Dinamično programiranje

Ena od nalog z lanskega južnoameriškega regijskega tekmovanja ACM (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 → 8 + 3 (davek: 8)

11 → 5 + 6 (davek: 6)

8 → 6 + 2 (davek: 6)

6 → 3 + 3 (davek: 3)

6 → 5 + 1 (davek: 5)

3 → 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.

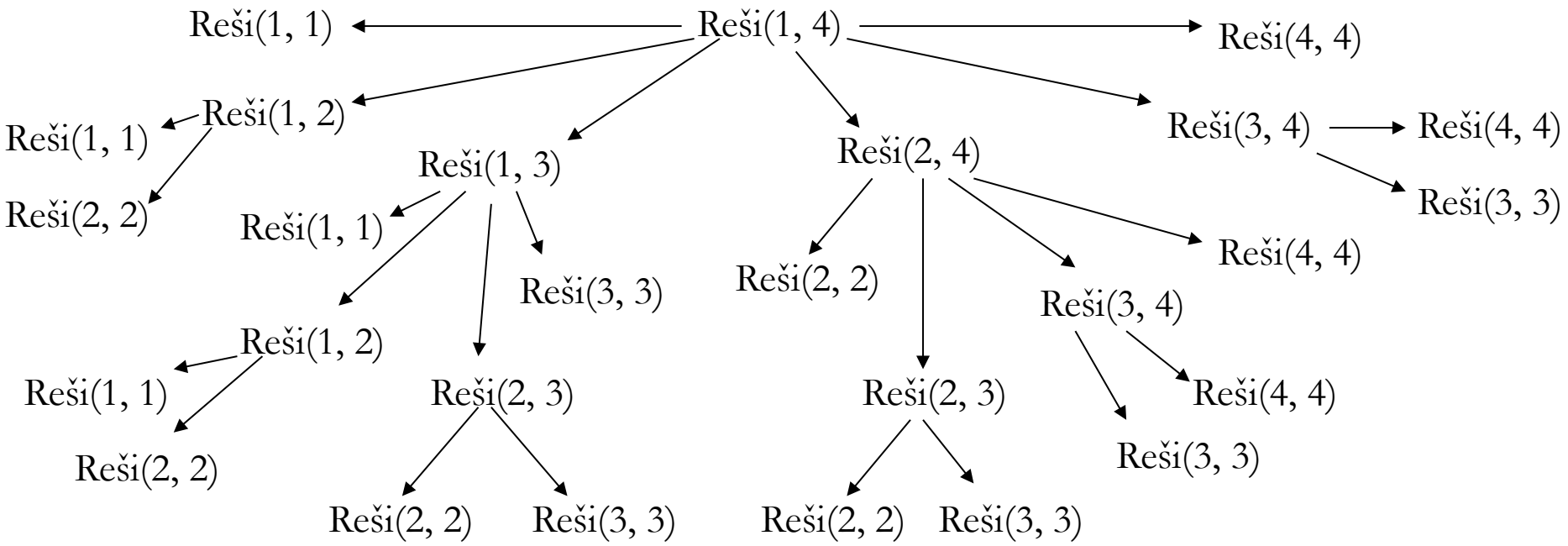
Časovna zahtevnost

- 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 $\#_m = 2(\#_{m+1} + \#_{m+2} + \ldots + \#_n)$.
- $\#_{n-1} = 2(\#_n) = 2$
  $\#_{n-2} = 2(\#_n + \#_{n-1}) = 2(1 + 2) = 6$
  $\#_{n-3} = 2(\#_n + \#_{n-1} + \#_{n-2}) = 2(1 + 2 + 6) = 18$
  $\#_{n-4} = 54$, $\#_{n-5} = 162$, $\#_{n-6} = 486$, ...
- 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] := ∞;
        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.
  - Na primer, če imamo opravka s podmnožicami namesto (strnjenimi) podzaporedji
  - Mogoče je takšna eksponentna zahtevnost še vseeno sprejemljiva, če je n majhen
- 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 primerov nalog za DP

