DINAMIČKO PROGRAMIRANJE

UVOD U DINAMIČKO PROGRAMIRANJE …

- Kao i divide-and-conquer pristup, rešava problem tako što kombinuje rešenja njegovih potproblema
- Dinamičko programiranje se koristi kada postoji preklapanje potproblema
- Divide-and-conquer nije dobar za ovakve probleme
- Zašto?
- Jer rešavanjem potproblema, rešava jednu te istu stvar više puta
- U dinamičkom programiranju, kada se reši neki potproblem, njegovo **rešenje se čuva** (u tabeli), tako da ne mora ponovo da se rešava
- Tipično, dinamičko programiranje se koristi za ***optimizacione probleme***

… UVOD U DINAMIČKO PROGRAMIRANJE …

- Dinamičko programiranje se koristi za ***optimizacione probleme***
  - Karakteristični po tome da za jedan problem postoji više rešenja
  - Za svako rešenje dobija se vrednost
  - Želimo rešenje sa optimalnom vrednošću (minimalno ili maksimalno)
  - Ovo rešenje zove se jednim optimalnim rešenjem, jer može biti više rešenja koja daju optimalnu vrednost (nema garancije da postoji samo jedno optimalno rešenje)

… UVOD U DINAMIČKO PROGRAMIRANJE

- Kada se razvijaju algoritmi dinamičkog programiranja, obično se sprovode sledeća četiri koraka:
  1. Definiše se struktura optimalnog rešenja
  2. Rekurzivno se definiše vrednost optimalnog rešenja
  3. Izračuna se jedno optimalno rešenje (bottom-up pristup)
  4. Konstruiše se optimalno rešenje na osnovu određenih informacija
- 1 – 3 osnova dinamičkog programiranja
- Ako nam ne treba rešenje, već samo optimalna vrednost, možemo preskočiti korak 4
- Tokom četvrtog koraka mogu se koristiti i vrednosti iz trećeg koraka

DINAMIČKO PROGRAMIRANJE, PRIMER – SEČENJE CEVI

SEČENJE CEVI …

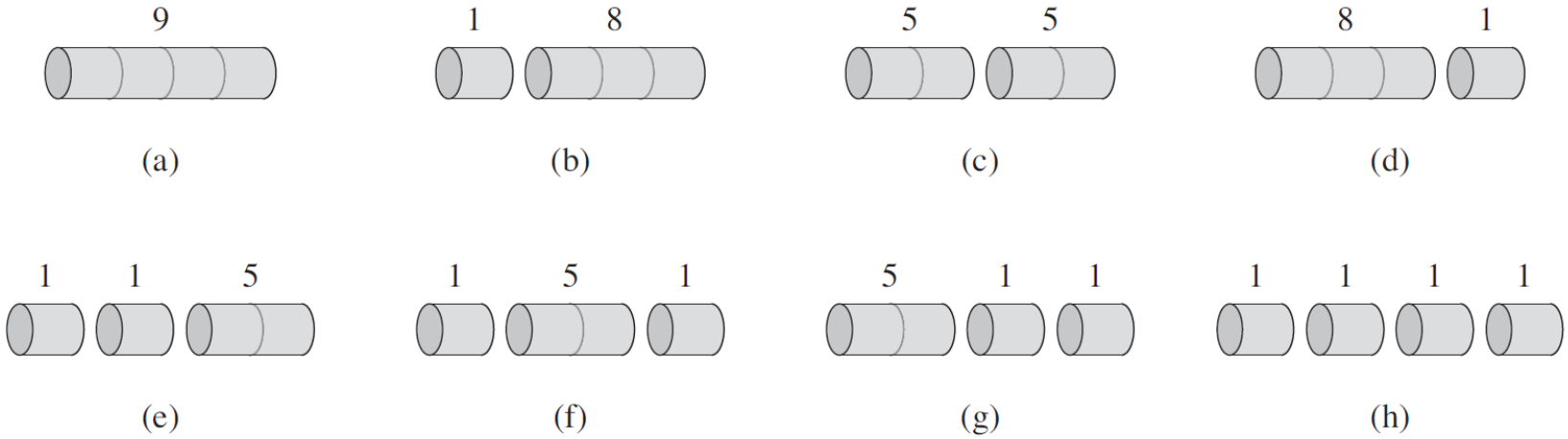
- Pretpostaviti da je potrebno iseći određen broj čeličnih cevi
- Sečenje je besplatno
- Traži se najbolji način da se cevi iseku kako bi se maksimizovao profit od prodaje cevi, pri čemu se zna cena za cevi određene dužine
  - Sve cevi su uvek pravilne dužine i predstavljaju celobrojni umnožak jednog centimetra
  - Cevi mogu biti različitih dužina
  - Moguće je da ne dođe do sečenja cevi, ako je ona dovoljno dugačka da ostvaruje maksimalni profit
- Odnos između cena i dužina cevi je:

Dužina i	1	2	3	4	5	6	7	8	9	10
Cena p_i	1	5	8	9	10	17	17	20	24	30

… SEČENJE CEVI …

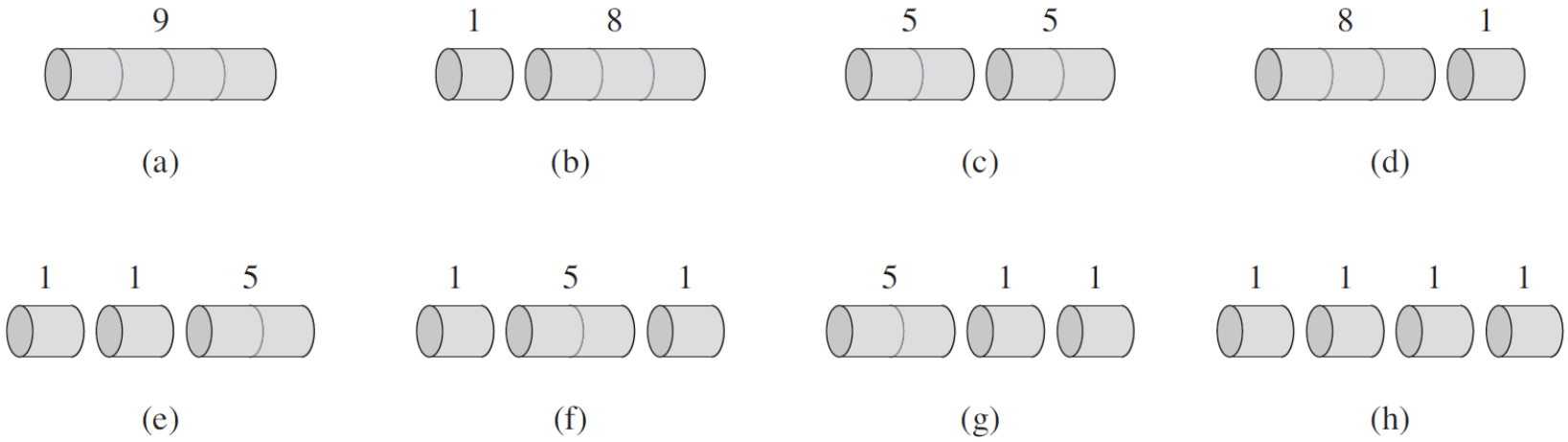
Dužina i	1	2	3	4	5	6	7	8	9	10
Cena p_i	1	5	8	9	10	17	17	20	24	30

- Šta algoritam treba da radi?
- Pa kad stigne cev neke dužine, treba da se odluči kako da se ona seče
- Na primer, ako je $n = 4$, postoji nekoliko opcija, ali je očigledno najbolje da se seče na dva jednaka dela, jer je tada zarada 10



- Očigledno se cev dužine n može iseći na 2^{n-1} načina, jer postoji mogućnost da se cev seče, ili ne, na i-tom rastojanju od leve ivice cevi

… SEČENJE CEVI …



- Naravno, ako se uvede pravilo da nema ponavljanja delova samo u različitom razmeštaju bilo bi manje od 2^{n-1} kombinacija
  - jer se u suštini (b) i (d), kao (e), (f) i (g) ne razlikuju po pitanju broja delova i njihovih dužina
  - Izbaci se dekrementovano sečenje (dozvoli se samo manje+veće)
- Deljenje na delove se uglavnom predstavlja putem običnog sabiranja:
  - $6 = 3 + 2 + 1$, cev dužine ***6*** deli se na tri dela dužina ***3***, ***2***, ***1***

