Dinamičko programiranje

Frane Kurtović frane.kurtovic@gmail.com

Martin Gluhak mr.gluhak@gmail.com

Što je dinamičko programiranje?

- Dinamičko programiranje je način rješavanja problema tako da se početni problem rastavi na više jednostavnijih potproblema te se rješenja svakog potproblema koriste da bi se dobilo rješenje početnog problema
- Problem pronaći n-ti Fibonaccijev broj f(n) = f(n-1) + f(n-2)
- Potproblemi su f(n-1) i f(n-2) ➔ njih je sigurno jednostavnije izračunati
- Sada je cilj izračunati problem f(n-1) pomoću potproblema f(n-2) i f(n-3)
- Rekurzivno se dolazi do problema f(0) i f(1) čije rješenje je poznato iz definicije

Rekurzivna implementacija

```
int f(int n) {
  if (n == 0) return 1;
  if (n == 1) return 1;
  return f(n-1) + f(n-2);
}
```

- Eksponencijalna vremenska složenost
- Razlog: funkcija se više puta poziva za istu vrijednost parametra n

Memoizacija

```
int mem[MXN]; // inicijalizirano na -1
int f(int n) {
  if (n == 0) return 1;
  if (n == 1) return 1;
  if (mem[n] != -1) return mem[n];
  return mem[n] = f(n-1) + f(n-2);
}
```

- Linearna složenost jer se svako stanje izračuna točno jednom i to u konstantnom broju operacija

Definicije

- **Stanje** čine parametri rekurzivne funkcije
- Vrijednost funkcije je određena samo pomoću njenog stanja
- **Prijelaz** je rekurzivna relacija koja određuje kako se iz potproblema dobiva rješenje trenutnog problema, npr. $f(n)=f(n-1)+f(n-2)$

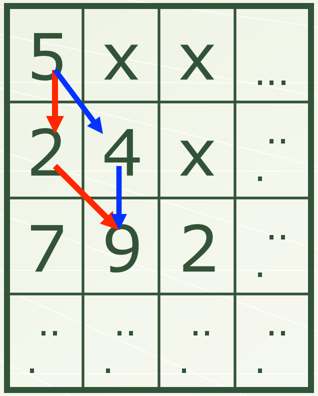
Piramida

- Zadana je kvadratna matrica koja se sastoji od prirodnih brojeva
- Pijun se nalazi u gornjem lijevom kutu te se može kretati jedno polje dolje ili jedno polje dijagonalno dolje-desno
- Cilj je doći do donjeg retka tako da je suma brojeva na putu maksimalna

5	x	x	x
2	4	x	x
7	9	2	x
7	7	6	7

Piramida – rekurzivna relacija

- Paradigma: Dinamičko programiranje!
- Osnovna ideja: na slici desno, je li moguće da put označen crvenom bojom ikada bude dio optimalnog rješenja? Zašto?
- Definirajmo funkciju ***cijena(r, s)*** kao vrijednost najboljeg puta od gornjeg lijevog ruba do pozicije ***(r, s)***.
- ***cijena(r, s) = max{ cijena(r-1, s) + cijena(r-1, c-1) } + A[r][s]***
- Rješenje je ***max{ cijena(n-1, i) za 0 <= i < n }***
- Složenost:
  - Broj bitnih stanja u dinamici: ***1+2+…+n = n(n+1)/2***
  - Vrijeme potrebno za izračunavanje jednog stanja = 1 (konstantno vrijeme, optimalno)
  - Broj operacija: |Stanja|*SloženostPrijelaza ~ ***n(n+1)/2***



Rekurzivno rješenje

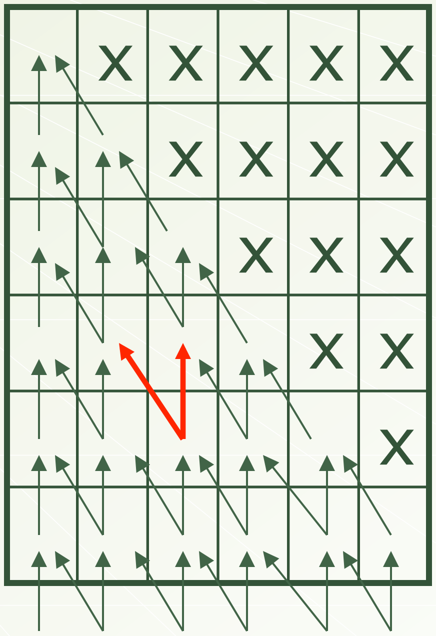
```
int mem[MXN][MXN]; // inicijalizirano na -1
int cijena(int r, int s) {
  if (r == 0) return A[r][s];
  if (mem[r][s] != -1) return mem[r][s];
  int best = cijena(r - 1, s);
  if (s > 0) best = max(best, cijena(r - 1, s - 1));
  return mem[r][s] = best + A[r][s];
}

int sol = 0; // krajnje rješenje
for (int i = 0; i < n; i++)
  sol = max(sol, cijena(n - 1, i));
```

