glavnom problemu. Za Fibonačijev niz to bi bio samo broj N. Potrebno je još nešto istaći, a to je da je potrebno pri implementaciji programa formirati k-dimenzionalnu strukturu, tj. matricu u koju ćemo smeštati rešenja podproblema. Za svaki parametar problema potrebna je jedna dimenzija te matrice, a ograničenja dimenzija matrice su jednaka maksimalnim i minimalnim vrednostima parametra koji odgovara datoj dimenziji matrice. Za Fibonačijev niz pravimo niz, pošto ima samo jedan parametar sa ograničenjima 1 i maksimalne vrednosti n:

```
array[1..maxn] of integer;
```

Drugi korak bi bio nalaženje onih podproblema tj. k-torki parametara koji nisu uslovljeni rešavanjem ni jednog drugog podproblema i njihovo rešavanje. To su takozvani *trivijalni problemi*. Bez njih ne bi mogli početi rešavanje ostalih problema.

Treći korak bi bio nalaženje zakonitosti između pojedinih podproblema. Svaki problem u dinamičkom rešavanju zadatka, osim onih trivijalnih, uslovljen je određenim podproblemima. To jest rešenje svakog problema možemo napisati

u funkciji rešenja nekih podproblema. Ako je R(p1,p2…pk) rešenje problema P(p1,p2…pk) onda je:

$$R(p1,p2...pk) = f(R_1(p_{1,1},p_{1,2}...p_{1,k}), R_2(p_{2,1},p_{2,2}...p_{2,k})... R_t(p_{t,1},p_{t,2}...p_{t,k}))$$

gde je *f* funkcija zavisnosti pojedinih rešenja, a *t* broj problema od kojih je problem *P* zavisan. Potrebno je, znači, naći onakvu funkciju *f* koja važi za sve probleme, uključujući i glavni problem i naravno ne mora da važi za trivijalne probleme.

Četvrti korak u svemu ovome bio bi nalaženje hijerarhije svih problema, tako da za svaka dva problema *P* i *Q* važi da ako je problem *P* u hijerarhiji direktno iznad problema *Q* onda je problem *P* direktno zavistan od problema *Q*, tj. da rešenje problema *P* zavisi od rešenja problema *Q*. Može se lako uočiti da u ovakvoj hijerarhiji glavni problem stoji na samom vrhu, a trivijalni problemi stoje na samom dnu hijerarhijskog drveta. Nešto što se podrazumeva je da ne postoje ciklusi zavisnosti. To je formalno rečeno niz problema $P_1,P_2...P_m$ tako da P_i zavisi od P_{i+1} za $1 \leq i \leq m-1$ i P_m zavisi od P_1. Ako bi ovakvi ciklusi postojali ne bi mogli da nađemo rešenje glavnog problema. Zato treba naći onakvu hijerarhiju u kojoj nema ciklusa zavisnosti.

Pošto smo rekli kako bi trebalo analizirati opšti problem koji se rešava dinamičkim programiranjem, pokazaćemo kako bi izgledao opšti program koji je rađen dinamičkim programiranjem.

Da bi došli do rešenja glavnog problema potrebno je, kao što je već rečeno rešiti čitavu hijerarhiju podproblema. Jedini problemi koji nisu zavisni ni od jednog drugog podproblema su trivijalni podproblemi, pa je logično da program prvo nalazi rešenja tih problema. Posle toga, program može krenuti na rešavanje svih ostalih podproblema idući po hijerarhiskim stepenima, sve do konačnog glavnog problema. Naš opšti program u pseudo-paskalu bi izgledao ovako:

```
Program Dinamičko_Programiranje
Učitaj_podatke
Reši_trivijalna_rešenja
i:=1
repeat
        Uvećaj(i)
        Reši_Hijerarhiski_Stepen(i)
until i=Najveći_stepen_hijerarhije
Ispisi_glavno_rešenje
end.
```

Primeri:

Posle formulacije navešćemo nekoliko primera dinamičkog programiranja koji su poređani po težini. Svaki primer je zadatak sa republičkog, saveznog ili nekog međunarodnog takmičenja. Primeri sadrže tri dela: tekst zadatka, rešenje u obliku objašnjenja i rešenje napisano u Paskalu.

Red za karte

U redu pred blagajnom se nalazi n ($n \leq 200$) ljudi koji čekaju da kupe kartu. Svako kupuje samo jednu kartu za sebe. Dat je niz t od 1 do n. Broj t_i označava vreme potrebno i-tom čoveku u redu da kupi jednu kartu za sebe. Kupci u redu su se organizovali tako da pojedini kupci mogu kupiti kartu sebi i onom iza sebe u redu za vreme r_i, ako se radi o osobi *i* i *i+1*. Dati su broj n i nizovi t i r. Potrebno je minimizirati vreme potrebno da svi kupci dobiju karte, odnosno potrebno je odrediti minimalno vreme koje je potrebno da svi kupci dobiju kartu.

