Dinamicno programiranje

December 18, 2024

1 Dinamično programiranje

Dinamično programiranje je algoritmičen pristop, ki je podoben pristopu deli in vladaj. Tudi pri uporabi dinamičnega programiranja bomo razbili problem na manjše podprobleme, poiskali optimalne rešitve podproblemov in si z njimi pomagali pri rešitvi začetnega problema. Pomembne lastnosti problema, pri katerem si lahko pomagamo z dinamičnim programiranjem so:

- neodvisnost podproblemov: Posamezen podproblem lahko rešujemo neodvisno od drugih podproblemov.
- optimalna podstruktura: Optimalna rešitev problema vsebuje optimalne rešitve podproblemov.
- **prekrivanje/ponavljanje podproblemov**: To je glavna lastnost, ki jo bomo izkoristili za izboljšave in v čemer se pristop razlikuje od tehnike deli in vladaj.

Tehniko lahko enostavno povzamemo z nasvetom "ne računaj enakih stvari večkrat", v praksi pa je kljub temu nekoliko bolj zapleteno - kako to doseči, katere stvari sploh so enake, …

Pristop nima nobene veze z dinamično alokacijo pomnilnika. Poimenoval ga je njen avtor Richard Bellman. "Programiranje" se nanaša na reševanje optimizacijskega problema, poodobno kot matematično programiranje/optimizacija. Pridevnik "dinamično" pa se nanaša na različne podprobleme.

[1]:
```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
```

1.1 Fibonaccijevo zaporedje

Osnovno idejo dinamičnega programiranja si oglejmo na trivialnem primeru Fibonacijevega zaporedja, ki je definirano rekurzivno kot: $F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2}$. Zanima nas n-to število v zaporedju. Pri večjih n-jih bodo vrednosti zaporedja precej velike, vendar se s tem ne bomo ukvarjali in bomo zadovoljni z rezultatom, ki je posledica preliva (*overflow*).

[2]:
```
int fib(int n) {
    if (n<=1) return n;
    return fib(n-1)+fib(n-2);
}
```

[3]:
```
for (int n=0;n<10;n++) {
    cout << n << ": " << fib(n) << endl;
}
```

```
0: 0
1: 1
2: 1
3: 2
4: 3
5: 5
6: 8
7: 13
8: 21
9: 34
```

Vrednosti izgledajo pravilne. Hitro pa ugotovimo, da na ta način ne bomo mogli računati vrednosti že za malo večje n-je. Težava je v eksponentni velikosti drevesa rekurzivnih klicev. Listov tega drevesa, kjer je rezultat funkcije 1, je natanko F_n. Poleg tega pa imamo še liste z vrednostjo 0 in vsa notranja vozlišča. Skratka, ogromno število vozlišč oz. klicev funkcije.

[4]:
```
//cout << fib(100) << endl;  // prepocasi
```

Opazimo lahko, da se bo funkcija izvedla večkrat z istim argumentom n. Če se nismo kje zmotili, bi moral imeti vsak tak klic funkcije tudi enak rezultat. Rezultat si lahko ob prvem klicu funkcije shranimo, v kasnejših klicih pa ga samo vrnemo. To je pristop **od zgoraj navzdol** (*top-down*), ki je znan tudi pod imenom **memoizacija** (*memoization*, brez "r"). Funkcija se bo torej za vsak možen argument izvedla natanko enkrat, ob ostalih klicih pa bo takoj vrnila vrednost, česar niti ne bomo šteli kot klic funkcije. Število klicev funkcije bo torej $O(n)$, čas izvedbe posameznega klica funkcije pa $O(1)$. Rešitev ima časovno in prostorsko zahtevnost $O(n)$.

Za ugotavljanje, ali je bil nek podproblem že rešen ali ne, lahko v tem primeru izkoristimo kar vrednost 0, saj bomo kot izračunane rezultate vpisovali samo večja števila. V splošnem pa bi lahko imeli eno tabelo, ki bi nam povedala, ali je bil nek podproblem že rešen, ter drugo tabelo, ki bi hranila dejanske rezultate. Zaradi enostavnosti bomo uporabili dovolj veliko fiksno tabelo dovolj. Namesto tega bi lahko uporabili katerokoli implementacijo slovarja, ki bi imel kot ključ argumente, ki predstavljajo opis podproblema, za pripadajočo vrednost pa njegovo rešitev.