Piramida – iterativno rješenje

- Kojim redoslijedom ćemo izračunavati vrijednosti funkcije ***cijena(r, s)***, tj. kojim redoslijedom rekurzija obliazi stanja?
  - Prije nego što pokušamo izračunati ***cijena(r, s)*** trebamo biti sigurni da su izračunate vrijednosti za ***cijena(r-1, s)*** i ***cijena(r-1, s-1)*** (pod uvjetom da postoje!).
  - Nemamo više funkciju cijena, već samo matricu ***dp[r][s]*** koja ima isto značenje
  - Zapravo direktno popunjavamo memoizacijsku matricu

- Na slici desno vidimo međusobne ovisnosti (**preduvjete**).
- Možemo prvo s lijeva na desno izračunati vrijednosti matrice ***dp*** u prvom retku, zatim opet slijeva nadesno u drugom retku, i tako dalje dok ne dođemo do ***n***-tog retka.
  - Svi preduvjeti se nalaze u prethodnom retku pa je jasno da su svi izračunati.
    - Matematički čistunci mogu slobodno navedeno svojstvo dokazivati indukcijom, unatoč tome što je točnost evidentna



Piramida – iterativno rješenje

```
dp[0][0] = A[0][0];
for (int r = 1; r < n; r++) {
  dp[r][0] = dp[r-1][0] + A[r][0];
  for (int c = 1; c <= r; c++) {
    dp[r][c] = max(dp[r-1][c], dp[r-1][c-1]);
    dp[r][c] += A[r][c];
  }
}
int best = 0;
for (int i = 0; i < n; i++)
  best = max(best, dp[n-1][i]);
```

Zadatak: Ruksak

- Mirko ima ruksak u koji stane maksimalno **N** kila prije nego pukne. Na raspolaganju ima **M** predmeta od kojih svaki ima svoju težinu **Ti** i vrijednost **Vi**. Svakog predmeta ima beskonačno mnogo kopija.
- Ruksak je prazan te u njega treba smjesiti predmete tako da zbroj vrijednosti svih predmeta bude maksimalan.
- Primjer :
- 6 2 (u ruksak stane 6 kila, a na raspolaganju imamo 2 predmeta)
- 3 4 (predmet težak 3 kila, vrijednosti 4) predmet #1
- 5 7 (predmet težak 5 kila, vrijednosti 7) predmet #2
- Potrebno je ispisati najveći zbroj vrijednosti u ruksaku
- Rješenje : 8 (više se isplati uzeti dva predmeta #1, nego jedan #2)

- Intuitivno će se mnogi sjetiti pohlepnog (greedy) rješenja:
  - stavi najvrijedniji predmet u ruksak koji stane
  - ponovi (sa prostorom ruksaka umanjenim za stavljeni predmet)
- ili varijaciju tog rješenja koja računa omjer cijene i težine
- Važno je primjetiti da takva rješenja **nisu** točna, kao što se vidi već iz priloženog jednostavnog test primjera.
- Za rješavanje ovog zadatka treba razmišljati drugačije, problem se treba svesti na jednostavniji oblik.

- Zamislite da imamo ruksak u koji stane X kila.
- Na raspolaganju imamo tri predmeta Y1 (9kg, 10kn), Y2 (12kg, 13kn) i Y3 (16kg, 18kn)
- Izmislimo funkciju **f** koja nam za f(**a**) vraća najveću vrijednost koju mozemo spremiti u ruksak veličine **a**
- Ako stavimo u ruksak prvi predmet, znači da će vrijednost u ruksaku biti 10 + idealno rješenje za ruksak u koji stane “X-9” kila. Zapisano preko funkcije f(x) = 10 + f(x-9). Isto možemo napisati i za predmete 2 i 3.
- f(x) = 13 + f(x-12) , f(x) = 18 + f(x-16)
- Pod pretpostavkom da znamo točno izračunati f(x-9), f(x-12) i f(x-16) koja od ove tri formule izračuna pravu vrijednost f(x)?
- Naravno najveća, jer tražimo maksimalni zbroj vrijednosti u ruksaku

- Cijelo rješenje se temelji na tome da znamo izračunati rješenje za neki f(x), tako da ga zapišemo kao:
- f(x-**a**) + b , gdje je **a** težina predmeta koji dodamo u ruksak, a **b** vrijednost tog predmeta. Poanta je u tome da se parametar u f(x) stalno smanjuje do broja na kojem je rješenje očito. Koliko je rješenje za f(0)?
- f(0)=0, jer svaki predmet ima težinu
- Koliko je rješenje za f(1), a za f(2)? Računa se preko iste formule koju smo zapisali na prošlom slide-u.
- Jedino treba obratiti pozornost da ne pokušamo staviti predmet koji je teži od preostalog mjesta u ruksaku.

