

Dinamičko programiranje

Kolegij Natjecateljsko programiranje 2011/2012

U ovom se dokumentu kroz zadatke objašnjavaju neki bitniji koncepti dinamičkog programiranja, a primarna joj je namjena kao materijal uz predavanja na Natjecateljskom programiranju.

Što je dinamičko programiranje:

Dinamičko je programiranje metoda tj. pristup rješavanju problema razbijanjem na manje potprobleme. Nadalje, koristi se činjenica da su mnogi od tih potproblema zapravo isti, stoga se rješenje svakog potproblema računa samo jednom, zbog čega su rješenja dinamičkim programiranjem često za više redova veličine efikasnija od naivnih rješenja.

Valja primijetiti da je osnova za rješavanje problema dinamičkim programiranjem rekurzija. Ukoliko imamo dobro napravljenu rekurziju, tada trivijalnim korakom (memoizacijom, o kojoj će biti govora) dobivamo rješenje dinamikom. Nadalje, kada god želimo rješavati problem dinamikom, bitno je da točno odredimo **stanje**. Stanje u dinamičkom programiranju definirat ćemo kao skup varijabli koje **potpuno** definiraju problem. Nama to znači da će rješenje za isto stanje uvijek biti isto.

Da ne duljimo s teorijom, slijedi nekoliko zadataka:

1. Zadatak

Zadana je ploča s $1 \times n$ kvadratića te 1×2 , 1×3 i 1×4 domine. Na koliko se načina ploča može popločati zadanim dominama ako od svake domine imamo neograničeno mnogo primjeraka. Naravno, svako polje mora biti pokriveno točno jednom.

Rješenje:

Neka nam $f(n)$ označava broj načina da se poploča $1 \times n$ ploča sa zadanim dominama. Sada razmotrimo kako definirati tu funkciju. Kao kod matematičke indukcije, krećemo s trivijalnim slučajevima.

$$f(0) = 1$$

Ovo je očito, budući da ploču 1×0 (ploču bez ijednog stupca) možemo popločati na samo jedan način, i to tako da ne upotrijebimo nijednu dominu.

Nadalje, kako bismo riješili naš početni problem, valja pretpostaviti da znamo rješenja za sve ploče koje su manje od $1 \times n$, odnosno da nam je za svaki $m \leq n$ poznat $f(m)$.

Također znamo da ako popločavamo ploču $1 \times n$, tada na njenom zadnjem polju završava neka domina. Tako za ploču koja je veća od 1×3 imamo 3 slučaja:

- 1) Na zadnjem polju završava domina 1×2
- 2) Na zadnjem polju završava domina 1×3
- 3) Na zadnjem polju završava domina 1×4

Ako razmislimo o slučaju 1), lako je uočiti da će broj popločavanja $1 \times n$ ploče tako da na zadnjem polju završava domina 1×2 biti jednaka broju popločavanja $1 \times n$ ploče bez zadnja dva stupca, što je upravo jednako $f(n-2)$, budući da je ploča $1 \times n$ bez zadnja dva stupca ekvivalentna ploči $1 \times (n-2)$.

Analogno zaključujemo da je broj načina za slučaj 2) jednak $f(n-3)$, dok je za slučaj 3) jednak $f(n-4)$.

Budući da za zadnje polje imamo sve tri mogućnosti, konačno će nam rješenje biti zbroj rješenja ta tri slučaja.

Tako dolazimo do:

$$f(n) = f(n-2) + f(n-3) + f(n-4), \text{ za } n \geq 4$$

Za $n = 1, 2, 3$ funkcije lako izračunamo ručno, i dobivamo 0, 1, 1 (1x1 ploču ne možemo popločati).

Zadatak 2:

Opet imamo 1xn ploču te 1x2, 1x3 i 1x4 domine, no ovoga je puta pitanje koliko najmanje domina moramo potrošiti da bismo popločali ploču.

Rješenje:

Na ovom ćemo problemu označiti $f(n)$ kao funkciju koja nam kaže minimalni broj domina za popločati 1xn ploču. Krenimo kao i u prošlom primjeru od trivijalnog slučaja, opet je to 0. Lako je vidjeti da je $f(0) = 0$, jer da bismo popločali praznu ploču ne moramo upotrijebiti nijednu dominu.