… SEČENJE CEVI …

- Ako optimalno rešenje deli cev u $\boldsymbol{k}$ delova, za $\boldsymbol{1 \leq k \leq n}$, onda optimalna dekompozicija

  $$\boldsymbol{n = i_1 + i_2 + \cdots + i_k}$$

  koja deli cev na delove dužina $\boldsymbol{i_1}$, $\boldsymbol{i_2}$, …, $\boldsymbol{i_k}$ omogućuje maksimalnu zaradu od:

  $$\boldsymbol{r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}}$$

- U ovom primeru za svako $\boldsymbol{r_i}$, $\boldsymbol{i = 1, 2, \ldots, 10}$, može se odrediti optimalna vrednost

Dužina i	1	2	3	4	5	6	7	8	9	10
Cena p_i	1	5	8	9	10	17	17	20	24	30

… SEČENJE CEVI …

Dužina i	1	2	3	4	5	6	7	8	9	10
Cena p_i	1	5	8	9	10	17	17	20	24	30

- U ovom primeru za svako r_i, $i = 1, 2, \ldots, 10$, može se odrediti optimalna vrednost

$r_1 = 1$ iz rešenja $1 = 1$ (nema sečenja)

$r_2 = 5$ iz rešenja $2 = 2$ (nema sečenja)

$r_3 = 8$ iz rešenja $3 = 3$ (nema sečenja)

$r_4 = 10$ iz rešenja $4 = 2 + 2$

$r_5 = 13$ iz rešenja $5 = 2 + 3$

$r_6 = 17$ iz rešenja $6 = 6$ (nema sečenja)

$r_7 = 18$ iz rešenja $7 = 1 + 6$ ili $7 = 2 + 2 + 3$

$r_8 = 22$ iz rešenja $8 = 2 + 6$

$r_9 = 25$ iz rešenja $9 = 3 + 6$

$r_{10} = 30$ iz rešenja $10 = 10$ (nema sečenja)

… SEČENJE CEVI …

- Uopšteno se problem određivanja maksimalne zarade $\boldsymbol{r_n}$ za cev dužine $\boldsymbol{n \geq 1}$ može izraziti kao:

$$\boldsymbol{r_n} = \max(\boldsymbol{p_n}, \boldsymbol{r_1} + \boldsymbol{r_{n-1}}, \boldsymbol{r_2} + \boldsymbol{r_{n-2}}, \dots, \boldsymbol{r_{n-1}} + \boldsymbol{r_1})$$

  - Prvi parametar ***max*** funkcije je zarada $\boldsymbol{p_i}$ kada se cev ne seče
  - Preostalih $\boldsymbol{n-1}$ parametara ***max*** funkcije nastaje deljenjem cevi na dva dela veličine $\boldsymbol{i}$ i $\boldsymbol{n-i}$, za svako $\boldsymbol{i = 1, 2, \dots, n-1}$, a onda optimalno sečenjem tih delova kako bi se dobile zarade $\boldsymbol{r_i}$ i $\boldsymbol{r_{n-i}}$
  - Kako ne znamo koji deo ***i*** optimizuje zaradu unapred, moramo da proučimo sve moguće vrednosti za ***i*** i da odaberemo ono koje maksimizuje zaradu
  - Takođe mora postojati i opcija da se ne odabere ni jedno ***i*** ako će maksimalna zarada nastati zadržavanjem originalne dužine cevi

… SEČENJE CEVI …

- Obratiti pažnju da se problem rešava deljenjem na potprobleme koji su manji, ali istog tipa
- Kada se cev dužine ***n*** preseče prvi put, za svaki od 2 dela može se analizirati kako njih dalje seći (kao da se prvi put susreće taj problem)
- Optimalno rešenje za cev dužine ***n*** uključuje i optimalna rešenja za njegove delove
- Kaže se da problem sečenja cevi ima osobinu optimalne podstrukture *(optimal substructure)*
  - *Optimalna rešenja nekog problema uključuju optimalna rešenja potproblema na koje se on može podeliti, a koji se svaki može rešiti nezavisno*