[5]:
```
const int N=10000;
int memo[N+1];  // memoizacijska tabela
```

[6]:
```
int fib2(int n) {
    if (n<=1) return n;
    if (memo[n]!=0) return memo[n];
    memo[n]=fib2(n-1)+fib2(n-2);
    return memo[n];
}
```

Če smo malo bolj sistematični, lahko rešujemo podprobleme v takem vrstnem redu, da imamo rešitve manjših podproblemov vedno že rešene, ko jih potrebujemo. Podprobleme bomo torej reševali od manjših proti večjim, kar v tem primeru pomeni od manjših proti večjim n-jem. Takemu

reševanju rečemo **od spodaj navzgor** (*bottom-up*). Časovna in prostorska zahtevnost sta enaki kot v prejšnjem primeru, le da sta še bolj očitni.

[7]:
```
int fib3[N+1];
fib3[0]=0;
fib3[1]=1;
for (int n=2;n<=N;n++) fib3[n]=fib3[n-1]+fib3[n-2];
cout << fib3[100] << endl;  // overflow
cout << fib3[10] << endl;
```

```
-980107325
55
```

Zaradi sistematičnosti pa smo lahko malo bolj prostorsko učinkoviti. Vedno namreč potrebujemo rezultate samo zadnjih dveh izračunanih problemov. Tako lahko prostorsko zahtevnost zmanjšamo na $O(1)$.

[8]:
```
int fib4(int n) {
    int f2=0, f1=1;
    for (int i=2;i<=n;i++) {
        int fi=f1+f2;
        f2=f1;
        f1=fi;
    }
    return f1;
}
```

[9]:
```
cout << fib4(10) << endl;
```

```
55
```

1.2 Žabji skoki

Vzdolž potoka gleda iz vode n skal na koordinatah $x_1 < x_2 < ... < x_n$. Žabec sedi na prvi skali in bi rad z zaporedjem skokov po skalah prispel do zadnje skale. V enem skoku lahko skoči najmanj a in največ b enot daleč v smeri proti cilju. Kakšno je najmanjše število skokov, ki jih potrebuje za to?

Če je $a = 0$, smo že v poglavju o požrešnih algoritmih na podobnem problemu ugotovili, da lahko z vsakim skokom skoči do najbolj oddaljene skale, ki jo še doseže, in bo s tem minimiziral število svojih skokov. Vpeljava spodnje meje dolžine skoka pa problem zakomplicira.

Če razmišljamo rekurzivno, se bo žabec v prvem skoku premaknil na neko skalo x_i, ki je oddaljena med a in b od skale x_1. Če take skale sploh ni, pot do cilja ne obstaja. Za to je porabil en skok, nato pa se mora v čim manjšem številu skokov premakniti s skale x_i do cilja. Definirajmo podproblem $f(i)$ kot najmanjše število skokov, ki ga žabec potrebuje, da pride na cilj z i-te skale:

- $f(n) = 0$
- $f(i) = \min_{j>i:\ a \le x_j - x_i \le b} (1 + f(j))$

Očitno bo prišlo do ponavljanja podproblemov. Do neke skale lahko žabec pride na več načinov,

ampak za optimalno pot od tam do cilja je povsem nepomembno, kako je do tja prišel. Pomembno je samo, na kateri skali se nahaja. Zato si lahko rešitev shranimo in jo kasneje po potrebi uporabimo, ne da bi jo računali ponovno. Lahko pa bi probleme reševali tudi sistematično po principu od spodaj navzgor, kar v tem primeru pomeni od skal bližje cilju proti tistim bližje začetku.

Rešiti moramo $O(n)$ podproblemov, za rešitev vsakega od njih pa moramo preveriti $O(n)$ možnosti za naslednji skok. Časovna zahtevnost je $O(n^2)$, prostorska pa $O(n)$.

[10]:
```
const int inf=1e9;
int a=3, b=4;
int mem_jump[1000];
```

[11]:
```
int jump(int i, vector<int> &x) {
    int n=x.size();
    if (i==n-1) return 0;
    if (mem_jump[i]!=0) return mem_jump[i];
    int best=inf;
    for (int j=i+1;j<n;j++) {
        int d=x[j]-x[i];
        if (a<=d && d<=b) best=min(best, 1+jump(j,x));
    }
    mem_jump[i]=best;
    return best;
}
```

[12]:
```
vector<int> x = {0,3,4,6,10};
cout << jump(0,x) << endl;
```

```
3
```

1.3 Rezanje palice

Pri problemu rezanja palice (*rod cutting*) imamo podano palico dolžine n, ki jo želimo razrezati na manjše kose in te kose prodati posamično za čim večjo skupno ceno. Dolžina palice in dolžine kosov morajo biti celoštevilske. Podano imamo tabelo cen c, v kateri nam i-to število c_i pove, za kakšno ceno bomo lahko prodali palico dolžine i. Daljši kot je kos, za večjo ceno ga bomo lahko prodali: veljalo bo $c_i \leq c_{i+1}$. Kakšen je največji možen izkupiček od prodaje razrezane palice?