Nadalje, prisutna su ista tri slučaja kao i prije. Pogledajmo opet slučaj 1). Najmanji broj domina za 1xn ploču ako na njenom zadnjem polju završava 1x2 domina bit će jednak najmanjem broju domina za 1x(n-2) ploču, ali na to moramo dodati i samu 1x2 dominu, pa će ukupno rješenje za slučaj 1) biti $f(n-2) + 1$.

Analogno je i za slučajeve 2) i 3). No ovoga puta, umjesto da zbrajamo te tri vrijednosti, zanima nas najmanja od njih, budući da će samo jedan slučaj zapravo vrijediti.

Stoga dolazimo do rekurzivne funkcije:

$$f(n) = \min(f(n-2)+1, f(n-3)+1, f(n-4)+1) \text{ za } n \geq 4$$

ili, ljepše zapisano

$$f(n) = 1 + \min(f(n-2), f(n-3), f(n-4)) \text{ za } n \geq 4$$

Rješenja za 2 i 3 su 1, dok rješenje za 1 ne postoji. Budući da se radi o minimizacijskom problemu, kada rješenje ne postoji, obično mu se dodjeljuje beskonačna vrijednost, tj. vrijednost koja je daleko veća od bilo kojeg legalnog rješenja.

Zadatak 3:

Zadan je broj n , a zatim n vrijednosti kovanica. Ako svih kovanica imamo beskonačno, koliko je minimalno kovanica potrebno da se postigne neka zadana suma s .

Rješenje:

Ako malo bolje razmislimo, zadatak je gotovo identičan 2. zadatku, osim što u 2. zadatku imamo fiksne tri veličine, dok ih ovdje imamo n . No vrlo ćemo lako riješiti i ovo.

Jednako tako, umjesto tri slučaja u drugom zadatku, ovdje ih imamo n , a kako n nije unaprijed poznat, riješit ćemo to tako da iteriramo po skupu dostupnih vrijednosti kovanica. Osim toga, rješenje je potpuno isto kao i u 2. zadatku.

Pogledajmo kako bi funkcija f izgledala u C-u:

```
int f(int s){
if(s == 0)
```

```

        return 1;

    int ret = inf;
    for(int i = 0; i < n; ++i){
        if(s - kovanice[i] >= 0)
            ret = min(ret, 1 + f(s-kovanice[i]));
    }

    return ret;
}

```

Komentar (memoizacija): Uz prva dva zadatka nema koda, dok je u trećem složenost ovako direktno implementiranog koda eksponencijalna. U ovoj implementaciji ne koristimo jednu od važnih činjenica u dinamici, a to je da ne računamo rješenja istih potproblema više puta.

Tome ćemo doskočiti vrlo jednostavno. Uvedimo pomoćni niz *mem*. Na početku ćemo postaviti sva mjesta u nizu *mem* na vrijednost koja nije legalno rješenje, recimo -1, a nakon toga na mjesto *mem[x]* spremati vrijednosti funkcije $f(x)$ kako ju računamo. Tako ćemo kad pozovemo $f(x)$ znati jesmo li već izračunali vrijednost za isti taj x , te ako jesmo, samo pročitamo vrijednost iz niza i vratimo ju. Ako još nismo izračunali vrijednost za dotični x , računamo normalno.

```

int f(int s){
    if(s == 0)

        return 1;

    if(mem[s] != -1) //znamo da smo već izračunali
        return mem[s];

    int ret = inf;
    //zasad nema rješenja, postavimo na beskonačno

    for(int i = 0; i < n; ++i){
        if(s - kovanice[i] >= 0)
            ret = min(ret, 1 + f(s-kovanice[i]));
    }

    mem[s] = ret;
    return ret;
}

```

Ovako uistinu nikad ne računamo istu stvar dvaput. Nakon što jednom izračunamo $f(x)$, pospremamo vrijednost u niz *mem*, i zatim idući put kad pozovemo f za taj isti x , samo ćemo pogledati u niz, shvatiti da smo već izračunali i vratiti vrijednost. Pri tome je bitno da niz bude dovoljno velik. Dobivamo vremensku složenost $O(n*s)$.