… SEČENJE CEVI …

- Problem sečenja cevi može se pojednostaviti tako što se od cevi odseče deo sa leve strane dužine $\boldsymbol{i}$ i ostatak dužine $\boldsymbol{n - i}$
- Deljenje se dalje dozvoljava samo za ostatak dužine $\boldsymbol{n - i}$
- Na taj način dekompozicija čitave cevi može se videti kao deljenje na levi i desni deo, pri čemu se nastavlja dalje samo sa deljenjem desnog dela
- Jasno da se algoritam može modifikovati tako da se rešenje bez sečenja opiše kao rešenje u kojem se cev prvo seče na levi deo dužine $\boldsymbol{i = n}$ čija cena je $\boldsymbol{p_n}$ i ostatka $\boldsymbol{0}$ čija cena je $\boldsymbol{r_0 = 0}$
- Na ovaj način se jednačina:

  $$\boldsymbol{r_n} = \max(\boldsymbol{p_n}, \boldsymbol{r_1} + \boldsymbol{r_{n-1}}, \boldsymbol{r_2} + \boldsymbol{r_{n-2}}, \dots, \boldsymbol{r_{n-1}} + \boldsymbol{r_1})$$

  može pojednostaviti:

  $$\boldsymbol{r_n} = \max(\boldsymbol{p_i} + \boldsymbol{r_{n-i}})$$

… SEČENJE CEVI …

- Algoritam koji implementira jednačinu $\boldsymbol{r_n = \max(p_i + r_{n-i})}$ ima sledeći oblik:

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

- Procedura CUT-ROD uzima niz cena, $\boldsymbol{p[1 \ldots n]}$ i ceo broj $\boldsymbol{n}$ i vraća maksimalnu zaradu za cev dužine $\boldsymbol{n}$

… SEČENJE CEVI …

- Procedura CUT-ROD uzima niz cena, $\boldsymbol{p[1 \ldots n]}$ i ceo broj $\boldsymbol{n}$ i vraća maksimalnu zaradu za cev dužine $\boldsymbol{n}$
- Ako je $\boldsymbol{n = 0}$, ne postoji zarada i procedura vraća 0

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

- Promenljiva $\boldsymbol{q}$ čuva u sebi maksimalnu zaradu i inicijalno se postavlja na najmanju moguću vrednost
- Zašto?
- Da bi moglo korektno da se proračuna:

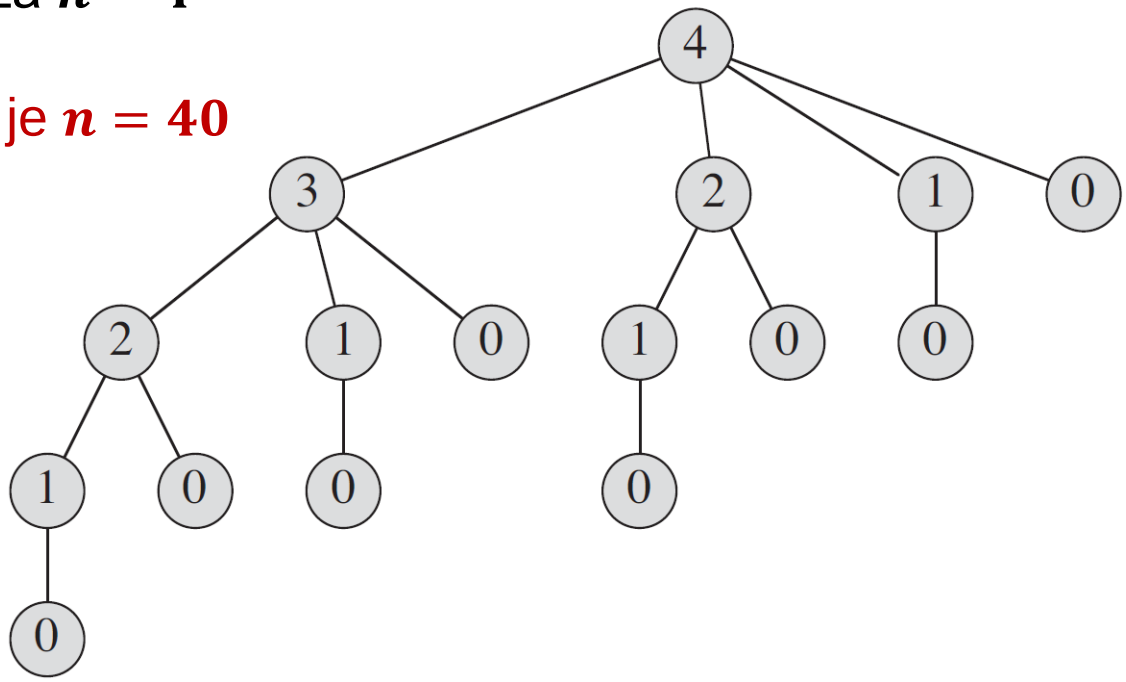
$$\boldsymbol{q = \max_{1 \le i \le n} (q, p[i] + CUT - ROD(p, n - i))}$$