Iterativno rješenje

```
// Prva petlja racuna f(x), za brojeve od 1 do n
    for ( int i=1; i<=n; ++i ){
        // Druga petlja prolazi kroz sve predmete i pokusava
    ih ubaciti u ruksak
            for ( int j=0; j<m; ++j ){
            // Provjeravamo stane li predmet "j" u ruksak
    velicine "i"
                if ( t[j] <= i  ){
                    f[i] = max ( f[i], v[ j ] + f[ i - t[ j ] ] );
                }
            }
    }
    printf ("%d\n", f[ n ] );
```

Iterativno rješenje

```
int f[ 10000 ];
int t[ 10000 ] ,v[ 10000 ];

int main (){
    int n,m;
    scanf ("%d%d",&n,&m);
    for ( int i=0; i<m; ++i ){
        scanf ("%d%d",&t[i],&v[i]);
    }

    // Prva petlja racuna f(x), za brojeve od 1 do n
    for ( int i=1; i<=n; ++i ){
        // Druga petlja prolazi kroz sve predmete i pokusava ih ubaciti u ruksak
        for ( int j=0; j<m; ++j ){
            // Provjeravamo stane li predmet "j" u ruksak velicine "i"
            if ( t[j] <= i  ){
               f[i] = max ( f[i], v[ j ] + f[ i - t[ j ] ] );
            }
        }
    }
    printf ("%d\n", f[ n ] );
    return 0;
}
```

Rekurzivno rješenje

```
int n,m;
int memo[ 10000 ];
bool bio[ 10000 ];
int t[ 10000 ] ,v[ 10000 ];

int f ( int x ){
    if ( x == 0 ) return 0;
    if ( bio[ x ] == 1 ) return memo[ x ];
    for ( int i=0; i<m; ++i ){
        if ( t[i] <= x ) memo[ x ] = max( memo[ x ], v[i] + f( x - t[i] )  );
    }
    bio[ x ] = 1;
    return memo[ x ];
}

int main (){
    scanf ("%d%d",&n,&m);
    for (int i=0;i<m;++i){
        scanf ("%d%d",&t[i],&v[i]);
    }
    printf ("%d\n", f( n ) );
    return 0;
}
```

Palindrom

- Svaka riječ se može rastaviti na palindrome (banana – b anana, abbabbaab – abba bb aa b, ...). Na koliko se najmanje dijelova mora podijeliti zadana riječ, a da je svaki dio palindrom.
- Neka je ***A*** oznaka za zadanu riječ.

- Neka je ***P(l, r)*** najmanji broj dijelova na koji se može podijeliti zadana podriječ koja počinje znakom ***A[l]*** i završava s ***A[r]***.

- Pogledajmo koje se sve situacije mogu dogoditi.
- Ako je ***A*** palindrom (što se lako provjeri) onda je rezultat 1, a ako nije onda riječ možemo podijeliti na dva dijela i za ta dva dijela izračunati optimalan rastav i zbrojit ih. Znači da ovaj problem znamo podijeliti na 2 manja problema, ali istog tipa. Npr. Riječ 'banana' možemo rastaviti na 'ban' i 'ana' i onda izračunati optimalan rastav za 'ban' i 'ana'. Još samo treba provjeriti koji od rastava riječi na dvije je najbolji, jer je b i anana bolji od ban i ana.
- ***P(l, r) = 1*** ako je podriječ ***A[l…r]*** palindrom
P(l, r) = min{P(l, k) + P(k+1, r); l ≤ k < r}
- Kako implementirati iterativno?

Palindrom kod

```
int mem[MXN][MXN]; // inicijalizirano na -1
int f(int l, int r) {
  if (palindrom(l, r)) return 1;
  if (mem[l][r] != -1) return mem[l][r];
  int ret = r - l + 1;
  for (int i = l; i < r; i++) {
    ret = min(ret, f(l, i) + f(i + 1, r));
  }
  return mem[l][r] = ret;
}
```

Još zadataka

- Zadana je pravokutna matrica u kojoj su neka polja prohodna, a neka neprohodna. Figura se nalazi u gornjem lijevom kutu i treba doći do donjeg desnog kuta kretajući se samo dolje ili desno. Na koliko načina je moguće doći do donjeg desnog kuta?

- Što ako dodamo uvjet da smije proći kroz najviše 10 prepreka?

Još zadataka

- Čokolada se sastoji od kvadratića u ***n*** redaka i ***m*** stupaca. Mirko može čokoladu presjeći (ne nužno popola) između nekog redka ili stupca. Znači, on može čokoladu od 3*7 prepoloviti u jednu od 3*3 i jednu od 3*4. Ili čokoladu od 6*7 u jednu od 2*7 i jednu od 5*7. Mirko želi imati samo **kvadratne** čokolade. Koliki je minimalan broj presjecanja čokolade da bi svi djelovi bili kvadrati?

  Npr. čokolada 6*9, se može prepoloviti na 6*6 i 6*3, a 6*3 na 3*3 i 3*3. Dakle za čokoladu 6*9 su potrebna 2 presjecanja.

Dodatni materijali

- Skripta iz dinamika na stranicama predmeta
- http://help.topcoder.com/data-science/competing-in-algorithm-challenges/algorithm-tutorials/dynamic-programming-from-novice-to-advanced/

Hvala na pažnji !

Ne zaboravite popuniti anketu

Pitanja?