Komentar 1: Ovdje je stanje malo manje očito nego u prva dva zadatka, budući da bi se moglo pomisliti da izbor skupa kovanica ima neke veze sa stanjem. To nije točno, budući da skup kovanica zapravo samo definira moguće prijelaze između stanja te nam ni u kojem momentu nije bitno kako smo došli u bilo koje stanje. Bitno nam je da, kad već jesmo u nekom stanju, znamo u koja stanja možemo iz njega. Stoga je stanje, kao i u prva dva zadatka, samo jedan broj.

Komentar 2: Stanja u ova tri zadatka uistinu nisu baš raznolika, uvijek imamo samo jedan broj, pa je tako i mnogo lakše napraviti niz za memoizaciju. Kad bi stanje bilo kompleksnije, imali bismo niz s više od jedne dimenzije, točnije s po jednom dimenzijom za svaku varijablu koja nam treba za stanje.

Zadatak 4:

Zadana je $n \times n$ ploča sa pozitivnim cijelim brojem u svakom polju. Imamo figuru koja se u jednom potezu može pomaknuti ili za jedno polje dolje ili za jedno polje desno. Figura kreće iz gornjeg lijevog ugla i završava u donjem desnom. Kolika je maksimalna suma brojeva na putu?

Rješenje:

Ovdje je odmah jasno da nam jedan broj neće biti dovoljan za određivanje stanja, evidentno moramo imati dva broja u stanju, i to jedan za red, a drugi za stupac. Prema tome, stanje odgovara jednom polju. Označimo funkciju $f(x, y)$ koja nam kaže najveću sumu koju možemo ostvariti od polja (x, y) (x -ti redak, y -ti stupac) do polja $(n-1, n-1)$ (donje desno). Vidimo da čim napravimo funkciju, rješenje dobivamo kao $f(0, 0)$.

Nadalje, označimo s $A[i][j]$ broj u i -tom retku i j -tom stupcu ploče.

Bacimo se sada na funkciju. Počinjemo s poljem (x, y) . Trivijalni je slučaj $x = n-1$ i $y = n-1$, za taj je slučaj rješenje $A[n-1][n-1]$.

Pretpostavimo da znamo $f(x', y')$ za sva polja na koja možemo doći s polja (x, y) . U tom slučaju znamo i vrijednost funkcije za polja na koja možemo doći direktno iz (x, y) , a to su $(x+1, y)$ i $(x, y+1)$. Kada smo na polju (x, y) , imamo samo dvije mogućnosti, ili ići na polje $(x+1, y)$ ili na $(x, y+1)$. Optimalno rješenje dobivamo tako da idemo onim od ta dva puta koji je bolji (veći). To nam govore vrijednosti f za ta dva polja. Stoga uzimamo veću od te dvije vrijednosti i na nju dodajemo $A[x][y]$, budući da smo, čim smo došli na polje (x, y) , pokupili taj broj. Pri tome naravno, moramo paziti da nas potez ne odvede preko ruba ploče.

Funkcija bi izgledala ovako, odmah radimo i memoizaciju.

```
int f(int x, int y){
    if(x == n-1 && y == n-1)
        return A[x][y];
    if(mem[x][y] != -1)
        return mem[x][y];

    int ret = 0;
    if(x < n-1)
        ret = max(ret, f(x+1, y) + A[x][y]);
    if(y < n-1)
        ret = max(ret, f(x, y+1) + A[x][y]);
    mem[x][y] = ret;
    return ret;
}
```

Komentar: Implementacija je prilično jednostavna, dvije stvari vjerojatno zaslužuju kratki komentar. Najprije, kao što je i za očekivati, niz *mem* u ovom primjeru ima dimenzija onoliko koliko ima varijabli u

stanju, u ovom slučaju dvije. Osim toga, nakon prva dva if-a, ret se postavlja na nulu. To je zato što se ovdje, za razliku od prethodna dva zadatka radi o maksimizacijskom problemu, stoga ću za inicijalnu vrijednost odabrati nešto što je manje od svih legalnih rješenja.

Vremenska složenost je $O(n^2)$, isto kao i memorijska.

