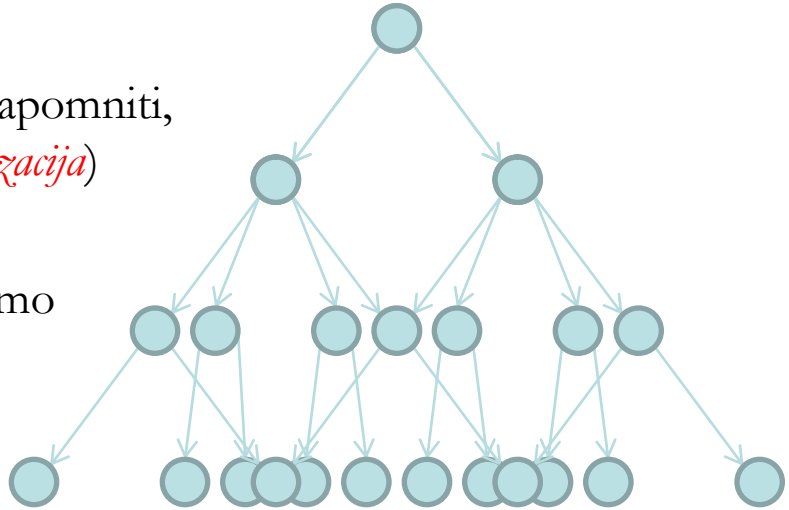
Dinamično programiranje

Janez Brank

Uvod

- Mnoge optimizacijske probleme lahko rešimo tako, da v problemu opazimo manjše podprobleme
  - Vsak podproblem je iste vrste kot prvotni problem, le malo manjši je
    - Rešimo ga tako, da v njem opazimo še manjše podpodprobleme in tako naprej
    - Sčasoma pridemo do trivialno majhnih pod…problemov, ki jih znamo rešiti
  - Rešitev večjega problema znamo izračunati iz rešitev njegovih podproblemov
- O dinamičnem programiranju govorimo v primerih, ko se podproblemi ponavljajo
  - Takrat si je koristno rešitve podproblemov zapomniti, da jih ne bo treba računati po večkrat (*memoizacija*)
  - S tem lahko prihranimo *ogromno* časa
  - Pogosto lahko rešitve podproblemov računamo zelo sistematično, od manjših k večjim
- Izziv je ponavadi:
  - Opaziti, da je naša naloga primerna za reševanje z dinamičnim programiranjem
  - Domisliti se, kako razbiti problem na podprobleme
    - Pri tem moramo problem pogosto malo posplošiti



Dinamično programiranje

Množenje matrik, oklepajski izrazi,
delitev zemljišča

Ena od nalog z južnoameriškega regijskega tekmovanja ACM 2005 (malo okleščena)

- Imamo podolgovato zemljišče, široko 1 enoto in dolgo $x_1 + x_2 + ... + x_n$ enot.
- Radi bi ga z navpičnimi črtami razdelili na n zemljišč, dolgih po $x_1, x_2, ..., x_n$ enot (v tem vrstnem redu).
- Vsakič ko razrežemo zemljišče dolžine $a + b$ na zemljišče dolžine a in zemljišče dolžine b, moramo plačati max(a, b) denarnih enot davka.
- V kakšnem vrstnem redu naj režemo zemljišče?

Primer

- $n = 4, x_1 = 5, x_2 = 1, x_3 = 2, x_4 = 3.$

$11 \rightarrow 8 + 3$ (davek: 8)

$11 \rightarrow 5 + 6$ (davek: 6)

$8 \rightarrow 6 + 2$ (davek: 6)

$6 \rightarrow 3 + 3$ (davek: 3)

$6 \rightarrow 5 + 1$ (davek: 5)

$3 \rightarrow 2 + 1$ (davek: 2)

(davek skupaj: 19)

(davek skupaj: 11)

Problem in podproblemi

- Naš prvi rez razreže zemljišče na dva kosa.
  Recimo, da je levi kos širok $x_1 + x_2 + \ldots + x_k$ enot,
  desni pa $x_{k+1} + x_{k+2} + \ldots + x_n$ enot.
- Zdaj je, kar je; davek bomo plačali; potem pa nam preostane le še to, da vsakega od teh dveh kosov posebej razrežemo čim ceneje.
  - Levi kos nič ne vpliva na desnega in obratno.
  - Torej je vsak od njiju sam zase pravzaprav čisto enak problem kot prvotni, le da je malo manjši:
    namesto na n zemljišč bi radi delili na k (pri levem kosu)
    oz. na $n - k$ (pri desnem kosu)
- Ker pa vnaprej ne vemo, kateri k bo pripeljal do najcenejše rešitve, moramo pač preizkusiti vse.

Rekurzivna rešitev

- Problem smo razbili na podprobleme, ki so mu v vseh pogledih enaki (le da so manjši), tako da se ga je pametno lotiti z rekurzijo.

function *Reši*(x_1, x_2, ..., x_n)
  if $n = 1$ **then return** 0; { robni primer — ni česa rezati }
  MinCena := ∞;
  for $k := 1$ **to** $n - 1$ **do**
    Cena := max($x_1 + ... + x_k$, $x_{k+1} + ... + x_n$)
      + *Reši*(x_1, ..., x_k) + *Reši*(x_{k+1}, ..., x_n);
    if *Cena* < *MinCena* **then** *MinCena* := *Cena*;
  return *MinCena*;

- Vidimo lahko, da je zaporedje, ki ga prenašamo kot parameter, vedno neko podzaporedje prvotnega (x_1, ..., x_n).
  - Zato je v praksi dovolj, če povemo le začetni in končni indeks, celotno zaporedje pa hranimo v neki globalni spremenljivki.