- Jednostavnom indukcijom se može dokazati invarijantnost petlje i validnost algoritma

… SEČENJE CEVI …

- Koji je problem sa procedurom CUT-ROD?
- Algoritam je izrazito neefikasan
- Pogledajte koliko postoji grana izvršavanja za $n = 4$
- Zamislite da je $n = 40$

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```



… SEČENJE CEVI …


```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

- Neka $\boldsymbol{T(n)}$ označava koliko puta se poziva procedure CUT-ROD kada je dužina cevi ***n***
- Očigledno je da broj poziva odgovara broju grana i čvorova u stablu rekurzije počev od korena ***n*** plus inicijalni poziv same procedure za ***n***

$$\boldsymbol{T(n) = 1 + \sum_{j=0}^{n-1} T(j)}$$

- Jasno je da se za ***n*** procedura poziva $\boldsymbol{n + 1}$ put
- Kada se ovo primeni na svaki korak u rekurziji dolazi se do

$$\boldsymbol{T(n) = 2^n}$$

- Tako da je vreme izvršavanja algoritma eksponencijalna funkcija

… SEČENJE CEVI …

- Rešenje je primena dinamičkog programiranja
- Naivni rekurzivni pristup nije efikasan, jer rešava svaki potproblem ponovo
- Kod dinamičkog programiranja, trebalo bi preurediti algoritam tako da se svaki potproblem rešava samo jednom
  - Rezultat svakog potproblema se čuva za kasnije
  - Tako da može da se preuzme, ako je potreban u rešenju nekog drugog potproblema
- Dinamičko programiranje se izvršava u polinomijalnom vremenu, pod uslovom da je broj potproblema polinomijalan spram ulaza i pod uslovom da se svaki potproblem izvršava u polinomijalnom vremenu

… SEČENJE CEVI …

- Dva pristupa u implementaciji rešenja dinamičkim programiranjem:
  - Top-down pristup sa beleženjem (***top-down with memoization***)

    (*In computing, memoization or memoisation is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.*)
  - Bottom-up pristup
- U top-down pristupu procedure se pozivaju rekurzivno
  - Rezultat svake procedure (potproblema) beleži se u niz ili heš-tabelu
  - Procedura pre svog izvršavanja proveri da li postoji rezultat za dati potproblem
  - Ako rezultat postoji, onda se ono samo preuzme i ne vrši se procesiranja na tom nivou
  - Ako rezultat ne postoji, onda se izvrši procesiranje na tom nivou i taj rezultat se beleži za kasniju upotrebu

… SEČENJE CEVI …

- Bottom-pristup bazira se na poznavanju veličine potproblema
  - Rešenje određenog potproblema (određene veličine) zavisi isključivo od rešenja manjih potproblema (manje veličine)
  - Iskoristimo tu činjenicu, pa sortiramo potprobleme po njihovoj veličini
  - Rešavamo svaki potproblem po njihovoj veličini, počev od najmanjeg ka najvećem
  - Svaki potproblem se rešava samo jednom
  - Kada se naiđe na neki potproblem, svi potproblemi koji čine njegovo rešenje su već rešeni

… SEČENJE CEVI …

- Ovi pristupi dovode do dva rešenja koja imaju isto asimptotsko vreme izvršavanja