Zadatak 5:

Zadana je tabla čokolade veličine $n \times m$. Moguće je napraviti rez po jednom od stupaca ili po jednom od redaka. Nakon tog reza, čokolada se raspada na dvije manje. Svaku od njih posebno možemo rezati kao i prvu. Cilj nam je cijelu čokoladu razrezati na kvadratne dijelove, a budući da su rezovi vrlo bolni, želimo to postići u što manje rezova.

Rješenje:

Opet moramo najprije odrediti stanje. I ovoga će puta u stanju biti dva broja, i to dimenzije table čokolade. Funkcija $f(x, y)$ će nam za tablu čokolade veličine x redaka \times y stupaca reći minimalni broj bolnih rezova da bi svi komadi bili kvadratni.

Krenimo opet na funkciju. Trivijalni slučaj dobivamo kada je x jednak y , odnosno kada tabla već jest kvadrat. Tada vraćamo 0. Ako pak tabla nije kvadratna, moguće je $n-1$ bolnih rezova po recima i $m-1$ po stupcima. Koji god da odaberemo i napravimo, tabla se raspada na dvije manje, stoga je jasno da ćemo moći izabrati samo jedan.

Upravo je to rješenje zadatka. Pogledajmo prvo moguće rezove po recima. Moguće ih je $n-1$, a ukoliko napravimo rez prije i -tog retka, tada se tabla raspada da dvije nove table dimenzija (i, y) te $(n-i, y)$. Pri tome, $1 \leq i \leq n-1$.

Analogno, ako režemo po stupcima, moguće je $m-1$ rezova, a ukoliko napravimo rez prije i -tog stupca, tabla se raspada na nove, dimenzija (n, i) te $(n, m-i)$. Pri tome $1 \leq i \leq m-1$. Kod je prepušten kao vježba čitatelju.

Komentar: Memorijska složenost na ovom zadatku je $O(n \cdot m)$, jer je tolika veličina niza u koji ćemo spremati vrijednosti funkcije. Za vremensku složenost valja primijetiti da je broj stanje $n \cdot m$, a za svako stanje isprobavamo $n-1 + m-1$ rezova, pa će složenost biti $O(n \cdot m \cdot (n+m))$. Ako su n i m istog reda veličine, to će biti $O(n^3)$.

Zadatak 6 (Nikola);

(Svi prijašnji zadaci bili su opća kultura, dok je ovaj preuzet sa HSIN-ovog županijskog natjecanja te je korišten na domaćoj zadaći iz dinamika 1. Ovdje dolazi u sažetoj verziji.)

Zadan je niz od n polja, označenih s 1 do N . Čovjek (u zadatku evidentno Nikola) skače po poljima. Kreće od polja 1.

Prvi skok mu mora biti na polje 2, ali za svaki sljedeći ima dvije opcije:

- skočiti unatrag onoliko polja koliko mu je dug bio zadnji skok
- skočiti unaprijed onoliko koliko mu je dug bio zadnji skok + 1

Dakle, kad napravi prvi skok i nađe se na polju 2, ima opciju ili skočiti na 1 (unatrag za duljinu zadnjeg skoka) ili pak na 4 (unaprijed za zadnji skok + 1).

Svako polje ima porez koji se mora platiti kad se stane na njega. Nikoli je cilj doći na polje N tako da plati najmanje poreza.

Rješenje:

Neiskusni bi čitatelj mogao doći u iskušenje da zaključi da je za stanje dovoljna samo pozicija. To je, naravno, netočno. Ukoliko definiramo funkciju f koja nam daje rješenje za stanje, a stanje, pak, odredimo samo preko pozicije, to bi značilo da će $f(x)$ vratiti vrijednost najjeftinijeg puta od x -te pozicije do zadnje.

Moralo bi, stoga, vrijediti da je $f(x)$ uvijek jednak, inače nemamo temeljne pretpostavke za rješenje dinamikom. No iz stanje x nemamo točno definirane prijelaze u druga stanja, budući da nam je potrebna informacija o tome s kojeg smo polja došli.

Ovome je lako doskočiti ako stanje promatramo malo apstraktnije. Budući da smo shvatili da nam je bitna informacija o polju s kojeg smo skočili na polje x , pokušajmo stanje definirati kao par (trenutno polje, prethodno polje). Iz stanja lako možemo izračunati skokove koje možemo napraviti.