Rešenje. Označimo naš problem sa P(n), pošto trebamo odrediti minimalno vreme da prvih n kupaca kupe kartu. Ako imamo rešenje problema P(n-1) i P(n-2) onda je rešenje problema veoma lako pronaći. To je onda:

$R(n)=\min((R(n-1)+t_n),(R(n-2)+r_{n-1}))$

Možemo lako uočiti da ovo važi i za svaki podproblem P(k) za $3 \leq k \leq n$. Probleme P(1) i P(2) ne možemo rešiti na ovaj način jer bi onda morao postojati problem P(0). To su trivijalni problemi i njih rešavamo na sledeći način:

$R(1)=t_1$ - to jest vreme da se prvi kupac snadbe kartom je t_1

$R(2)=\min((t_1+ t_2),r_1)$ - vreme potrebno da se prva dva kupca snadbeju kartom jednako je minimumu sumi vremena za prvog i drugog da kupe kartu i vremenu da prvi kupi kartu za obojicu.

Sva rešenja stavljamo u niz *din[1..maxn]*.

Očigledno je da probleme treba rešavati redom od P(1) do P(n).

Program

```
const maxn=200;
var t:array[1..maxn] of integer;
    r:array[1..maxn-1] of integer;
    din:array[1..maxn] of integer;
    n,i:byte;
```

```
function min(x,y:integer):integer;
begin
        if x<y then min:=x
                else min:=y;
end;

begin
      writeln('Unesi broj kupaca');
      readln(n);
      writeln('Prvi niz':);
      for i:=1 to n do read(t[i]);
      readln;
      writeln('Prvi niz':);
      for i:=1 to n-1 do read(r[i]);
      readln;
      din[1]:=t[1];
      din[2]:=min(t[1]+t[2],r[1]);
      for i:=3 to n do
          din[i]:=min(din[i-2]+r[i-1],din[i-1]+t[i]);
      writeln(din[n]);
      readln;
end.
```

Najveća suma u trouglu

Na slici je dat brojni trougao. Napisati program koji nalazi najveću sumu brojeva, po putu koji počinje na vrhu, a završava se na nekom polju baze.

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

U svakom koraku možemo ići dijagonalno dole levo ili dijagonalno dole desno.

Rešenje. Ovo je već problem koji ima dva parametra. Sam problem možemo formulistai kao P(1,1), tj. put naniže sa najvećom sumom koji počinje u redu 1 i u koloni 1. Da bi rešili problem P(i,j) potrebna su nam rešenja problema P(i+1,j) i P(i+1,j+1). Ako imamo ova rešenja onda je:

R(i,j)=max(R(i+1,j),R(i+1,j+1))+trougao(i,j)

Trougao (i,j) je vrednost u trouglu u i-tom redu i j-toj koloni.

Da bi dobili rešenje formiramo trougao sa istim dimenzijama i to tako da mesto (i,j) u novoformiranom trouglu označava maksimalnu sumu od tog polja pa naniže. Očigledno je da je polje (1,1) u novoformiranom trouglu konačno rešenje. Mesta u novoformiranom trouglu čiji je broj reda n imaju vrednost jednaku sa istim mestom u zadatom trouglu. To su trivijalni problemi. Uviđamo da je hijerarhija podproblema takva da treba krenuti od reda sa rednim brojem n pa naniže do prvog reda.

Program

```
program Suma_u_trouglu;
const maxn=100;
var t,din:array[1..maxn,1..maxn] of integer;
    i,j,n:byte;

function max(x,y:integer):integer;
begin
     if x>y then max:=x
            else max:=y;
end;

begin
     readln(n);
     for i:=1 to n do begin
         for j:=1 to i do
             read(t[i,j]);
     readln;
     end;
     for j:=1 to n do
         din[n,j]:=t[n,j];
     for i:=n-1 downto 1 do
       for j:=1 to n do
         din[i,j]:=max(din[i+1,j],din[i+1,j+1])+t[i,j];
     writeln(din[1,1]);
     readln;
end.
```

Najduži palindrom

Dat je niz a od n karaktera. Podniz niza se dobija izbacivanjem proizvoljnog broja karaktera datog niza (na primer podniz niza *abc* je niz *ac* jer se on dobija izbacivanjem karaktera *b*, ali podniz nije i *ca*, podnizom nekog niza podrazumeva se i sam taj niz). Potrebno je odrediti dužinu najdužeg podniza datog niza koji je palindrom. Palindrom je svaki niz $x_{1..m}$ tako da za svako i ($1 \le i \le m$) važi $x_i = x_{m+1-i}$.

Rešenje. Naš problem je da odredimo dužinu najdužeg palindroma od prvog do n-tog člana, pa ga možemo zapisati u obliku P(1,N). Neka je R(i,j) dužina najdužeg palindroma koji je podniz niza od člana i do člana j. Možemo primetiti da ako je $a_i=a_j$, onda je R(i,j)=R(i+1,j-1)+2. To znači da je dužina palindroma od i do j jednaka dužini palindroma od i+1 do j-1 uvećanoj za 2, ako je $a_i=a_j$. U suprotnom, R(i,j)=max(R(i+1,j),R(i,j-1)), tj. dužina najdužeg palindroma je jednaka većem od dužina najdužeg palindroma koji se dobija skidanjem jednog znaka sleva i dužine najdužeg palindroma koji se dobija skidanjem jednog znaka sdesna.