Oglejmo si primer s spodnjo tabelo cen:

i

1

2

3

4

5

6

7

8

c_i

2

5

6

9

15

16

17

20

Naj bo dolžina palice $n = 8$:

- Če razrežemo palico na kose dolžine 1, bomo zanjo dobili $n \cdot c_1 = 16$.
- Če pustimo palico celo, dobimo zanjo $c_8 = 20$.
- Če jo razrežemo na dva kose dolžin 2 in 6, pa bomo dobili $c_2 + c_6 = 21$.
- Če jo razrežemo na dva kose dolžin 1, 2 in 5, bomo dobili $c_1 + c_2 + c_5 = 22$.

Rekurzivni razmislek o zaslužku $f(n)$ pri optimalnem rezanju palice dolžine n nam pove, da bomo morali izbrati dolžino prvega reza. Če je palica dolžine n, si moramo izbrati enega od rezov dolžine $x \leq n$ (s čimer zaslužimo c_x) ter optimalno zrezati preostanek palice dolžine $n - x$. Ker ne vemo, katera dolžina reza bo najboljša, rekurzivno preverimo vse. Uporabimo tokrat pristop od spodaj navzgor in izračunajmo zaslužke za vedno daljše palice: $f(n) = \max_{x \leq n} f(n - x) + c_x$.

[13]:
```
vector<int> c = {0,2,5,6,9,15,16,17,20};
int N=8;
int f[1000];
f[0]=0;
for (int n=1;n<=N;n++) {
    f[n]=0;
    for (int x=1;x<=n;x++) {
        f[n]=max(f[n], f[n-x]+c[x]);
    }
}
cout << f[N] << endl;
```

22

Časovna zahtevnost algoritma je $O(n^2)$, prostorska pa $O(n)$.

Razmislimo še o **rekonstrukciji** rešitve. Katere reze je treba narediti, da dosežemo optimalno ceno? Za vsak podproblem poiščemo potezo, ki je vodila do optimalnega rezultata. Druga možnost pa je, da si že ob reševanju podproblema shranimo optimalno potezo: npr. v dodatni tabeli $g(n)$ bi lahko hranili x, pri katerem funkcija $f(n)$ doseže svoj maksimum.

[14]:
```
int n=N;
while (n>0) {
    for (int x=1;x<=n;x++) {
        if (f[n]==f[n-x]+c[x]) {
            cout << x << ": " << c[x] << endl;
            n-=x;
            break;
        }
    }
}
```

```
1: 2
2: 5
5: 15
```

1.4 Pot v mreži

V labirintu višine h in širine w oz. tabeli znakov '.', ki predstavljajo prosto polje in '#', ki predstavlajo blokirano polje, nas zanima, na koliko načinov lahko pridemo iz levega-zgornjega kota v desni-spodnji kot, pri čemer se lahko premikamo samo desno in navzdol. V spodnjem primeru obstajajo tri take poti.

```
.#....
....#.
.#..#.
......
```

Rekurzivno bi problem reševali tako, da bi se s trenutne celice poskusili premakniti desno in navzdol (če sta oba premika sploh možna) in sešteli možne poti do cilja iz nove lokacije (sosednje celice). Dosedanji problemi so imel eno-dimenzionalen opis podproblema, kjer smo podproblem opisali z eno spremenljivko. Tokrat pa podproblem opišemo z dvema dimenzijama - vrstico in stolpcem celice. Če je polje zasedeno ali se nahaja izven mreže, je število poti do cilja enako 0, sicer pa velja $f(i,j) = f(i+1,j) + f(i,j+1)$. Robni pogoj v desnem-spodnjem kotu je $f(h-1,w-1) = 0$.

Podprobleme lahko rešujemo sistematično po vrsticah od spodaj navzgor in znotraj vrstice od desne proti levi. Tako imamo potrebne rešitve podproblemov vsakič že na voljo. Časovna in prostorska zahtevnost sta $O(hw)$.

[2]:
```
vector<string> lab = {".#....",
                      "....#.",
                      ".#..#.",
                      "......"};
int h=lab.size(), w=lab[0].size();
int f[10][10];
memset(f,0,sizeof(f));
for (int i=h-1;i>=0;i--) {
    for (int j=w-1;j>=0;j--) {
        if (i==h-1 && j==w-1) f[i][j]=1;
        else if (lab[i][j]=='#') f[i][j]=0;
```

```
        else f[i][j]=f[i+1][j]+f[i][j+1];
    }
}
cout << f[0][0] << endl;
```

```
4
```

Prostorsko zahtevnost bi lahko izboljšali na $O(w)$, ker pri računanju vrednosti $f(i,*)$ potrebujemo samo že izračunane rezultate desno v isti vrstici $f(i,*)$ in eno vrstico nižje $f(i+1,*)$.

1.5 Najdaljše skupno podzaporedje

Pri problemu najdaljšega skupnega podzaporedja (*longest common subsequence, LCS*) nizov S in T (dolžine n in m), iščemo najdaljši niz $\mathrm{LCS}(S,T)$, ki se pojavi kot podzaporedje (ne nujno podniz) v S in v T. Oglejmo si primer $S = A\bar{B}C\bar{B}\bar{B}D\bar{A}B$ in $T = \bar{B}DC\bar{B}\bar{B}B\bar{A}$, kjer je eno izmed najdaljših skupnih podzaporedij $\mathrm{LCS}(S,T) = BCBA$ dolžine 4.

Drugačen pogled na isti problem je poravnava obeh nizov, da se pri tem čim več znakov ujema.

```
AB CB DAB
 BDCBB A
```

Rekurzivni razmislek je sledeč:

- Če se oba niza začneta z enakim znakom, je ta znak lahko začetek LCS-ja, preostanek pa je LCS za en znak krajših nizov.
- Če se niza razlikujeta v prvem znaku, potem vsaj en od teh dveh znakov ne bo del LCS-ja. Preizkusimo obe možnosti in rešimo problem z nizoma, kjer je en malo krajši.

Naj bo $\mathrm{LCS}(i,j)$ najdaljši skupni podniz nizov $S_iS_{i+1}\ldots S_{n-1}$ in $T_jT_{j+1}\ldots T_{m-1}$:

$$\mathrm{LCS}(i,j) = \max\begin{cases} 1+\mathrm{LCS}(i+1,j+1) & \text{če } S_i = T_j \\ \mathrm{LCS}(i+1,j) \\ \mathrm{LCS}(i,j+1) \end{cases}$$

Robni primeri pa so $\mathrm{LCS}(n,*) = 0$ in $\mathrm{LCS}(*,m) = 0$.

Problem lahko rešujemo sistematično od večjih proti manjšim i-jem in enako za j. Rešujemo torej probleme z vedno daljšimi priponami nizov S in T. S tem pravzaprav izpolnjujemo 2D tabelo od desnega spodnjega kota proti levemu zgornjemu, tako da izberemo večjo od spodnje in desne celice. Če sta začetna znaka enaka, pa upoštevamo še diagonalen rezultat povečan za 1. Dokažemo lahko tudi, da bo ta diagonalna poteza vedno optimalna, če je na voljo.

[18]:
```
string LCS(string s, string t) {
    int n=s.size(), m=t.size();
    int lcs[n+1][m+1];  // dodatna vrstica in stolpec nicel
    memset(lcs,0,sizeof(lcs));
    for (int i=n-1;i>=0;i--) {
        for (int j=m-1;j>=0;j--) {
            lcs[i][j]=max(lcs[i+1][j], lcs[i][j+1]);
            if (s[i]==t[j]) lcs[i][j]=max(lcs[i][j], 1+lcs[i+1][j+1]);
        }
```

```
    }
    // izpis izracunane tabele
    for (int i=0;i<n;i++) {
        for (int j=0;j<m;j++) {
            cout << lcs[i][j] << '\t';
        }
        cout << endl;
    }
    // rekonstrukcija
    string l="";
    int i=0, j=0;
    while (i<n && j<m) {
        if (lcs[i][j]==lcs[i+1][j]) i++;
        else if (lcs[i][j]==lcs[i][j+1]) j++;
        else { l+=s[i]; i++; j++; }
    }
    return l;
}
```

[19]:
```
string l = LCS("ABCBDAB", "BDCBBA");
cout << "LCS = " << l << endl;
```

```
4       3       3       3       2       1
4       3       3       3       2       1
3       3       3       2       2       1
3       2       2       2       2       1
2       2       1       1       1       1
1       1       1       1       1       1
1       1       1       1       1       0
LCS = BCBA
```

Časovna in prostorska zahtevnost sta $O(nm)$. Problem lahko rešujemo tudi v obratni smeri od konca proti začetkom nizov, kjer se vprašamo, kaj se bo zgodilo z zadnjima znakoma obeh nizov (namesto prvima), kar boste pogosto videli v drugih virih.

Kako pa bi problem rešili za tri nize? $\mathrm{LCS}(S,T,U)$ namreč ni enak $\mathrm{LCS}(\mathrm{LCS}(S,T),U)$! Stanje bi opisali s trojico indeksov $\mathrm{LCS}(i,j,k)$ in obravnavali primere podobno kot za dva niza. Če velja $S_i = T_j = U_k$, je ta znak lahko del LCS-ja, sicer pa vsaj en izmed njih ne bo in lahko enega od nizov skrajšamo.

Soroden problem je iskanje najdaljšega skupnega podniza (ne podzaporedja; *longest common substring*), kjer mora biti pojavitev podniza strnjena v obeh nizih. Ta problem ima drugačne in bolj učinkovite rešitve.

1.6 Nahrbtnik

Problem nahrbtnika (*knapsack, backpack*) je še en klasičen primer uporabe dinamičnega programiranja. Podan imamo nabor n predmetov, za katere poznamo njihove teže t_i in vrednosti v_i (oboje so cela števila). Izbrali bi radi neko podmnožico S teh predmetov, ki bo imela čim večjo vrednost ($\sum_{j\in S} v_j$) in jih bomo lahko spravili v nahrbtnik z nosilnostjo T ($\sum_{j\in S} t_j \leq T$). Problemu se

natančneje reče 0-1 nahrbtnik, ker vsak predmet vzamemo v celoti ali pa ga pustimo, ne moremo pa vzeti samo dela predmeta.

V rekurzivni rešitvi bi se lahko za vsak predmet odličili, ali ga bomo vzeli ali ne. Če ga vzamemo, imamo za preostale predmeta na voljo nekoliko manjšo nosilnost. Podproblem torej opišemo z dvema atributoma.

- Nabor predmetov, za katere se moramo še odločiti, kaj bomo z njimi. Če smo sistematični, se lahko o vključenosti predmetov odločamo po vrsti od prvega do zadnjega.
- Nosilnost nahrbtnika, ki je na voljo za preostale predmete.

Naj bo $f(i, x)$ največja vrednost, ki jo lahko dobimo v nahrbtniku z nosilnostjo x, če lahko vanj dodajamo predmete $i, i+1, \ldots, n$. Obravnavamo dva primera, glede na (ne)uporabo i-tega predmeta. Robni primer je $f(n, *) = 0$ (če nam zmanjka predmetov, lahko dobimo samo vrednost 0).

$$f(i,x) = \max \begin{cases} f(i+1,x) & \text{ne uporabimo i-tega predmeta} \\ f(i+1,x-t_i)+v_i & \text{če je } t_i \leq x\text{, lahko uporabimo i-ti predmet} \end{cases}$$

Časovna zahtevnost je $O(nT)$. Če nimamo meje za T, vemo, da teža predmetov ne bo presegla $\sum t_i$. Ta rešitev z dinamičnim programiranjem izkorišča majhne celoštevilske teže predmetov in nosilnost nahrbtnika. Če bi bile teže in vrednosti neka realna števila, postane problem izrazito težji (NP-težek). V tem primeru imajo različne kombinacije predmetov različne teže in vrednosti, zato se nam podproblemi ne bi ponavljali. V primeru celih števil pa so bile te vrednosti samo z omejenega intervala celih števil. Čeprav obstaja $O(2^n)$ podmnožic, je na razpolago samo $O(T)$ različnih nosilnosti nahrbtnika.

[8]:
```
const int n = 4;
const int nosilnost = 40;
vector<int> teza     = {30,10,40,20};
vector<int> vrednost = {10,20,30,40};

int f[n+1][nosilnost+1];
memset(f,0,sizeof(f));
for (int i=n-1;i>=0;i--) {
    for (int x=0;x<=nosilnost;x++) {
        f[i][x] = f[i+1][x];  // ne uporabimo i-tega predmeta
        if (teza[i]<=x) {  // poskusimo uporabiti i-ti predmet
            f[i][x] = max(f[i][x], vrednost[i]+f[i+1][x-teza[i]]);
        }
    }
}
cout << f[0][nosilnost] << endl;
```

```
60
```