Rekurzivna rešitev

function *Reši*(i, j) { rešuje podzaporedje $(x_i, x_{i+1}, ..., x_{j-1}, x_j)$ }
  if $n = 1$ **then return** 0; { ni česa rezati }
  MinCena := ∞;
  for $k := i$ **to** $j - 1$ **do**
    Cena := $\max(x_i + ... + x_k , x_{k+1} + ... + x_j)$
      $+$ *Reši*(i, k) $+$ *Reši*$(k + 1, j)$;
    if *Cena* < *MinCena* **then** *MinCena* := *Cena* ;
  return *MinCena* ;

- Žal je ta rešitev časovno precej potratna.

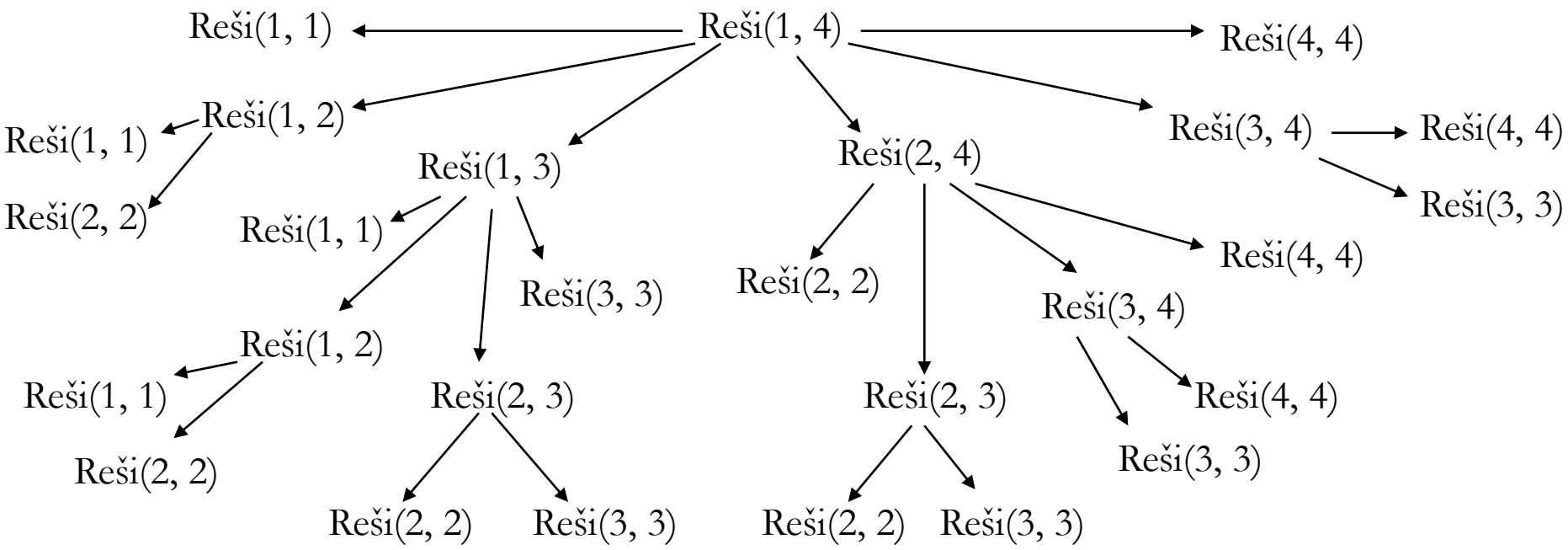
Časovna zahtevnost

- Če je $m := j - i + 1$ dolžina opazovanega podzaporedja, bo *Reši* poklicala po dva klica za podzaporedja dolžine 1, 2, ..., $m - 1$.
- Naj bo $\#_m$ število klicev za podzaporedja dolžine m. Vidimo torej, da je $\#_n = 1$ (glavni klic), nato pa $\#_m = 2(\#_{m+1} + \#_{m+2} + \ldots + \#_n)$.
- $\#_{n-1} = 2(\#_n) = 2$
  $\#_{n-2} = 2(\#_n + \#_{n-1}) = 2(1 + 2) = 6$
  $\#_{n-3} = 2(\#_n + \#_{n-1} + \#_{n-2}) = 2(1 + 2 + 6) = 18$
  $\#_{n-4} = 54, \quad \#_{n-5} = 162, \quad \#_{n-6} = 486, \ldots$
- Če malo potelovadimo z rodovnimi funkcijami, vidimo:
  $\#_{n-k} = 2 \cdot 3^{k-1}$
- Vseh klicev skupaj bo
  $\#_n + \#_{n-1} + \#_{n-2} + \ldots + \#_2 + \#_1 =$ ravno 3^{n-1}.

Časovna zahtevnost

- Če malo potelovadimo z rodovnimi funkcijami, vidimo:
  $$\#_{n-k} = 2 \cdot 3^{k-1}$$
  - Torej $2 \cdot 3^{k-1}$ klicev za podzaporedja dolžine $n - k$.
  - Toda zaporedje dolžine n ima samo $k + 1$ podzaporedij dolžine $n - k$.
  - Torej očitno veliko teh klicev po večkrat obdeluje ena in ista podzaporedja!
- Ideja: ko prvič obdelamo neko podzaporedje, si shranimo rezultat v neko tabelo.
  - Ob kasnejših klicih ga poberemo od tam in ga ni treba ponovno računati.

