# Dinamično programiranje

O dinamičnem programiranju, Verižno množenje matrik, Nahrbtnik

Tomaž Dobravec, Algoritmi in podatkovne strukture 2

# O dinamičnem programiranju

# Definicija in osnovni primeri

- Dinamično programiranje je pristop za reševanje problemov.
- Osnovni princi je podoben rekurzivnemu razcepu: nalogo razbijemo na več podnalog in iz rešitev podnalog sestavimo rešitev prvotne naloge.
- Razlika: pri rekurzivnem razcepu so bile naloge med seboj neodvisne, pri dinamičnem programiranju pa so lahko odvisne

Primer:

- Hitro urejanje  z rekurzivnim razcepom.
- Dinamično programiranje: računanje fibonaccijevih števil:  $f(n) = f(n-1) + f(n-2)$.

- Ker so podnaloge med seboj odvisne, je smiselno, da njihove rešitve **shranjujemo**. Ko bomo pri raševanju drugih podnalog naleteli na zahtevo po rešitvi shranjene podnaloge, bomo namesto (rekurzivnega) računanja uporabili shranjeno vrednost.

# Fibonaccijava števila

- Matematična definicija: : $f(n) = f(n-1) + f(n-2)$, $f(0)=f(1)=1$

Rekurzivna rešitev:

```
fib(n):
  if (n==0) || (n==1)
    result=1;
  else
    result=fib(n-1) + fib(n-2)
  return result;
```

Računanje `fib(5)` po rekurzivnem postopku:

# Fibonaccijava števila - časovna zahtevnost rekurzivnega postopka

- **Časovna zahtevnost rekurzivnega postopka?**

# Fibonaccijava števila – dinamično programiranje

- Pri rekurzivnem računanju nismo uporabili dejstva, da so podnaloge odvisne
- Ob upoštevanju tega dejstva dobimo dinamično rešitev:

```
fib_stevila[*]= nedefinirano; // tabela nedef. vrednosti
fibM(n):
  if (fib_stevila[n] == nedefinirano) {
    if (n==0) || (n==1)
      result = 1;
    else
      result = fibM(n-1) + fibM(n-2);
    fib_stevila[n] = result;
  }
  return fib_stevila[n];
```

Računanje `fib(5)` po dinamičnem postopku:

# Fibonaccijava števila - časovna zahtevnost dinamičnega postopka

- Časovna zahtevnost računanja fibM(n):
  - po levi veji se ”sprehodimo” do lista (ki je na globini n);
  - na poti navzgor opravljamo operacije + (**oba operanda sta že izračunana)** in shranjevanje v tabelo;
  - za celoten izračun potrebujemo **n** seštevanj in **n** shranjevanj v tabelo.
  - $\rightarrow T_{fibM}(n) = \Theta(n)$.

- Prostorska zahtevnost: potrebujemo $\Theta(n)$ dodatnega prostora.

- Uporabljena tehnika:
  - **memoizacija** (pomnenje),
  - princip “**od-zgoraj-navzdol**” (top-down).

# Fibonaccijeva števila – dinamično programiranje – obrnjen pristop

- Obrnjen pristop:
  - "**od spodaj-navzgor**" (bottom-up),
  - rezultate shranjujemo s **tabeliranjem.**

```
fibT(n):
  fib_stevila[0]=fib_stevila[1] = 1;
  for i=2…n
    fib_stevila[i] = fib_stevila[i-1] + fib_stevila[i-2]
  return fib_stevila[n]
```

**Časovna zahtevnost?**

- gre za eno zanko → $T_{fibT}(n) = \Theta(n)$

**<u>Prednosti in slabosti ene in druge metode:</u>**

- Memoizacijo zelo preprosto sprogramiram.

- Memoizacija je rekurziven, tabeliranje pa iterativen (torej praviloma hitrejši) postopek.
- Pri tabeliranju zapolnim vsa mesta v tabeli, pri memoizaciji pa samo tista, ki jih potrebujem

- Ocenjevanje časovne zahtevnosti za tabeliranje je običajno bolj preprosto.

**Splošnega odgovora na vprašanje, kaj je bolje (hitrejše) – memoizacija ali tabeliranje – ni!**

# Dinamično programiranje - zaključki

- Dinamično programiranje je postopek, pri katerem si rešitve podproblemov shranjujemo, da jih uporabimo pri računanju drugih podproblemov.

- Na ta način vsak podproblem rešimo samo enkrat in tako prihranimo ogromno časa (pri Fibonacciju smo z lahkoto prišli iz eksponentnega algoritma na linearnega).

- Za implementacijo dinamičnega programa potrebujemo dodatni prostor.