Zapitajmo se sad ponovo: želimo li da nam $f(\text{trenutno}, \text{prethodno})$ uvijek vraća istu vrijednost kad ju pozovemo s istim parametrima? Također, jesu li nam dobro definirani prijelazi? S obzirom da je skup poteza iz tog stanja uvijek isti, odgovor je da. Zato možemo zaključiti da je $(\text{trenutno}, \text{prethodno})$ dobro određeno stanje.

Sada valja razraditi funkciju. Ako je trenutno polje N , došli smo do kraja i tada vraćamo samo vrijednost poreza za ulazak na to polje.

Ako, pak, nismo na kraju, iz trenutnog i prethodnog polja dobivamo duljine prethodnog skoka kao $\text{abs}(\text{trenutno} - \text{prethodno})$. Stoga možemo skočiti na polja $\text{trenutno} - \text{abs}(\text{trenutno} - \text{prethodno})$ i $\text{trenutno} + \text{abs}(\text{trenutno} - \text{prethodno}) + 1$, ako time ne bismo ispali s ploče. Prema tome, sa stanja $(\text{trenutno}, \text{prethodno})$ možemo skočiti na stanje $(\text{trenutno} - \text{abs}(\text{trenutno} - \text{prethodno}), \text{trenutno})$ te

$(\text{trenutno} + \text{abs}(\text{trenutno} - \text{prethodno}) + 1, \text{trenutno})$. Pri tome u stanja kao drugi parametar stavljamo trenutno polje, zato što smo upravo skočili s njega, pa to trenutno polje novim stanjima postaje prethodno.

Budući da radimo minimizaciju, isto kao i prije, uzimamo bolju (manju) od ovih funkcija za ova dva stanja te dodajemo porez za trenutno polje. Konačno rješenje dobivamo kao $f(1, 0)$.

Implementacija bi bila ovakva.

```
int f(int trenutno, int prethodno){
    if(trenutno == N)
        return porez[trenutno];

    if(mem[trenutno][prethodno] != -1)
        return mem[trenutno][prethodno];

    int skok = abs(trenutno - prethodno);
    int ret = inf;

    if(trenutno - skok >= 1)
        ret = min(ret, porez[trenutno] + f(trenutno - skok, trenutno));
    if(trenutno + skok + 1 <= N)
        ret = min(ret, porez[trenutno] + f(trenutno+skok+1, trenutno));

    mem[trenutno][prethodno] = ret;
    return ret;
}
```

Komentar: Memorijska je složenost $O(n^2)$, a vremenska jednako toliko.

Zadatak 7:

(Preuzeto sa županijskog natjecanja 2010. godine, iako je također opća kultura i postoji već dugo.)

Zadan nam je broj n , koji označava broj raspoloživih ljudi, a ujedno i broj poslova. Nadalje, zadana je $n \times n$ matrica A , takva da $A[i][j]$ označava koliko moramo platiti i -tom čovjeku da odradi j -ti posao. Svakom čovjeku moramo dodijeliti točno jedan posao, što ujedno znači da će svaki posao obavljati točno jedan čovjek. Cilj nam je napraviti najjeftiniji raspored.

Na ovom ćemo zadatku, za razliku od ostalih dosad, napisati ograničenje, $n \leq 20$.

Rješenje:

U ovome je zadatku najveći problem izbor stanja. Možemo odlučiti da ćemo raditi tako da u svakom koraku uzmemo jednog čovjeka i dodijelimo mu jedan od slobodnih poslova. Najefikasnije će biti ako ljude obrađujemo po unaprijed definiranom redosljedu, npr. od prvog do zadnjeg (od nultog do $(n-1)$ -og). Tada su u svakom momentu potrebna dva podatka: čovjek kojem trenutno dodjeljujemo posao, i skup poslova koji su dodijeljeni ljudima prije njega.

Po prvi put imamo situaciju da nam je za stanje potreban skup, točnije podskup zadanih poslova. Pamćenje bilo kakve STL strukture bilo bi gubljenje memorije i znatno bi zakompliciralo implementaciju. Budući da nam je čitav skup poslova zadan na početku, možemo koristiti tzv. bitmaske, koje omogućuju predstavljanje podskupa kao jednog jedinog broja.