- Dan je niz $s_1 s_2 ... s_n$. Dovoljene operacije: brisanje črke, dodajanje črke, vrivanje črke. Koliko korakov potrebuješ, da dobiš iz njega niz $t_1 t_2 ... t_m$? (Levenshtein, urejevalniška razdalja)
  - Podproblem: $f(i, j)$ = koliko korakov potrebuješ, da iz niza $s_i s_{i+1} ... s_n$ narediš niz $t_j t_{j+1} ... t_m$?
  - $f(i, j) = f(i + 1, j + 1)$, če $s_i = t_j$
    $f(i, j) = \min(1 + f(i + 1, j), 1 + f(i, j + 1)\}$ sicer.
- Dan je graf, poišči najkrajše poti med vsemi pari točk. (Floyd-Warshall)
  - Podproblem: $f(u, v, k)$ = dolžina najkrajše poti od točke u do točke v, pri čemer med njima hodi le po točkah $\{1, ..., k\}$, po ostalih pa ne.
  - $f(u, v, k) = \min(f(u, v, k - 1), f(u, k, k - 1) + f(k, v, k - 1)\}$

Še nekaj primerov nalog za DP

- (Triangulacija) Dan je mnogokotnik, izberi nekaj diagonal (ki se ne smejo sekati), tako da bo razpadel na trikotnike in bo vsota njihovih cen minimalna.
  - Naj bodo $A_1, A_2, \ldots, A_n$ oglišča našega mnogokotnika.
  - Podproblem: $f(i,j)$ = trianguliraj mnogokotnik $A_i A_{i+1} \ldots A_{j-1} A_j$.
  - Ideja: daljica $A_j A_i$ bo v trikotniku z nekim oglišč̌em A_k.
    $f(i,j) = \min\{ cena(A_i A_k A_j) + f(i,k) + f(k,j) : i < k < j \}$.
- Rad bi zmnožil zaporedje matrik. Kako postaviti oklepaje, da bo skupna cena množenja najmanjša?
- Dan je "vzorec" — niz, v katerem nastopata wildcarda * in ?. Ali se dano ime datoteke ujema s tem vzorcem?
  - Posplošitev: dan je poljuben regularni izraz.
- Dana sta dva niza, poišči najdaljši skupni nestrnjeni podniz.
- Itd., itd., itd.

Požrešni algoritmi

Greed is good. Greed works.
GORDON GEKKO

Ideja

- Požrešni algoritmi gradijo rešitev postopoma
  - Vsakič naredijo tak korak, ki nam v tistem hipu rešitev najbolj izboljša
  - Ne obremenjujejo se s tem, ali je ta korak tudi dolgoročno dober ali ne
  - Ne vračajo se nazaj in popravljajo prej sprejetih odločitev, če se izkaže, da niso bile dobre
- Ponavadi to ni najbolj pametno, vendar…
  - Pri nekaterih problemih se izkaže, da požrešni algoritem vrne optimalno rešitev.
    - Spodobi se, da to tudi dokažemo
    - Ti dokazi gredo vsi po istem kopitu: vzamemo optimalno rešitev in jo korak za korakom predelamo v požrešno rešitev, ne da bi se pri tem kdaj poslabšala
  - Pri nekaterih drugih problemih se izkaže vsaj, da požrešni algoritem vrne nek približek optimalne rešitve
    - In mogoče lahko tudi povemo, za koliko je slabši od optimalne

Trajekt

- Imamo n vozil z dolžinami $d_1, d_2, \ldots, d_n$
- Nekaj jih moramo izbrati za na trajekt, pri čemer skupna dolžina ne sme preseči D
- Katere naj izberemo, da jih bo čim več?
- Rešitev: uredimo jih po dolžini

  $d = 0$; $i = 0$;
  while ($i + 1 < n$ **and** $d + d_{i+1} > D$) {
  $i = i + 1$; $d = d + d_i$; }
- Intuitivno je zelo očitno, da je ta izbor vozil optimalen
  - Spodobi pa se, da znamo to tudi dokazati
  - Pri nekaterih nalogah sploh ni tako očitno, da je požrešna rešitev res optimalna

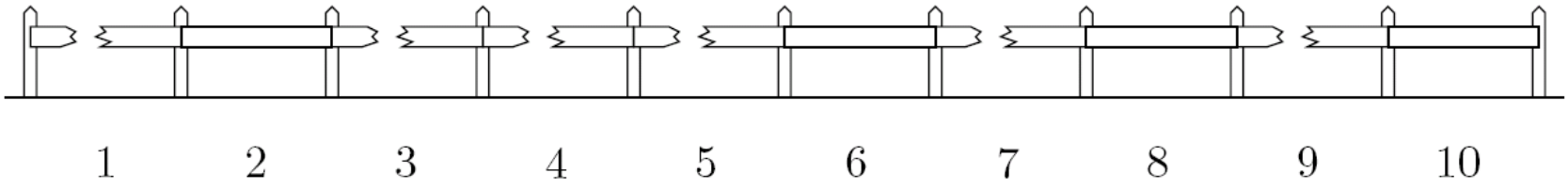
Trajekti – dokaz

1. Mislimo si vozila oštevilčena po naraščajoči dolžini
2. Naj bo $A \subseteq \{1, \ldots, n\}$ množica indeksov, ki tvorijo optimalno rešitev
3. Naj bo $B = \{1, 2, \ldots, k\}$ naša požrešna rešitev
4. Očitno je $|A| \geq |B|$. Če velja enakost, je B tudi optimalna in smo končali.
5. Ostane še primer, ko je $|A| > |B|$.
6. Ali je mogoče $B \subseteq A$? Ne, kajti B smo sestavili tako, da vanj ni mogoče dodati niti najkrajšega izmed vozil $\{k+1, \ldots, n\}$, ne da bi presegli mejo D
7. Torej obstaja nek $x \in B - A$; če je takih več, vzemimo med njimi najmanjšega.
   To med drugim pomeni, da so indeksi $1, 2, \ldots, x - 1$ prisotni tako v A kot v B.
8. Po drugi strani, ker je $|A| > |B|$, obstaja nek $y \in A - B$.
   Iz prejšnje točke sledi, da y ni eden od $1, 2, \ldots, x - 1$, torej mora biti $y > x$.
9. Zato pa tudi $d_y \geq d_x$ in če v A zamenjamo vozilo y z vozilom x, se skupna dolžina ne poveča (lahko ostane enaka ali pa se celo zmanjša).
10. Tako popravljena A ostane veljavna, ima enako število vozil kot prej, njen presek z B pa je za eno večji kot prej.
11. Razmislek v točkah 7 – 10 lahko ponavljamo in presek $A \cap B$ ves čas narašča. Prej ali slej pridemo do $B \subseteq A$, kar pa je nemogoče (točka 6), torej smo v protislovju. Torej je bila predpostavka $|A| > |B|$ iz točke 5 nemogoča.

Ograja

- Imamo ograjo iz n deščic (vsaka je dolga po 1 meter), nekatere deščice so poškodovane
- Zamenjali jih bomo tako, da bomo k zaporednih deščic zamenjali z eno novo (k-metrsko) desko
  - Lahko je kakšna od zamenjanih deščic tudi nepoškodovana
- Koliko je najmanjše potrebno število novih k-metrskih desk?
- Požrešna rešitev:

  $i = 0$; **while** $(i < n)$
  if (*poškodovana*[i]) { začni tu novo desko; $i \mathrel{+}= k$; }
  else $i = i + 1$;



Ograja – dokaz

- Razpored desk lahko predstavimo tako, da zapišemo položaj levega konca vsake deske – dobimo neko podmnožico množice $\{1, \ldots, n\}$.
- Naj bo A optimalni razpored desk, B pa naš požrešni razpored; elemente obeh množic uredimo naraščajoče: $a_1, \ldots, a_k$ in $b_1, \ldots, b_m$
  (očitno je $k \leq m$, ker je A optimalna)
- Primerjajmo istoležne elemente: a_i in b_i po naraščajočih i. Oglejmo si prvi i, kjer nastopi neujemanje: $a_i \neq b_i$.
  - Ali je lahko $a_i > b_i$? To, da je naš požrešni algoritem začel desko na b_i, pomeni, da je bila b_i polomljena in da je dotedanje deske ($b_1, \ldots b_{i-1}$) niso pokrile. Torej je tudi ($a_1, \ldots, a_{i-1}$) ne pokrijejo (saj so to iste deske). Če bi bila $a_i > b_i$, je potemtakem tudi a_i (in $a_{i+1}, \ldots, a_n$) ne bi pokrile, torej A sploh ne bi bila veljavna rešitev.
  - Torej je $a_i < b_i$. Toda vse poškodovane deščice, ki ležijo levo od b_i, so pokrite že z deskami $b_1, \ldots, b_{i-1}$ (saj bi sicer požrešni algoritem postavil i-to desko bolj levo, ne šele na b_i). Torej ni nobene škode, če v rešitvi A pomaknemo i-to desko z a_i na b_i.
- Gornji razmislek ponavljamo, dokler je še kaj neujemanj. Sčasoma torej pridemo do tega, da je $a_i = b_i$ za vse i od 1 do m.
  - Torej vsebuje A vse deske, ki jih vsebuje tudi B.
  - Če bi vsebovala A še kakšno desko več ($k > m$), ne bi bila optimalna, saj vemo, da že deske iz B zadoščajo za popravilo cele ograje.
  - Torej je $k = m$ in B je optimalna.