Rekurzivni klici



Memoizacija (pomnjenje)

for i := 1 **to** n **do for** j := i **to** n **do** $r[i,j]$:= –1; { inicializacija tabele r }

function *Reši*(i, j) { rešuje podzaporedje $(x_i, x_{i+1}, ..., x_{j-1}, x_j)$ }
  if $n = 1$ **then return** 0; { ni česa rezati }
  if $r[i,j] \geq 0$ **then return** $r[i,j]$; { rešitev že poznamo }
  MinCena := ∞;
  for k := i **to** $j - 1$ **do**
    Cena := max$(x_i + ... + x_k , x_{k+1} + ... + x_j)$
      + *Reši*(i, k) + *Reši*$(k + 1, j)$;
    if *Cena* < *MinCena* **then** *MinCena* := *Cena* ;
  $r[i,j]$:= *MinCena*;
  return *MinCena* ;

- *Reši*(i, j) porabi konstantno mnogo časa, razen če se prvič srečuje s parom (i, j).
  - Takrat pa porabi $O(j - i)$ časa, če odmislimo čas za izvajanje rekurzivnih klicev.
- Skupaj imamo torej časovno zahtevnost
  $\Sigma_{k=1}^{n}$ [št. zaporedij dolžine k][čas obdelave zaporedja dolžine k] =
  $\Sigma_{k=1}^{n} (n - k + 1)k = (n^3 + 3n^2 + 2n)/6 = O(n^3)$.

Dinamično programiranje

- Rešitev s prejšnje folije je v bistvu že dinamično programiranje.
- Lahko pa smo še bolj sistematični.
  - Ko se izvajajo naši rekurzivni klici, bomo prej ali slej izračunali rešitev $r[i,j]$ za vse pare (i,j), $1 \leq i \leq j \leq n$.
  - Preden lahko izračunamo $r[i,j]$, potrebujemo vse $r[i',j']$ za $i \leq i' \leq j' \leq j$.
- Tabelo $r[i,j]$ lahko torej polnimo čisto sistematično, od krajših podzaporedij proti daljšim.
  - To nam zagotavlja, da bomo imeli vedno pri roki rešitve podproblemov, ki jih bomo potrebovali za trenutni problem.

Dinamično programiranje

for $i := 1$ **to** n **do** $r[i, i] := 0$; { ni česa rezati }
for $L := 2$ **to** n **do**
  for $i := 1$ **to** $n - L + 1$ **do begin**
    $j := i + L - 1$;
    $r[i,j] := \infty$;
    for $k := i$ **to** $j - 1$ **do**
      $Cena := \max(x_i + ... + x_k , x_{k+1} + ... + x_j)$
        $+ r[i, k] + r[k + 1, j]$;
      if $Cena < r[i,j]$ **then** $r[i,j] := Cena$;
  end;
return $r[1, n]$;

- To je še vedno $O(n^3)$, tako kot prejšnja rešitev.
- Bi pa znala biti v praksi malo hitrejša, ker je zdaj manj overheada z rekurzivnimi klici, knjigovodstvom ipd.

Recept

- V našem problemu opazimo podprobleme, ki so istega tipa kot glavni problem, le da so manjši.
  - Na primer: problem je zaporedje $(x_1, ..., x_n)$, podproblemi so podzaporedja $(x_i, x_{i+1}, ..., x_{j-1}, x_j)$.
  - V bistvu si lahko mislimo, da smo problem malo posplošili. Včasih je treba biti malo zvit, da opaziš primerno posplošitev.
  - Rešitev problema znamo izračunati iz rešitev podproblemov.
- Podproblemi imajo spet svoje podpodrobleme, itd.
  - Opazimo, da se začnejo podpodproblemi, podpodpodproblemi, itd. ponavljati.
  - Zato pazimo, da ne računamo istega podpod...problema po večkrat.
- Rešitev podpod...problema si zapomnimo, ker bo prišla prav še kasneje (memoizacija).
  - Lahko pa tudi sistematično obdelamo vse podpod...probleme od manjših proti večjim.
  - Lahko si tudi zapomnimo, *kako* smo prišli do rešitve. S pomočjo teh podatkov na koncu rekonstruiramo najboljšo rešitev (npr. to, v kakšnem vrstnem redu moramo rezati zemljišče).

Težave

- Če se podpod...problemi ne ponavljajo
  oz. če pač obstaja eksponentno mnogo različnih podpod...problemov, si z dinamičnim programiranjem pač ne moremo pomagati do polinomske rešitve.
- Če je podpod...problemov polinomsko mnogo, jih je lahko še vseeno preveč, da bi hranili vse njihove rešitve v pomnilniku.
  - Odvisno seveda od tega, s kakšnim n imamo opravka.
  - Včasih se da rešitve majhnih pod...problemov sproti pozabljati (npr. če je podproblem velikosti k odvisen le od podproblemov velikosti $k - 1$, ne pa od tistih velikosti $k - 2$ in manjših).

Še nekaj podobnih problemov

- Rad bi zmnožil zaporedje matrik. Kako postaviti oklepaje, da bo skupna cena množenja najmanjša?
- Dan je izraz $x_1 / x_2 / x_3 / \ldots / x_n$, pri čemer so $x_1, \ldots, x_n$ neka znana števila. Postavi oklepaje v ta izraz tako, da bo imel največjo možno vrednost.

Dinamično programiranje

Floyd-Warshall

Floyd-Warshallov algoritem

- Dan je graf $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$, vsaka povezava ima tudi dolžino
- Zanimajo nas dolžine najkrajših poti med vsemi pari točk
  - Seveda lahko npr. poženemo Dijkstro po enkrat za vsako možno začetno točko → $O(VE \log V)$
  - Ali pa Bellman-Forda za vsako možno začetno točko → $O(V^2E)$
  - Ali pa Bellman-Forda z množenjem matrik → $O(V^3 \log V)$
  - Floyd-Warshall pa je $O(V^3)$, kar je še posebej lepo, če je graf gost

Floyd-Warshallov algoritem

- Definirajmo si podproblem: $f(i, j, k)$ = dolžina najkrajše poti od v_i do v_j, ki se med tema točkama sprehaja le po točkah $\{v_1, \ldots, v_k\}$, po ostalih pa ne
- [robni primer] Očitno je $f(i, j, 0)$ kar enaka dolžini povezave (v_i, v_j), oz. je ∞, če take povezave ni
- [rekurzija] Glede $f(i, j, k)$: mogoče najkrajša taka pot gre skozi v_k, mogoče pa ne
  - Če ne, je to kar enako $f(i, j, k - 1)$
  - Če pa gre skozi v_k, je do tam pot dolga $f(i, k, k - 1)$, od tam naprej pa $f(k, j, k - 1)$
- Na koncu imamo v $f(i, j, n)$ dolžino najkrajše poti od i do j sploh
  - Če je kakšna $f(i, i, n) < 0$, vemo, da obstajajo v grafu negativni cikli (in so naši rezultati bolj ali manj numerično nesmiselni)