Naša dinamička matrica imaće dva parametra i to mesto sa leva od koga počinjemo tražiti najveći palindrom i mesto sdesna. Da bi rešili problem P(i,j) (podrazumeva se i<j) potrebna su nam rešenja problema P(i+1,j),P(i+1,j-1) i P(i,j-1). Daljom hijerarhijom vidimo da za svako rešenje R(i,j) potrebna su sva rešenja R(k,l) tako da $i\le k\le l\le j$. Trivijalni podproblemi ovog problema su očigledno svi podproblemi tipa P(i,i) i P(i,i+1). Očigledno je R(i,i)=1 za svako $1\le i\le n$. Ako su u nizu a susedni članovi a_i i a_{i+1} jednaki onda je R(i,i+1)=2, a ako su različiti onda je R(i,i+1)=1. Očigledna hijerarhija podproblema je da prvo treba rešiti takve P(i,j) sa manjim razlikama *i* i *j*, a onda sa većim, tj. prvo treba rešavati probleme manjih podnizova pa onda većih.

Program

```
program Palindrom;
const maxn=100;
var a:string[maxn];
   din:array[1..maxn,1..maxn] of byte;
   i,j,k,n:byte;
function max(x,y:byte):byte;
begin
    if x>y then max:=x
          else max:=y;
end;

begin
    writeln('Unesi string:');
    readln(a);
    n:=length(a);
    for i:=1 to n do
       din[i,i]:=1;
    for i:=1 to n-1 do
       if a[i]=a[i+1] then din[i,i+1]:=2
                      else din[i,i+1]:=1;
    for k:=3 to n do
       for j:=k to n do begin
          i:=j-k+1;
          if a[i]=a[j] then din[i,j]:=din[i+1,j-1]+2
```

```
                else din[i,j]:=max(din[i+1,j],din[i,j-1]);
        end;
    writeln(din[1,n]);
    readln;
end.
```

Bratska podela

Posle očeve smrti, njegovi sinovi nasleđuju svu njegovu imovinu. Pošto je otac imao samo dva sina, oni su se dogovorili da njegovu imovinu podele na dva dela tako da je absolutna razlika između cena ta dva dela minimalna. Cela očeva imovina se sastoji od predmeta određene vrednosti. Dat je broj N, broj predmeta i vrednost svakog od njih. Te vrednosti su celi brojevi. Potrebno je podeliti skup predmeta na dva dela, tako da je absolutna razlika izeđu ukupnih vrednosti delova minimalna i ispisati te dve vrednosti.

Rešenje. Ovaj problem se svodi na problem ranca - *KnapSack*. Ako sa S označimo sumu vrednosti svih predmeta, onda je problem u tome da se ranac u koji može da stane najviše S/2 vrednosnih jedinica napuni sa što većom ukupnom vrednošću predmeta, puneći ga sa datim predmetima nasledstva. Skup predmeta koji smo izabrali za ranac dajemo jednom sinu, a sve ostale dajemo drugom.

Problem možemo posmatrati kao P(N,S/2), tj. kako na najbolji način napuniti ranac zapremine S/2 sa prvih N predmeta. U opštem slučaju je to P(i,j). Ako je vrednost i-tog predmeta k, možemo videti da je P(i,j) bolja vrednost od P(i-1,j-k) i P(i-1,j). Uvodimo niz *din* na sledeći način: član din_j je jednak *true* ako postoji skup predmeta čija je ukupna vrednost *j*, a u protivnom je *false*. Inicijalizujemo niz sa din_0 = *true* (pošto je praznom skupu vrednost 0), a sve ostale članove sa *false*. Niz *din* rešavamo po broju predmeta, tj. rešavamo prvo samo za prvi predmet, onda za prva dva, pa za prva tri itd. Vidimo da ako je niz *din* rešen već za prvih *i-1* predmeta, onda je za prvih *i* predmeta za svako *j* din_j postaje *true* ako din_{j-k} = *true* za prvih *i-1* predmeta (k – vrednost *i-tog* predmeta).

Program

```
const maxn=100; maxt=200;
var a:array[1..maxn] of 0..maxt;
   din:array[0..maxn*maxt div 2] of boolean;
   suma,polasume:integer;
   i,j,l,n:byte;

begin
    readln(n);
    for i:=1 to n do
       read(a[i]);
```

```
    readln;
    suma:=0;
    for i:=1 to n do
        inc(suma,a[i]);
    polasume:=suma div 2;
    din[0]:=true;
    for i:=1 to polasume do
        din[i]:=false;
    for i:=1 to n do
        for j:=polasume downto a[i] do
            if din[j-a[i]] then din[j]:=true;
    i:=polasume;
    while not din[i] do dec(i);
    writeln(i,' ',suma-i);
    readln;
end
```

Sadržaj: