Osnove dinamičnega programiranja

Filip Koprivec

Fakulteta za Matematiko in Fiziko, IMFM

December 2021

Dinamično programiranje

- Na začetku je bilo uporabljeno za ročno planiranje (1950+)

Dinamično programiranje

- Na začetku je bilo uporabljeno za ročno planiranje (1950+)
- Problem razdelimo na več manjših koščkov (kot pri deli in vladaj)

Dinamično programiranje

- Na začetku je bilo uporabljeno za ročno planiranje (1950+)
- Problem razdelimo na več manjših koščkov (kot pri deli in vladaj)
- Če dovolj časa razbijamo stvari postanejo trivialne (ali pa nerešljive)
- Rešimo vsakega posebej in jih združimo
- Rešitve še manjših problemov se *prekrivajo*

Dinamično programiranje

- Naivno reševanje problema nam navadno prinese eksponentne časovne zahtevnosti ($2^n, n!, \dots$)
- Če znamo problem primerno razbiti navadno dobimo polinomsko časovno zahtevnost (razlika med 30 in 100 točkami)
- Hitro in pravilno opaziti podprobleme terja nekaj vaje.

Fibonačijeva števila

- $F_0 = 1, F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$
- Izračunaj F_k za nek podan k

Naivna implementacija

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

Naivna implementacija

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

Nekaj komentarjev

- Številke bodo velike, skoraj vedno uporabimo `long`

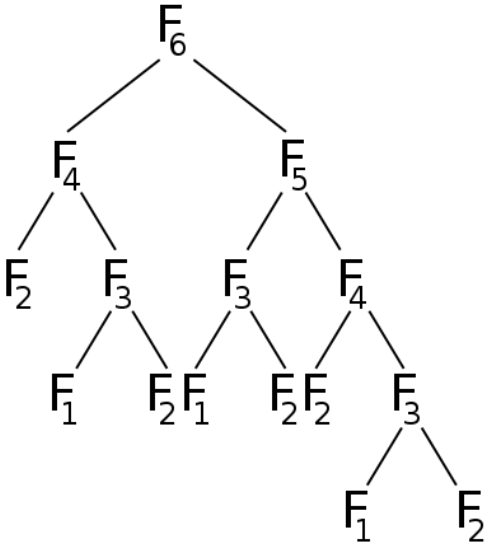
Naivna implementacija

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

Nekaj komentarjev

- Številke bodo velike, skoraj vedno uporabimo `long`
- Veliko dela opravimo večkrat

Ponavljanje $\approx O(1.6^n)$



Tega nočemo

- Enakih stvari nočemo računati večkrat
- Zapomnimo si jih
- memoizacija

Tega nočemo

- Enakih stvari nočemo računati večkrat
- Zapomnimo si jih
- memoizacija
- Kako si jih pametno zapomnimo?

Memoizacija

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

Memoizacija

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

- Lahko bi uporabili `std::map`, ampak je array navadno boljši

Memoizacija

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

- Lahko bi uporabili `std::map`, ampak je array navadno boljši
- Hitro in učinkovito, a porabimo precej več spomina $O(n)$.

Memoizacija

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

- Lahko bi uporabili `std::map`, ampak je array navadno boljši
- Hitro in učinkovito, a porabimo precej več spomina $O(n)$.
- Veliko časa si prihranimo z nekaj malega več spomina.
- Optimalna podstruktura
- O zgoraj navzdol (top down)

Od spodaj navzgor

- Da izračunamo število F_n bomo potrebovali vseh n manjših števil.
- Običajno je bolj učinkovito da tabelo gradimo od spodaj navzgor

```
long table[M];
long fib(int n){
    table[0] = table[1] = 1;
    for(int i = 2; i <= n; ++j)
        table[j] = table[j-1] + table[j-2];
    return table[n];
}
```

Časovna in prostorska zahtevnost: $O(n)$

Izboljšava

```
long fib(int n){
    if(n <= 1) return 1;
    long a = 1;
    long b = 1;
    for(int j = 2; j <= n; ++j){
        long t = b;
        b = a + b;
        a = t;
    }
    return b;
}
```

Časovna in prostorska zahtevnost: $O(n), O(1)$.
Lažje opazno če gradimo navzgor

IOI 1994, Trikotnik

- Vhodni podatek je trikotnik števil
- zanima nas pot od korena do spodnje stranice z največjo vsoto.

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

Kako rešimo problem?

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

- Stojimo na vrhu trikotnika in se sprašujemo kam?

Kako rešimo problem?

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

- Stojimo na vrhu trikotnika in se sprašujemo kam?
- Imamo dve možnosti (načeloma bi jih lahko imeli več)

Kako rešimo problem?

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

- Stojimo na vrhu trikotnika in se sprašujemo kam?
- Imamo dve možnosti (načeloma bi jih lahko imeli več)
- Če najdemo rešitev manjšega problema, ali nam to pomaga?

Kako rešimo problem?

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

- Stojimo na vrhu trikotnika in se sprašujemo kam?
- Imamo dve možnosti (načeloma bi jih lahko imeli več)
- Če najdemo rešitev manjšega problema, ali nam to pomaga?
- Cena najdražje poti je vrednost korena + maksimum poti vseh otrok

Ponavadi posplošimo problem

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

- $C[i,j]$ najdražja pot, ki se začne v $i-$ti vrstici na $j-$tem mestu
- $C[i,j] = T[i,j] + \max(C[i+1,j], C[i+1,j+1])$
- Kje končamo?
- $C[N,] = T[N,\_]$

Ponavadi posplošimo problem

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

- $C[i, j]$ najdražja pot, ki se začne v $i-$ti vrstici na $j-$tem mestu
- $C[i, j] = T[i, j] + \max(C[i + 1, j], C[i + 1, j + 1])$
- Kje končamo?
- $C[N,] = T[N, \_]$
- Ali so vsi pari i, j smiselni?

Ponavadi posplošimo problem

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

- $C[i, j]$ najdražja pot, ki se začne v $i-$ti vrstici na $j-$tem mestu
- $C[i, j] = T[i, j] + \max(C[i + 1, j], C[i + 1, j + 1])$
- Kje končamo?
- $C[N,] = T[N, \_]$
- Ali so vsi pari i, j smiselni?
- Nesmiselnim priredimo tako vrednost, ki nas ne moti (recimo 0).
- Recimo $C[N + 1, \_] = 0$

Analiza?

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

- Na vsakem koraku preverimo dve možnosti: $O(2^N)$, kaj pa prostorska?
- Ali se problemi prekrivajo?

Analiza?

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

- Na vsakem koraku preverimo dve možnosti: $O(2^N)$, kaj pa prostorska?
- Ali se problemi prekrivajo?
- Memoizacija to the rescue!
- $C[i, j]$ si zapomnimo ko ga izračunamo prvič (in ga velikokrat ponovno uporabimo)
- $O(N * (N - 1)/2) = O(N^2)$ tako prostorsko kot časovno

Od spodaj navzgor

```
for(int j = 0; j < N; ++j)
    memo[N][j] = T[N][j];

for(int i = N; i > 0; --i){ // Vrstice
    for(int j = 0; j < i; ++j){ // Stolpci
        memo[i-1][j] = T[i][j] + max(memo[i][j], memo[i][j+1]);
    }
}
```

- Dve for zanki, $O(N^2)$

Ne potrebujemo cele tabele

```
for(int j = 0; j < N; ++j)
    memo[j] = T[N][j];

for(int i = N; i > 0; --i){ // Vrstice
    for(int j = 0; j < i; ++j){ // Stolpci
        memo[j] = T[i][j] + max(memo[j], memo[j+1]);
    }
}
```

- Dve for zanki, $O(N^2)$, porabimo samo $O(N)$ dodatnega prostora (

Splošno

- Rešitev problema je sestavljena iz rešitev podobnih, a manjših problemov
- Včasih je treba problem nekoliko posplošiti
- Napiši si formulo, nariši si odvisnosti

Splošno

- Rešitev problema je sestavljena iz rešitev podobnih, a manjših problemov
- Včasih je treba problem nekoliko posplošiti
- Napiši si formulo, nariši si odvisnosti
- Rešimo podprobleme in iz njih sestavimo rešitev
- Podproblemi so spet sestavljeni iz podproblemov, le-te se pogosto ponavljajo
- NE RAČUNAMO STVARI VEČKRAT
- Že znane rešitve si zapomnimo (memoizacija)
- Gradimo rešitev od spodaj navzgor (običajno manj direktno, a je lahko hitreje)
- Prostorska in časovna zahtevnost
- Občasno se splača majhne probleme pozabiti, če je računanje poceni, ali pa nam zmanjkuje pomnilnika

Memoizacija ali bottom up

- Načeloma primerljiva
- Memoizacija je običajno bolj direktna
- bottom up vedno porabi ves prostor, če ne potrebujemo vseh podproblemov je lahko memoizacija hitrejša
- Pravilni vrsti red pri bottom up

Namigi

- Občasno moramo rekonstruirati optimalno pot
- To si vodimo posebej in ne vodimo celotne poti
- V trikotniku bi si za vsako polje vodili samo zadnji korak (ostale najdemo naprej rekurzivno)
- Številke so lahko velike (!!`long long`!!)
- Pogosto se uporablja modularna artimetika
  $(x + y)\%M = ((x\%M) + (y\%M))\%M$, in enako za $*, -$
- Pri deljenju je potrebno računati modularne inverze (zelo redko)

Coin change

- Imamo množico N kovancev z vrednostmi $a_1, a_2, \ldots, a_N$.
- Radi bi našli najmanjše število kovancev, da dobimo vrednost M, ali sporočimo, da to ne gre
- Ideje od zadnjič?

Coin change

- Imamo množico N kovancev z vrednostmi $a_1, a_2, \ldots, a_N$.
- Radi bi našli najmanjše število kovancev, da dobimo vrednost M, ali sporočimo, da to ne gre
- Ideje od zadnjič?
- Požrešna metoda ne deluje za vse množice kovancev
- $\{1, 4, 9, 10\}, M = 13$, požrešna metoda da $10 + 3 * 1$, bolje je recimo $9 + 4$

Kovanci

- Rešujemo korak za korakom
- Če hočemo dobiti znesek M, si lahko pomagamo s katerimi manjšimi zneski?

Kovanci

- Rešujemo korak za korakom
- Če hočemo dobiti znesek M, si lahko pomagamo s katerimi manjšimi zneski?
- $f(M) = 1 + \min\{f(M - a_i)\}$
- Veliko stvari se ponovi in jih lahko pokrijemo z memoizacijo
- Bottom up: začnemo pri 0 in polnimo kovance

```
memo[_] = inf;
memo[0] = 0;
for(int j = 0; j < M; ++j){
    if(memo[j] = inf) continue; // Tega se ne da doseci
    for(int i = 0; i < N; ++i){
        memo[j + a[i]] = min(memo[j + a[i]], memo[j] + 1);
        // a_i ali ze obstojeca
    }
}
```

Zahtevnost?

Variacije

- 101 variacija, kjer je treba biti previden pri učinkovitosti
- Število kovancev je 1
- Število kovancev je omejeno b_i kovancev z vrednostjo a_i
- ...
- $f(M, kovanciNaVoljo)$ ...
- Pazite na rezultat: TO NI MOGOČE in ga lepo vključite v računanje

Posplošitve

- Časovna zahtevnost je tipično $O(N)$, kjer je N nek parameter - število stvari...
- Če so kakšne druge omejitve majhne lahko podprobleme gradimo na njih (tu se da biti hudo inovativen)

0/1 nahrbtnik

- Vdrli smo banko, kjer smo našli N predmetov z vrednostmi v_i in masami m_i. Radi bi jo pobrali s čim večjo vsoto, a v nahrbtnik lahko spravimo zgolj M kilogramov.
- Direktna rešitev: preverimo vseh 2^N možnosti (zna trajati)

0/1 nahrbtnik

- Vdrli smo banko, kjer smo našli N predmetov z vrednostmi v_i in masami m_i. Radi bi jo pobrali s čim večjo vsoto, a v nahrbtnik lahko spravimo zgolj M kilogramov.
- Direktna rešitev: preverimo vseh 2^N možnosti (zna trajati)
- Omejimo problem: M je relativno majhno celo število, m_i so cela števila

Dinamično

- Dinamično glede na M
- Stojimo na točki m in se odločimo, ali bomo vzeli predmet m_i
- Lahko ga vzamemo in nam ostane $m - m_i$ prostora in dobimo c_i vrednosti
- Lahko ga ne vzamemo in nadaljujemo s predmetom $i + 1$

Dinamično

- Dinamično glede na M
- Stojimo na točki m in se odločimo, ali bomo vzeli predmet m_i
- Lahko ga vzamemo in nam ostane $m - m_i$ prostora in dobimo c_i vrednosti
- Lahko ga ne vzamemo in nadaljujemo s predmetom $i + 1$
- $f(k, c)$ največja vrednost, ki jo lahko ukrademo, če imamo nahrbtnik, ki nosi vsaj c kilogramov in lahko izbiramo izmed prvih k predmetov
- $f(k, c) = \max\{f(k-1, c-m_k) + v_k, f(k-1, c)\}$
- memoizacija, ali pa navzgor po k in c

Dinamično

- Dinamično glede na M
- Stojimo na točki m in se odločimo, ali bomo vzeli predmet m_i
- Lahko ga vzamemo in nam ostane $m - m_i$ prostora in dobimo c_i vrednosti
- Lahko ga ne vzamemo in nadaljujemo s predmetom $i + 1$
- $f(k, c)$ največja vrednost, ki jo lahko ukrademo, če imamo nahrbtnik, ki nosi vsaj c kilogramov in lahko izbiramo izmed prvih k predmetov
- $f(k, c) = \max\{f(k-1, c-m_k) + v_k, f(k-1, c)\}$
- memoizacija, ali pa navzgor po k in c
- Če gremo pametno, potrebujemo samo zadnjo vrstico $k-$jev
- $O(n * M)$

Posplošitve

- Kaj če M ni celo število -> skupni imenovalec
- ponavljanja predmetov, omejitve, izpis najboljše množice...

Najdaljše naraščajoče podzaporedje

- Imamo zaporedje $a_1, a_2, \ldots, a_n$
- Iščemo najdaljše (lahko ima luknje) podzaporedje a_i, ki je strogo naraščajoče
- Želimo najti zaporedje indeksov $i_1 < i_2 < \cdots < i_m$, da velja $a[i_1] < a[i_2] < \cdots < a[t_m]$
- Naivna rešitev $O(2^n)$

Reševanje

- Stojimo na a_j in se sprašujemo kaj naj naredimo
- a_j lahko nekatera podzaporedja, ki se končajo pred j podaljša za 1.
- Kaj nas o takih zaporedjih zanima in koliko jih je?

Reševanje

- Stojimo na a_j in se sprašujemo kaj naj naredimo
- a_j lahko nekatera podzaporedja, ki se končajo pred j podaljša za 1.
- Kaj nas o takih zaporedjih zanima in koliko jih je?
- Zanimajo nas njihovi elementi (v resnici samo zadnji) -> največ j
- v $d[i]$ si zapomnimo najdaljše naraščajoče podzaporedje, ki se konča z a_i
- Če je trenutni element večji lahko zaporedje podaljšamo za 1
- $d[i] = \max_{k<i; a[k]<a[i]}\{d[j] + 1\}$
- $O(n^2)$

Boljše

- $d[i]$ naj bo element a, s katerim se konča podzaporedje dolžine i

```
for (int i = 0; i < n; i++) {
    for (int j = 1; j <= n; j++) {
        if (d[j-1] < a[i] && a[i] < d[j])
            d[j] = a[i];
    }
}
```

- d je naraščajoč
- Vsak korak i bo popravil največ en element v d

Boljše

- $d[i]$ naj bo element a, s katerim se konča podzaporedje dolžine i

```
for (int i = 0; i < n; i++) {
    for (int j = 1; j <= n; j++) {
        if (d[j-1] < a[i] && a[i] < d[j])
            d[j] = a[i];
    }
}
```

- d je naraščajoč
- Vsak korak i bo popravil največ en element v d
- Bisekcija

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int j = upper_bound(d.begin(), d.end(), a[i]) - d.begin()
        if (d[j-1] < a[i] && a[i] < d[j])
            d[j] = a[i];
    }

    int ans = 0;
    for (int i = 0; i <= n; i++) {
        if (d[i] < INF)
            ans = i;
    }
    return ans;
}
```