Implementacija

- Zapišimo ga s psevdokodo:

  for $i = 1$ **to** n **do for** $j = 1$ **to** n **do**:

  $f[i, j, 0]$ = dolžina povezave od i do j

  (oz. ∞, če take povezave ni);

  for $k = 1$ **to** n **do for** $i = 1$ **to** n **do for** $j = 1$ **to** n **do**

  $f[i, j, k] = \min\{f[i, j, k-1], f[i, k, k-1] + f[k, j, k-1]$
- Ko računamo vrednosti $f(\cdot, \cdot, k)$, potrebujemo le $f(\cdot\ , \cdot\ , k-1)$, vse prejšnje pa lahko pozabimo
- Še več, dovolj je že, če imamo eno samo matriko:

  for $k = 1$ **to** n **do for** $i = 1$ **to** n **do for** j $= 1$ **to** n **do**

  $f[i, j] = \min\{f[i, j], f[i, k] + f[k, j]\}$

Podoben problem

- Dan je deterministični končni avtomat, sestavi nek regularni izraz, ki opisuje isti jezik kot ta končni avtomat
  - Avtomat je v bistvu graf s črkami na povezavah
  - Stanja avtomata so točke grafa; eno od stanj je "začetno", nekatera stanja so "končna"
  - Avtomat "sprejme" niz znakov, če obstaja v njem od začetnega do enega od končnih stanj taka pot, na kateri črke povezav tvorijo ravno ta niz
  - Oštevilčimo stanja in si zastavimo podproblem: naj bo $E(i, j, k)$ regularni izraz za jezik vseh tistih nizov, ki naš avtomat pripeljejo od stanja q_i do stanja q_j in pri tem gredo le skozi stanja $\{q_1, \ldots, q_k\}$?
  - Potem imamo
    $E(i, j, k) = E(i, j, \mathrm{k} - 1) \mid E(i, k, k - 1)\, E(k, k, k - 1)^*\, E(k, j, k - 1)$
  - In na koncu za jezik celotnega avtomata, če je q_s začetno stanje in je $F = \{f_1, \ldots, f_r\}$ množica indeksov končnih stanj
    $E = E(s, f_1, n) \mid E(s, f_2, n) \mid \ldots \mid E(s, f_r, n)$

Dinamično programiranje

Razdelitev na podzaporedja
z najmanjšo vsoto

Pisarji

- Naloga (s CERC 1998):
  - Imamo zaporedje n knjig s po d_1, d_2, …, d_n stranmi
  - Imamo p pisarjev
  - Prvi bo prepisal prvih a_1 knjig, drugi naslednjih a_2 knjig, tretji naslednjih a_3 knjig in tako naprej
  - Čas, ki ga nek pisar porabi za svoje delo, je sorazmeren skupnemu številu strani v knjigah, ki jih mora prepisati
    - Vsi začnejo pisati istočasno, vsi delajo enako hitro
  - Kako razdeliti knjige med pisarje, da bo vse končano čim prej?
    - Torej določi a_1, a_2, …, a_p tako, da bo največje število strani (po vseh pisarjih) čim manjše
    - Seveda ob omejitvi $a_1 + a_2 + \ldots + a_p = n$

Rekurzivni razmislek

- Naj bo s_r skupno število strani, ki jih mora prepisati pisar r
  - Torej $s_r = \Sigma_i d_i$ za $a_1 + \ldots + a_{r-1} < i \leq a_1 + \ldots + a_r$
- Recimo, da si nekako izberemo,
  koliko knjig bomo dali zadnjemu pisarju – torej a_p
  - On bo torej pisal s_p časa
  - Ostane nam vprašanje, kako prvih $n - a_p$ knjig razdeliti med preostalih $p - 1$ pisarjev in koliko časa bo v tem primeru pisal najbolj obremenjen pisar med njimi
- Tako smo dobili podproblem: $f(m, r)$ = čas, ki ga porabi najbolj obremenjeni pisar, če delimo prvih m knjig med r pisarjev
  - Če zadnji pisar, torej r, dobi k knjig, porabi
    $s_r = a_{m-k+1} + \ldots + a_m$ časa,
    ostali pisarji pa za ostale knjige v tem primeru porabijo $f(m - k, r - 1)$ časa
  - Torej je $f(m, r) = \min_{0 \leq k \leq r} \max\{a_{m-k+1} + \ldots + a_m, f(m - k, r - 1)\}$.

Implementacija

- Kot ponavadi pri dinamičnem programiranju vidimo, da je $f(m, r)$ sicer definirana rekurzivno, vendar bi pri rekurziji velikokrat obravnavali ene in iste podprobleme (m, r)
- Že izračunane rezultate hranimo v tabeli, lahko jih tudi računamo sistematično po naraščajočih m in r

$f[0, 1] = 0$; **for** $m = 1$ **to** n **do** $f[m, 1] = f[m - 1, 1] + d_m$;
for $r = 2$ **to** p **do**
  $f[0, r] = 0$;
  for $m = 1$ to n **do**
    $f[m, r] = f[m, r - 1]$; $s_r := 0$;
    for $k := 1$ **to** n **do**
      $s_r := s_r + d_{m-k+1}$;
      $kand := \max(f[m - k, r - 1], s_r)$;
      $f[m, r] := \min(f[m, r], kand)$;

- Vidimo tudi, da ko končamo z računanjem $f[., r]$, lahko pozabimo vse rezultate za $f[., r - 1]$, ker jih ne bomo več potrebovali