Objašnjenje bitmaski započet ćemo ovako: Zadan nam je neki skup S , za konkretan primjer uzmimo da ima tri elementa.

$S[2], S[1], S[0]$

(Obrnuti poredak je namjerno napravljen.)

Sad pokušajmo enumerirati sve podskupove zadanog skupa, i to na poseban način:

0. podskup: $\{ \}$, prazan skup
1. podskup: $\{ S[0] \}$
2. podskup: $\{ S[1] \}$
3. podskup: $\{ S[1], S[0] \}$
4. podskup: $\{ S[2] \}$
5. podskup: $\{ S[2], S[0] \}$
6. podskup: $\{ S[2], S[1] \}$
7. podskup: $\{ S[2], S[1], S[0] \}$

Zašto se bira baš ovaj način? Pogledajmo brojeve podskupova u binarnom:

000: $\{ \}$
001: $\{ S[0] \}$
010: $\{ S[1] \}$
011: $\{ S[1], S[0] \}$
100: $\{ S[2] \}$
101: $\{ S[2], S[0] \}$
110: $\{ S[2], S[1] \}$
111: $\{ S[2], S[1], S[0] \}$

Prema tome, vrlo je lako napraviti bijekciju između broja podskupa u našem kodiranju i samog podskupa.

Podskup s brojem x u našem kodiranju sadržavat će one elemente početnog skupa na čijim mjestima x u binarnoj bazi ima jedinice. Tako će u skupu s četiri elementa $\{ S[3], S[2], S[1], S[0] \}$ podskup broj 8 (1000 u binarnom) sadržavati samo $S[3]$, podskup 15 (1111 u binarnom) predstavlja cijeli skup. Podskup 3 će, kao i u prvom primjeru, sadržavati $S[1]$ i $S[0]$. Pri tome, broj podskupa u kodiranju nazivamo **bitmaskom** odgovarajućeg podskupa.

Operacije s bitmaskama:

Valja spomenuti i nekoliko temeljnih operacija/funkcija s bitmaskama.

1) Veličina podskupa:

Kardinalni broj podskupa bit će jednak broju jedinica u bitmaski. Za to koristimo funkciju `__builtin_popcount(x)` koja vraća broj jedinica u x .

2) Unija podskupova:

Uniju podskupova lako je dobiti operacijom ILI po bitovima (bitwise OR) nad bitmaskama tih podskupova. Rezultat te operacije je broj čiji je svaki bit jednak vrijednosti disjunkcije odgovarajuća dva bita u brojevima. Npr. $5 | 2 = 7$ ($|$ je oznaka za bitwise OR). Ovo vrijedi jer je 5 u binarnom 101, a 2 je 010. Ako napravimo ILI na svakoj poziciji, dobivamo 111, odnosno 7.

Zašto ovo vrijedi? Pogledajmo dva podskupa. Njihova unija sadržavat će element ako bilo koji od podskupova sadrži taj element. Ako bilo koji od ta dva podskupa sadrži neki element, odgovarajuća će bitmaska imati 1 na tom mjestu, što automatski znači da će i rezultat bitwise OR operacije imati 1 na tom mjestu, a to nadalje znači da će i podskup koji dobivamo kao rezultat sadržavati dotični element.

3) Presjek podskupova:

Presjek se dobiva analogno uniji, i to korištenjem bitwise AND operacije. Rezultat te operacije je broj čiji je svaki bit jednak logičkoj konjunkciji bitova na odgovarajućim pozicijama u brojevima. Za isti primjer kao za uniju, $5 \& 2 = 0$ ($\&$ je oznaka za bitwise AND). Ovo vrijedi jer je 5 u binarnom 101, dok je 2 jednak 010. Prema tome, nakon operacije I na svakom poziciji, dobivamo nulu.

Zašto ovo vrijedi? Jasno je da će rezultat sadržavati jedinicu ako i samo ako obje bitmaske sadržavaju jedinicu na odgovarajućim mjestima. Drugim riječima, rezultatni podskup će sadržavati element ako i samo ako ga sadržavaju oba podskupa.