- Velikost dodatnega prostora je odvisna od problema, a velikostni red porabljenega prostora nikoli ne more presegati časovnega velikostnega reda:
  - polinomski algoritem ne more napolniti eksponentno veliko podatkov ;
  - linearni algoritem lahko porabi le linearno mnogo dodatnega prostora (smo videli pri Fibonacciju).

# Verižno množenje matrik

# Množenje matrik

- Matriki A in B lahko zmnožimo, če je število stolpcev matrike A enako številu stolpcev B.

Število osnovnih množenj, ki jih opravimo pri množenju AB?

- Recimo, da želimo izračunati produkt matrik $M_1 \cdot M_2 \cdot M_3 \cdot \ldots \cdot M_n$. Kdaj lahko to storimo?

- Množenje matrik ima zelo lepo lastnost → asociativnost

# Množenje matrik

- **Koliko operacij opravim pri izračunu produkta $M_1 \cdot M_2 \cdot M_3 \cdot \ldots \cdot M_n$?**

# Verižno množenje matrik

**Problem:** imamo matrike $M_1, M_2, \ldots, M_n$. Dimenzija matrike $M_i$ je $d_{i-1} \times d_i$. Poišči

- najmanjše možno število množenj, ki jih potrebujemo za izračun produkta $M=M1M2\ldots Mn$ in
- pravilno postavitev oklepajev.

Definicija: **c(i,j)** *:=* najmanjša cena (== najmanjše število elementarnih produktov, ki jih potrebujemo za izračun) produkta matrik od $M_i$ do $M_j$.

Rešitev problema: $c(1,n)$.

Vemo: $c(i,i)=0$ za vsak $i$.

✧ Kako optimalno izračunamo produkt $M_i \ldots M_j$?

- poiskati moramo optimalno "delilno mesto" `k` (kam se najbolj splača postaviti oklepaje),
- na optimalni način izračunati $M_i \ldots M_k$ ter $(M_{k+1} \ldots M_j)$;
- izračunati produkt $Mij=(M_i \ldots M_k)(M_{k+1} \ldots M_j)$ .

Formula za izračun:

# Rekurzivni izračun c(i,j)

Implementacija:

- Časovna zahtevnost:

# Izračun c(i,j) – pristop od-zgoraj-navzdol

- Uporabim memoizacijo → že izračunane podatke bom shranil v tabelo in jih uporabil, ko bo to potrebno.
- Kodo rekurzivnega programa zelo preprosto popravim tako, da dodam tri vrstice (shranjevanje v tabelo in preverjanje prisotnosti rezultata v tabeli).

**Koliko dodatnega pomnilnika potrebujem za to memoizacijo?**

**Časovna zahtevnost?**

# Izračun c(i,j) – pristop od-spodaj-navzgor

- Uporabim tabulacijo in računam vrednosti po vrsti od c(1,1), c(2,2), …c(n,n) navzgor do c(1,n).
- Katere vrednosti c(i,j) dejansko potrebujem?

Pri računanju c(i,j) za vsak k=1…j-1 seštejem:

- element i-te vrstice
- element j-tega stolpca
- produkt $d_{i-1}d_k d_j$

… in si zapomnim minimum.

# Izračun c(i,j) – pristop od-spodaj-navzgor

- Pri računanju elementa c(i,j) potrebujem le elemente, ki so levo in pod c(i,j);
- pravilni vrstni red računanja je po diagonalah;
- rezultat je v zgornjem desnem kotu.

- Primer: množenje matrik `5x4, 4x6, 6x2, 2x7`

- **Časovna zahtevnost pristopa od-spodaj-navzgor?**

## Primer: množenje matrik 5x4, 4x6, 6x2, 2x7

# Postavitev oklepajev

- Sedaj znam izračunati c(1,n) → število potrebnih množenj.
- Kako mi to pomaga pri postavitvi Oklepajev?
- Zanima me, kam moram postaviti oklepaje, da bo število množenj enako c(1,n)!
- Preprosta rešitev: če si na vsakem koraku v dodatni matriki zapomnim, kateri `k` je prispeval minimum, lahko na koncu na podlagi te matrike rekonstruiram celotno postavitev oklepajev.

Primer: postavitev oklepajev pri računanju produkta matrik `5x4, 4x6, 6x2, 2x7`

# Primer: računanje produkta matrik `4x10, 10x3, 3x12, 12x20,20x7`

# Nahrbtnik

# Nahrbtnik

**Osnovni problem:** Imamo `n` predmetov. Velikost `i`-tega predmeta je $v_i$, njegova cena pa $c_i$.
V nahrbtnik velikosti `V` bi radi naložili predmete tako, da bo skupna cena predmetov v nahrbtniku čim večja.

Problem ima dve podvarianti:

a) predmete lahko režemo (ime problema: **Nahrbtnik z rezanjem**)

b) predmetov ne smemo rezati – za vsak predmet imamo dve možnosti: lahko ga damo v nahrbtnik (1) ali pa ga ne damo (0) – od tod ime problema: **0/1 nahrbtnik**.

**Nahrbtnik z rezanjem**

Požrešen pristop:

- za vsak predmet naračunamo njegovo relativno ceno ($t_i = c_i / v_i$),
- predmete uredimo po padajoči vrednosti $t_i$
- v nahrbtnik zlagamo predmete po vrsti, zadnjega odrežemo.

**Časovna zahtevnost:**

# Nahrbtnik z rezanjem - primer

# 0/1 nahrbtnik

Iščemo tak $x \in \{0,1\}^n$, da velja $\sum_i x_i v_i \leq V$ in $\sum_i x_i c_i$ maksimalen.

Problem lahko rešimo tako, da pregledamo vse možne **kombinacije predmetov**.

Uporaba dinamičnega programiranja:

- problem bomo razbili na podprobleme,
- podprobleme bomo reševali po velikosti (pristop od-spodaj-navzgor)
- rešitve podproblemov bomo shranjevali v tabeli.

# 0/1 nahrbtnik – rešitev z dinamičnim programiranjem

**Definiramo**: $k_i(W)$ = vrednost optimalnega nahrbtnika velikosti $W$, če lahko uporabimo le prvih $i$ predmetov.

- **Cilj** (rešitev prvotne naloge): $k_n(V)$
- Kako je naloga $k_i(W)$ odvisna od "manjših" nalog?
- Kaj pa ustavitveni pogoj?
- Vrednosti funkcije ki bomo računali po naraščajočem i:
  začeli bomo s $k_0$, nadaljevali s $k_1$, $k_2$, ...
- Graf funkcije $k_0$ imenujemo **stopnica**
- Funkcija $k_1$ je sestavljena iz $k_0(W)$ in $k_0(W-v_i)+c_i$
  torej iz osnovne in premaknjene stopnice ($-v_i$ pomeni premik stopnice desno, $+c_i$ pa navzgor). Funkcijo, ki jo dobimo iz $k_0$ in premaknjene $k_0$ imenujemo **stopnišče**.

# 0/1 nahrbtnik – rešitev z dinamičnim programiranjem

- Funkcijo $k_i$ sestavimo iz prejšnje $k_{i-1}$ in premaknjene $k_{i-1}$, torej iz dveh stopnišč.
- Maximum dveh stopnišč je spet stopnišče.

- Vogali stopnišča predstavljajo kombinacijo elementov v nahrbtniku.

- Če stopnišče $k_{i-1}$ vsebuje $t$ ”vogalov”, bo tudi premaknjeno stopnišče vsebovalo $t$ vogalov.
- Ko s funkcijo `max` sestavimo novo stopnišče, *načeloma* dobimo stopnišče z `2t` vogali.

# 0/1 nahrbtnik – rešitev z dinamičnim programiranjem

✧ Kdaj je število vogalov manjše od $2t$?

a. Ko vogala originalne in premaknjene funkcije ležita na isti navpičnici;
b. ko vogala originalne in premaknjene funkcije ležita na isti vodoravnici;

c. ko vogal originalne ali premaknjene funkcije leži pod vodoravnico originalnega ali premaknjenega grafa;

✧ <u>Pravilo:</u> če sta $(v_1, c_1)$ in $(v_2, c_2)$ eden vogal originalne, drugi pa vogal premaknjene funkcije in če zanju velja

$$(v_2 \geq v_1) \ \& \ (c_2 \leq c_1)$$

potem vogal $(v_2, c_2)$ v sestavljeni funkciji ne obstaja.

# 0/1 nahrbtnik – postopki reševanja

Postopki za reševanje problema:

A) <u>Na grafični način</u>
- Rišemo stopnišča in spotoma odstranjujemo nepotrebne vogale.
- Ko dodamo vse predmete, je rešitev v najvišjem vogalu končnega stopnišča, katerega `x` koordinata je manjša ali enaka `V`.
- Postopek je primeren za ročno reševanje.

B) Na tabelarični način
- V tabeli shranjujem vse vogale funkcij $k_i(W)$.
- Da prihranimo prostor, lahko shranjujemo le vogale trenutne funkcije $k_i(W)$.
- Postopek je bolj primeren za računalniško reševanje.

# Primer: grafično reševanje problema

# Primer: tabelarično reševanje problema