Drugačna rešitev

- Mimogrede, to nalogo lahko rešimo tudi brez dinamičnega programiranja
  - Vprašajmo se: ali je mogoče razdeliti knjige tako, da noben pisar ne dobi več kot *M* strani?
    - Dajmo prvemu pisarju toliko prvih knjig, kolikor je le mogoče, preden bi presegle *M* strani
    - Nato dajmo drugemu pisarju toliko nadaljnjih knjig, kolikor je le mogoče, preden bi presegle *M* strani
    - Itd.
  - Če nam zmanjka knjig prej kot pisarjev, je razpored z omejitvijo *M* mogoč, sicer pa ne
  - Najmanjši možni *M* poiščimo z bisekcijo

Dinamično programiranje

Najdaljše skupno podzaporedje,
urejevalniška razdalja

Najdaljše skupno podzaporedje

- Dani sta zaporedji $a = a_1\ a_2\ \ldots\ a_n$ in $b = b_1\ b_2\ \ldots\ b_m$
- Iščemo najdaljše skupno podzaporedje (ne nujno strnjeno!)
  - Z drugimi besedami, iščemo take indekse

    $$i_1, \ldots, i_r, j_1, \ldots, j_r,$$

    da bo

    $$1 \le i_1 < i_2 < \ldots < i_r \le n,$$
    $$1 \le j_1 < j_2 < \ldots < j_r \le m,$$
    $$a[i_1] = b[j_1],\ a[i_2] = b[j_2],\ \ldots,\ a[i_r] = b[j_r]$$

    in da bo r čim večji

Rekurzivni razmislek

- Za najdaljše skupno podzaporedje (po definiciji s prejšnje folije) gotovo velja nekaj od naslednjega:
  - Lahko da je $i_1 > 1$. V tem primeru je naše podzaporedje hkrati tudi najdaljše skupno podzaporedje nizov $a_2 \ldots a_n$ in $b_1 \ldots b_m$.
  - Lahko da je $j_1 > 1$. V tem primeru je naše podzaporedje hkrati tudi najdaljše skupno podzaporedje nizov $a_1 \ldots a_n$ in $b_2 \ldots b_m$.
  - Če pa ne velja nič od gornjega, imamo $i_1 = j_1 = 1$; no, to je očitno mogoče le, če je $a_1 = b_1$. V tem primeru je naše podzaporedje sestavljeno iz tega znaka (a_1 oz. b_1), ki mu sledi najdaljše skupno podzaporedje nizov $a_2 \ldots a_n$ in $b_1 \ldots b_m$.
- Prišli smo torej do podproblemov oblike: $f(i, j)$ = najdaljši skupni podniz nizov $a_i \ldots a_n$ in $b_j \ldots b_m$
  - In imamo $f(i, j) = \max \{ f(i + 1, j), f(i, j + 1), 1 + f(i + 1, j + 1)$ [le če je $a_i = a_j$]$\}$
  - Robni primeri: ko je eden od nizov prazen, imamo $f(n + 1, j) = 0$ in $f(i, m + 1) = 0$
  - Končni rezultat, ki nas zanima, je $f(1, 1)$

Implementacija

- Lahko bi imeli 2-d tabelo:

  for $j = 1$ **to** $m + 1$ **do** $f[n + 1, j] = 0$;
  for $i = n$ **downto** 1:
  $f[i, m + 1] = 0$
  for $j = m$ **downto** 1:
  $f[i,j] = \max\{ f[i + 1, j], f[i, j + 1],$
  $f[i + 1, j + 1]$ (le če $a_i = b_j$)$\}$

- Vidimo pa lahko, da je dovolj imeti v pomnilniku le dve vrstici te tabele – ko računamo $f(i, \cdot)$, potrebujemo vrednosti $f(i + 1, \cdot)$, ostale lahko že pozabimo
  - Šlo bi celo z eno samo vrstico + eno spremenljivko, v kateri bi si zapomnili $f[i + 1, j + 1]$ (ki smo jo v tabeli sicer tik pred tem povozili s $f[i, j + 1]$)
  - Če bi radi tudi izpisali najdaljši skupni podniz, ne le ugotovili njegove dolžine, bomo pa le potrebovali celotno 2-d tabelo

Podoben primer

- Urejevalniška (Levenštejnova) razdalja (*edit distance, Levenshtein distance*):
  - Dana sta niza $a = a_1\ a_2\ \ldots\ a_n$ in $b = b_1\ b_2\ \ldots\ b_m$
  - Kako predelati a v b s čim manj operacijami, pri čemer so dovoljene naslednje operacije:
    - Brisanje znaka (npr. *stol* → *sol*)
    - Vrivanje znaka (npr. *stol* → *stolp*)
    - Sprememba enega znaka v drugega (npr. *sol* → *vol*)
  - Podproblem: $f(i, j)$ = urejevalniška razdalja med $a_i \ldots a_n$ in $b_j \ldots b_m$
    - Potem je $f(i, j) = \min\{$

$$
\begin{array}{ll}
1 + f(i+1, j), & \text{// brisanje znaka } a_i \\
1 + f(i, j+1), & \text{// vrivanje znaka } b_j \\
1 + f(i+1, j+1), & \text{// sprememba } a_i \text{ v } b_j \\
f(i+1, j+1)\ \} & \text{// samo če je } a_i = b_j
\end{array}
$$

Dinamično programiranje

Najdaljše naraščajoče podzaporedje

Najdaljše naraščajoče podzaporedje

- Dano je zaporedje $a_1\ a_2\ \ldots\ a_n$, iščemo najdaljše naraščajoče podzaporedje:

  iščemo $k, i_1, i_2, \ldots, i_k$,
  tako da je $1 \le i_1 < i_2 < \ldots < i_k \le n$,
  $a[i_1] < a[i_2] < \ldots < a[i_k]$
  in k čim večji

- Lahko si zastavimo podproblem: $f(j)$ = najdaljše naraščajoče podzaporedje, ki se konča pri a_j
  - Če je zadnji člen tega podzaporedja a_j, kateri je predzadnji člen? Recimo mu a_t.
  - V poštev pridejo takšni t, za katere je $t < j$ in $a_t < a_j$.
  - Med njimi vzemimo tistega, ki bo dal najdaljše podzaporedje
  - Torej: $f(j) = 1 + \max\{f(t) : t < j, a_t < a_j\}$
  - Robni primer: $f(0) = 0, f(1) = 1$

Učinkovitejši postopek

- Postopek s prejšnje strani bi se dalo enostavno implementirati v času $O(n^2)$ (gnezdeni zanki po j in t)
  - Možna izboljšava: člene zaporedja, za katere smo že izračunali $f(j)$, dodajajmo sproti v binarno iskalno drevo, pri čemer je a_j ključ v drevesu, vsako vozlišče pa hrani tudi $\max_j f(j)$ po celem poddrevesu
  - Tako lahko v času $O(\log n)$ poiščemo $\max f(t)$ po vseh t, ki imajo $a_t < a_j$ (in $t < j$, a slednje je tako ali tako samoumevno, saj drugih še ni v drevesu) $\rightarrow$ skupaj $O(n \log n)$
- Do rešitve v času $O(n^2)$ pridemo lahko tudi tako, da si pripravimo še urejeno kopijo zaporedja a in nato poženemo postopek za najdaljši skupni podniz med prvotnim in urejenim zaporedjem

Učinkovitejši postopek 2

- Lahko pa pogledamo na problem malo drugače
- Če zaporedju na koncu dodamo en člen, se dolžina najdaljšega naraščajočega podzaporedja lahko poveča za 1 ali pa ostane enaka:

  $k = 0$;
  for $i = 1$ **to** n:
  if se pri a_i konča kakšno nar. zap. dolžine $k + 1$ **then** $k = k + 1$;
- Kako vemo, da se pri a_i konča naraščajoče zap. dolžine k + 1?
  - Obstajati mora nek a_t, $t < i$, $a_t < a_i$,
    pri katerem se konča naraščajoče zaporedje dolžine k
  - Če je takih a_t več, si je pametno zapomniti med njimi najmanjšega – recimo mu v
  - Tako smo dobili:

    $k = 0$; $v = -\infty$;
    for $i = 1$ **to** n:
    if $a_i > v$ **then** $k = k + 1$; $v = a_i$
    else if se pri a_i konča kakšno naraščajoče zaporedje dolžine k **then** $v = a_i$;
- Kako vemo, da se pri a_i konča naraščajoče zap. dolžine k?
  - Obstajati mora nek a_t, $t < i$, $a_t < a_i$,
    pri katerem se konča nar. zap. dolžine $k - 1$
  - Če je takih a_t več, si zapomnimo najmanjšega – recimo mu v'

Učinkovitejši postopek 2

- Tako smo dobili:

  $k = 0$; $v = -\infty$; $v' = -\infty$;
  for $i = 1$ **to** n:
  if $a_i > v$ **then** $k = k + 1$; $v = a_i$
  else if $a_i > v'$ **then** $v = a_i$;
  else if se pri a_i konča kakšno nar. zap. dolžine $k - 1$ **then** $v' = a_i$;

- Očitno bi zdaj potrebovali še v'' in tako naprej, namesto vseh tistih ifov pa zanko
  - Imejmo torej tabelo $v[0..k]$, pri čemer je v_l največji element (izmed $a_1, \ldots, a_{i-1}$), pri katerem se konča kakšno naraščajoče podzaporedje dolžine l

    $k = 0$; $v_k = -\infty$
    for $i = 1$ **to** n:
    if $a_i > v_k$ **then** $k = k + 1$; $v_k = a_i$
    else:
    $j = k$; **while** $j > 1$ do
    if $a_i > v_{j-1}$ **then break else** $j = j - 1$
    $v_j = a_i$;
  - Časovna zahtevnost je zdaj $O(n\,k)$, pri čemer je k dolžina najdaljšega nar. podzap. v celem zaporedju. V najslabšem primeru bo to $O(n^2)$
  - Izboljšava: ker je tabela v naraščajoča, lahko v njej prvi element, ki je $\leq a_i$, poiščemo kar z bisekcijo $\rightarrow$ $O(n \log k)$

Podoben problem

- Razbij dano zaporedje na čim manj disjunktnih padajočih podzaporedij (lahko nestrnjenih)
  - Naj bo k dolžina najdaljšega naraščajočega podzaporedja
  - Imeli bomo k padajočih podzaporedij
  - Vsakič ko prejšnji algoritem popravi v_j na a_i, dodamo a_i v j-to padajoče podzaporedje

Dinamično programiranje

0/1 nahrbtnik

Nahrbtnik

- Imamo n predmetov z masami m_i in vrednostmi v_i
  - V nahrbtnik bi radi naložili nekaj predmetov in to tako, da bo vsota vrednosti čim večja, vsota mas pa ne bo presegla M
  - Omejimo se na primere, ko so mase m_i in kapaciteta nahrbtnika M cela števila

Rekurzivni razmislek

- Predmet n lahko naložimo ali pa tudi ne
  - Če ga naložimo, nam ostane problem z $n - 1$ predmeti in s kapaciteto nahrbtnika $M - m_n$
  - Če ga ne naložimo, nam ostane problem z $n - 1$ predmeti in s kapaciteto nahrbtnika M
- Definirajmo torej podproblem:
  - $f(k, c)$ = največja vrednost, ki jo lahko zložimo v nahrbtnik, če ima le-ta kapaciteto c in če lahko uporabimo le prvih k predmetov
  - Velja torej
    $f(k, c) = \max\{ f(k - 1, c - m_k) + v_k , f(k - 1, c)\}$
- To lahko računamo sistematično po naraščajočih k in c
  - Ko izračunamo vse $f(k, .)$, lahko vrednosti $f(k - 1, .)$ pozabimo
- Ta postopek ni primeren, če mase niso cela števila
  - Mogoče jih lahko damo na skupni imenovalec
- Postopek je tudi neučinkovit, če so mase (in kapaciteta M) velike: $O(n \cdot M)$
- Zares učinkovite rešitve za problem nahrbtnika ne poznamo, saj je NP-težak
  - Možnih je 2^n kombinacij predmetov in če imamo smolo, ima vsaka kombinacija drugačno maso

Podoben problem

- Vračilo denarja:
  - Možne vrednosti kovancev so $m_1, \ldots, m_n$
  - S čim manj kovanci sestavi znesek M
  - Primer: kovanci 50, 20, 1, znesek $M = 60$;
    bolje je 20 + 20 + 20 kot 50 + 1 + 1 + … + 1
  - Rešitev: $f(k, z) = \min\{f(k-1, z), 1 + f(k, z - m_k)\}$

Drugačen problem

- Nahrbtnik, pri katerem lahko predmete poljubno razrežemo (in vzamemo predmet le delno, ne nujno v celoti)
  - Rešitev: jemljemo jih po padajočem v_i/m_i, dokler nahrbtnik ni poln
    - Zadnjega od teh predmetov po potrebi razrežemo

Dinamično programiranje

Trgovski potnik

Trgovski potnik

- Problem trgovskega potnika (*travelling salesman problem*, TSP):
  - Imamo n mest $\{1, 2, ..., n\}$
    in razdalje med njimi: $d(u, v)$ je dolžina povezave od u do v
  - Trgovski potnik želi obiskati vsa mesta,
    vsako natanko enkrat, in se vrniti v mesto, kjer je začel
  - Dolžina te poti naj bo najmanjša
- Vprašanje je torej, v kakšnem vrstnem redu naj obišče mesta
  - Rezultat je neka permutacija π množice $\{1, ..., n\}$
  - Dolžina poti pa je $\Sigma_{k=1..n}\, d(\pi(k), \pi(k + 1))$
  - Možnih permutacij je sicer $n! = n\,(n-1)\,(n-2)\,... \cdot 3 \cdot 2 \cdot 1$,
    vendar jih po n opisuje isti obhod, zato je možnih obhodov „le“ $(n-1)!$

Rekurzivna rešitev

- Očitno lahko vse poti sistematično pregledamo z rekurzijo
  - Brez izgube za splošnost se omejimo na poti, ki se začnejo v točki $z = 1$
  - Pot bomo hranili v tabeli $p[1..n]$, pri čemer $p[k]$ pove, katero mesto obiščemo kot k-to po vrsti

funkcija TSP(p, D, k, A):
// p = tabela, v kateri nastaja pot; D = dosedanja dolžina poti
// k = število mest, ki smo jih že dodali na pot (v $p[1..k]$)
// A = mesta, ki jih še nismo obiskali; $|A| = n - k$
// Vrne najkrajšo dolžino, s katero se da sestaviti preostanek poti.
if $k = n$ then return $D + d(p[n], p[1])$; // smo že na koncu poti
$naj = -\infty$
za vsako mesto $u \in A$:
$p[k + 1] = u$;
$naj = \min\{naj, \text{TSP}(p, D + d(p[k], u), k + 1, A - \{u\})\}$;
return naj;

Prihranek

- Doslej smo pri dinamičnem programiranju izkoristili dejstvo, da rekurzivna rešitev včasih rešuje po večkrat isti podproblem
  - Je tudi tu kaj takega?
  - Opazimo lahko, da je za nadaljevanje poti pomembno le to,
    - kateri kraji so še neobiskani (množica A),
    - in v katerem kraju se trenutno nahajamo – to je $p[k]$
    - Elementov $p[1, ..., k-1]$ pa se naš podprogram TSP sploh ni dotikal
  - Torej si lahko podrobleme definiramo takole: $f(v, A)$ naj bo dolžina najkrajše take poti, ki se začne v v, konča v z,
    vmes pa obišče vse točke iz A, vsako natanko enkrat
    - Na koncu nas bo zanimalo $f(z, \{1, ..., n\} - \{z\})$
    - Rekurzija je zdaj $f(v, A) = \min \{ d(v, u) + f(u, A - \{u\}) : u \in A \}$
  - Namesto $O(n!)$ podproblemov imamo zdaj „le“ $O(n\, 2^n)$ podproblemov
    - Še vedno eksponentno, a veliko manj kot prej

Podoben problem

- Dodeljevanje:
  - Imamo n otrok in n skirojev
  - a_{ij} pove, kako rad ima otrok i skiro j
  - Razdeli otrokom skiroje (permutacija π) tako, da maksimiziraš min $\{ a_{i, \pi(i)} : 1 \leq i \leq n \}$
  - Rešitev:
    - Otroke pregledujemo po vrsti in jim dodeljujemo skiroje
    - Podproblem $f(A)$ = prvim $|A|$ otrokom smo že razdelili skiroje; neuporabljeni so ostali še skiroji iz množice A, ki jih moramo razdeliti otrokom od $n - |A| + 1$ do n

Naloge

- 821 page hopping
- 10066 twin towers
- 10003 cutting sticks
- 10534 wavio sequence
- 10739 string to palindrome
- 10724 road construction
- 11456 trainsorting
- 10635 prince and princess

821 Page Hopping

- Dan je usmerjen graf na $n \leq 100$ točkah
- Poišči povprečno dolžino najkrajše poti med vsemi pari točk
  - Zagotovljeno je, da taka pot za vsak par točk res obstaja
  - Vse povezave so enako dolge (pomembno je torej le število povezav na poti)
- Rešitev: ta naloga je kot nalašč za Floyd-Warshallov algoritem