4) Podskup koji sadrži samo i -ti element:

Lako je vidjeti da je broj podskupa koji sadrži samo i -ti element 2^i , što se u C-u računa kao $(1 \ll i)$.

5) Provjera je li i -ti element u podskupu:

Ovo ćemo napraviti presjekom podskupa koji sadrži samo i -ti element i zadanog podskupa, a zatim usporediti s nulom. Ukoliko je rezultat nula, podskup ne sadrži i -ti element, u suprotnom sadrži.

6) Dodavanje i -tog elementa u podskup:

Napraviti uniju podskupa koji sadrži samo i -ti element i zadanog podskupa.

7) Broj bitmaski i najveća bitmaska:

Za zadani skup veličine n , broj bitmaski će biti jednak 2^n , što znači da će najveća maska biti $(2^n) - 1$ (upravo maska koja ima sve jedinice, tj. označava puni skup).

Napomena: Bitmaske kao metoda dolaze u obzir jedino kada je čitav skup poznat na početku i ne mijenja se.

Vratimo se na početni zadatak. Zaključujemo da ćemo u stanju imati dvije varijable: čovjeka kojem trenutno dodjeljujemo posao te podskup (bitmasku) poslova koji su već dodijeljeni.

Neka nam $f(x, maska)$ kaže koliko minimalno moramo potrošiti u podjeli poslova ljudima od x do n , ako smo ljudima prije x podijelili podskup poslova označen s $maska$.

Krenimo od trivijalnog slučaja, odnosno $x = n$, u tom slučaju će sigurno maska biti jednaka punom skupu jer smo sve poslove već podijelili. Vrijednost funkcije je 0, ne moramo ništa platiti jer smo već gotovi.

Inače, x -tom čovjeku dodjeljujemo neki od slobodnih poslova, te računamo f za sljedećeg čovjeka sa skupom koji je jednak uniji maske i posla koji smo dodijelili x -tom čovjeku.

```
int f(int x, int mask){
    if(x == n)
        return 0;
    if(mem[x][mask] != -1)
        return mem[x][mask];
    int ret = inf;
    for(int i = 0; i < n; ++i){
        if(0 != (mask & (1 << i))) //ovaj posao je netko prije uzeo, idemo dalje
            continue;

        //ako smo prošli if, posao je slobodan i možemo ga dodijeliti x-tom čovjeku
        int rj;

        //dali smo mu i-ti posao, sada krećemo na sljedećeg čovjeka, i kažemo da je
        //skup izabranih poslova jednak skupu ljudi prije x, ali mu još dodajemo,
        //odnosno radimo uniju s poslom koji smo dodijelili x-tom čovjeku.

        rj = f(x+1, mask | (1 << i)) + matrica[x][i];
        ret = min(ret, rj);
    }
    mem[x][mask] = ret;
    return ret;
}
```

Komentar: Broj stanja je $n * 2^n$. To se može poboljšati ako primijetimo da nam broj x u stanju zapravo ne treba, budući da nam broj jedinica maske jednoznačno određuje do kojeg smo čovjeka došli. Prema tome, broj stanja možemo spustiti na 2^n . Prijelaz je složenosti n prema tome, vremenska je složenost $O(n * 2^n)$, dočim memorijska iznosi $O(2^n)$.

Dodatni zadaci i materijali

Dinamičko je programiranje vrlo raširena metoda za rješavanje optimizacijskih problema, stoga nije teško pronaći dobre zadatke na tu temu.

Ovdje ću najprije navesti neke od tutoriala/tekstova koja je svakako preporučljivo proučiti.

- TopCoder tutorial za dinamike:

<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg>

- TopCoder tutorial za bitmaske:

<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=bitManipulation>

- *Napredno programiranje i algoritmi u C-u i C++-u*, Domagoj Kusalić

A sad zadaci:

- Luka Kalinovčić, Zadaci iz dinamičkog programiranja, objavljeno na repozitoriju predmeta

SPOJ (pišu samo imena zadataka, adresa je uvijek www.spoj.pl/IME):

MIXTURES

STREET

TRICOUNT

IVAN

TOURIST

PALIN

z-trening:

premestanje

babuske

dafina

let

lanparty

z-climber