Viadukt

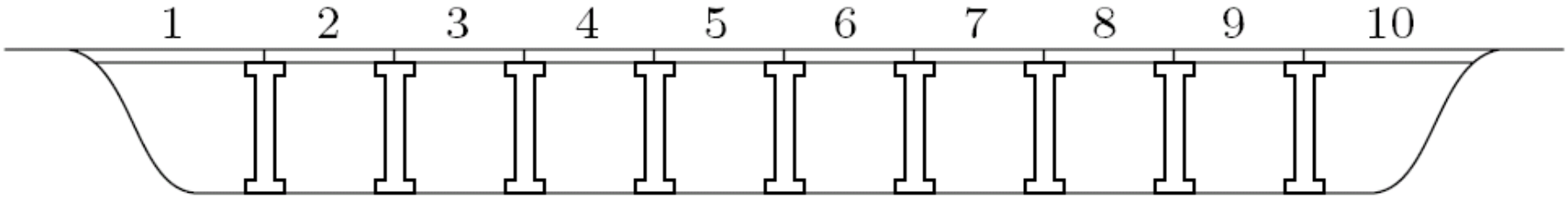
- Imamo viadukt, razdeljen na n členov
- Dane so omejitve nosilnosti: (u_i, v_i, w_i) pomeni, da sme biti skupno število avtomobilov na členih od u_i do v_i največ w_i

  $a[u_i] + a[u_i + 1] + \ldots + a[v_i - 1] + a[v_i] \leq w_i$
- Razporedi avtomobile po členih tako, da bodo upoštevane vse omejitve in da bo vsota $a[1] + \ldots + a[n]$ maksimalna
- Rešitev:

  for ($t = 0$; $t < n$; t++) $a[t] = 0$;

  for ($t = 0$; $t < n$; t++)

  $a[t] = \min \{ w_i - (a[u_i] + a[u_i + 1] + \ldots + a[v_i - 1] + a[v_i]) : u_i \leq t \leq v_i \}$



Viadukt – dokaz

- Naj bo $b[1..n]$ rešitev, ki jo je našel naš požrešni algoritem; $a[1..n]$ pa naj bo optimalna rešitev.
  - Če je več enako dobrih optimalnih rešitev, vzemimo tisto z največjim $a[1]$, med njimi tisto z največjim $a[2]$ itd.
- Poiščimo najmanjši i, kjer je $a[i] \neq b[i]$.
  - Ali je lahko $a[i] > b[i]$?
    - Preden je požrešni algoritem definiral $b[i]$, je imel razpored $(b[1], \ldots, b[i-1], 0, 0, \ldots, 0)$.
    - In za $b[i]$ je vzel največje število, s katerim takrat še ne prekršimo nobene od omejitev. Pri $a[i] > b[i]$ bi to pomenilo, da je razpored $(b[1], \ldots, b[i-1], b[i], 0, \ldots, 0)$ že neveljaven.
    - Torej je $a = (b[1], \ldots, b[i-1], a[i], a[i+1], \ldots, a[n])$ še bolj neveljaven – protislovje.
  - Torej je $a[i] < b[i]$. Ker pa je skupno število vozil pri a večje kot pri b, mora imeti na nekem kasnejšem členu a več vozil kot b. Vzemimo torej najmanjši $j > i$, pri katerem je $a[j] > 0$.
    $a = (b[1], \ldots, b[i\text{-}1], a[i], \quad 0, \quad \ldots, \quad 0, \quad a[j], \ldots)$
    $b = (b[1], \ldots, b[i\text{-}1], b[i], b[i+1], \ldots\ b[j-1], b[j], \ldots)$
  - Ker je a optimalna, se $a[i]$ gotovo ne da povečati, ne da bi prekoračili kakšno omejitev, recimo (u_k, v_k, w_k).
    - Če bi ta omejitev v celoti ležala na območju od 1 do $j-1$, bi bila v b torej prekoračena.
    - Torej lahko $a[j]$ zmanjšamo za 1 in $a[i]$ povečamo za 1, pa a ostane veljavna (kajti ta razmislek velja za vsako omejitev, ki pokriva indeks i) in postane leksikografsko večja, kar je protislovje.