10066 Twin Towers

- Imamo dva stolpa iz okroglih plošč
  - Prvega sestavlja n_1 plošč, drugega n_2 plošč
  - Dane so velikosti vseh plošč (po vrsti, kot so zložene v stolp)
  - Radi bi pobrisali nekaj plošč tako, da bosta stolpa postala enaka (in čim višja)
- Rešitev: to je ravno problem najdaljšega skupnega podzaporedja
  - Za vhod vzamemo zaporedji velikosti plošč za vsakega od stolpov

10003 Cutting Sticks

- Imamo palico z dano dolžino, ki bi jo radi razrezali na danih n (največ 50) mestih
  - Cena vsakega reza je definirana kot dolžina kosa pred rezom
  - V kakšnem vrstnem redu naj režemo, da bo čim ceneje?
- Rešitev:
  - Zelo podobna naloga kot tista z delitvijo zemljišč od zjutraj
  - Razlika je le v tem, da niso podane dolžine končnih kosov, ampak položaji rezov (glede na začetno palico)
  - In da je cena reza iz $a + b$ v a in b tu enaka $a + b$ namesto $\max\{a, b\}$

10534 Wavio Sequence

- V danem zaporedju n ≤ 10000 števil poišči najdaljše tako podzaporedje (lahko nestrnjeno), ki:
  - Je lihe dolžine (na primer 2m + 1)
  - Prvih m + 1 členov je naraščajoče
  - Zadnjih m + 1 členov je padajoče
- Rešitev:
  - Pri algoritmu za najdaljše naraščajoče podzaporedje smo že videli, da spotoma za vsak element vhodnega zaporedja izvemo, kako dolgo je najdaljše naraščajoče podzaporedje, ki se konča pri njem
    - Ta dolžina je prav tisti j, za katerega smo ai vpisali v vj
  - Nato zaporedje obrnimo in spet iščimo najdaljše naraščajoče podzaporedje – to ustreza padajočim podzaporedjem v prvotnem zaporedju
  - Na koncu rezultate le še skombiniramo

10739 String to Palindrome

- Dan je niz n ≤ 1000 znakov
  - Radi bi ga predelali v poljuben palindrom s čim manj operacijami
    - Dodajanje znaka, brisannje znaka, sprememba enega znaka
- Rešitev:
  - Naj bo $f(i, j)$ najmanjše število operacij, s kateerim iz $ai \ldots aj$ naredimo palindrom
  - Potem je $f(i, j) = \min \{$

$$1 + f(i + 1, j), \quad // \text{ brisanje } a_i$$
$$1 + f(i, j - 1), \quad // \text{ brisanje } a_j$$
$$1 + f(i + 1, j - 1), \quad // \text{ sprememba } a_i \text{ v } a_j$$
$$f(i + 1, j - 1) \} \quad // \text{ le, če je } a_i = a_j$$

10724 Road Construction

- Dan je povezan neusmerjen graf z n ≤ 50 točkami
  - Za vsako povezavo je dana dolžina, za vsako manjkajočo povezavo pa vemo, kakšna bi bila njena dolžina, če bi jo dodali v graf
  - Ugotovi, katero povezavo se najbolj splača dodati
    - Minimiziraj vsoto dolžin najkrajših poti med vsemi pari točk po dodajanju nove povezave
- Rešitev:
  - Če bi za vsako možno novo povezavo (ki jih je $O(n^2)$) poganjali Floyd-Warshalla (ki je $O(n^3)$), bi imeli $O(n^5)$, kar je prepočasi
  - Recimo, da je $d(i, j)$ dolžina najkrajše poti od i do j v prvotnem grafu
  - Če dodamo v graf povezavo (u, v) z dolžino D, se najkrajše poti spremenijo takole:

    $$d_{nova}(i,j) = \min\{d(i,j),\ d(i,u) + D + d(v,j),\ d(i,v) + D + d(u,j)\}$$
  - Tako smo pri $O(n^4)$ namesto $O(n^5)$

11456 Trainsorting

- Dobimo zaporedje vagonov, za vsakega je znana teža
  - Iz njih bi radi sestavili čim daljši vlak, v katerem bodo vagoni urejeni padajoče
  - Vsak vagon lahko dodamo na začetek vlaka, na konec vlaka ali pa ga zavrnemo (ga sploh ne dodamo)
- Rešitev:
  - Naj bo *i* prvi vagon, ki ga ne zavrnemo
  - Potem nas zanima najdaljše naraščajoče zaporedje, ki se začne pri *i*, in najdaljše padajoče zaporedje, ki se začne pri *i*
  - Vagone iz prvega bomo dodajali na začetek, tiste iz drugega pa na konec
  - Vzamemo seveda maksimum po vseh *i*

10635 Prince and Princess

- Imamo dve zaporedji, ki sta permutaciji množice $\{1, \ldots, n\}$; poišči najdaljše skupno podzaporedje
- Rešitev:
  - Težava je v tem, da je n lahko do 250^2, zato je $O(n^2)$ prepočasi
  - Pomagati si moramo z dejstvom, da sta naši zaporedji permutaciji – vsako število od 1 do n se pojavi natanko enkrat
  - Naj bo $f(u)$ dolžina najdaljšega skupnega podzaporedja, ki se začne z elementom u
  - Potem je $f(u) = 1 + \max\{f(v) : v$ se pojavlja v obeh zaporedjih desno od $u\}$
  - Kako ta max izračunati učinkovito?
    - Pojdimo z u v prvem zaporedju od konca proti začetku
    - Vsak obdelani u dodajmo v binarno iskalno drevo, pri čemer kot ključ uporabimo njegov položaj v drugem zaporedju, v vsakem vozlišču pa vzdržujemo max $f(v)$ po vseh v iz njegovega poddrevesa
    - Tako lahko v $O(\log n)$ časa izračunamo max, ki ga iščemo