- Ipak, svaki ima određenu prednost
- Koju?
- Da li rekurzivni pristup mora proći kroz sve potprobleme?
  - Ako ne prođe, eto ubrzanja
- Takođe, bottom-up pristup ima bolje konstantne faktore, zašto?
  - Nema previše dodatnih poziva procedura

… SEČENJE CEVI …

- Top-down pristup, osnovna procedura

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

- U osnovnom algoritmu inicijalizuje se niz $\boldsymbol{r}[\boldsymbol{0} \ldots \boldsymbol{n}]$ na vrednost $-\infty$
- U datom nizu se beleže rezultati rešenja za pojedinačne potprobleme

… SEČENJE CEVI …

- Top-down pristup, sečenje cevi sa pamćenjem

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = −∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

… SEČENJE CEVI …

- Vreme izvršavanja algoritma baziranog na top-down pristupu iznosi $\Theta(n^2)$
- Zašto?
- Svaki rekurzivni poziv već rešenog problema rezultuje u trenutnom povratku
- Rekurzivni poziv dovodi do rešavanja svakog potproblema samo jednom

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

- Da bi se rešio potproblem n, kod iterira u linijama 6 – 7 n puta
- Ukupan broj iteracija za sve rekurzivne pozive MEMOIZED-CUT-ROD procedure formira aritmetičku progresiju što daje ukupno $\Theta(n^2)$ iteracija

… SEČENJE CEVI …

- Bottom-up pristup

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

- Koristi prirodnu uređenost potproblema: potproblem i je manji od potproblema j ako je $i < j$, te će rešavati potprobleme veličina $j = 0, 1, \ldots, n$ u tom redosledu

… SEČENJE CEVI …

- U liniji 1 se kreira novi niz $r[0 \ldots n]$ u kojem će se čuvati rezultati procesiranja
- U liniji 2 se inicijalizuje $r[0] = 0$, jer cev dužine 0 ne donosi nikakvu zaradu
- Linije od 3 – 6 rešavaju potproblem veličine j, za $j = 0, 1, \ldots, n$ u rastućem redosledu

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

- Problem se rešava slično kao u CUT-ROD proceduri za potproblem veličine j jedino što u liniji 6 nema rekurzivnog poziva, već se tačno referencira vrednost za odgovarajući manji potproblem iz niza $r[j - i]$
- U liniji 7 se čuva rešenje za potproblem veličine j, $r[j]$
- U poslednjoj liniji se vraća $r[n]$ što odgovara optimalnoj vrednosti r_n

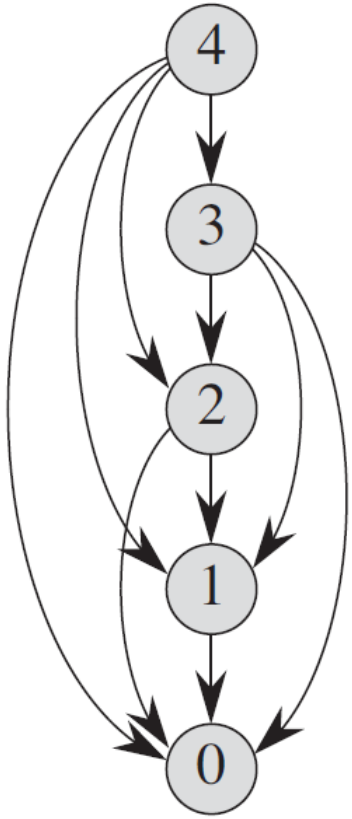
… SEČENJE CEVI

- Vreme izvršavanja algoritma baziranog na top-down pristupu iznosi $\Theta(n^2)$
- Zašto?
- Imamo dve ***for*** petlje !
- Asiptomski su ova dva algoritma iste kompleksnosti, ali to nije tako u implementaciji

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

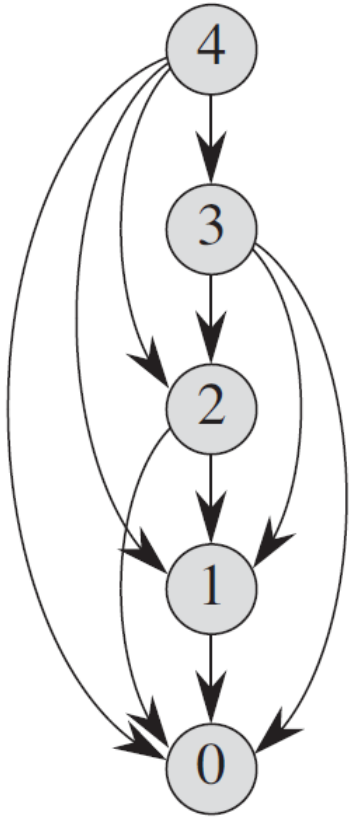
GRAF POTPROBLEMA ...

- Da bi se uopšte došlo do rešenja dinamičkog programiranja, trebalo bi razumeti skup potproblema, kao i odnos između datih potproblema
- Jedan način da se prikaže taj odnos je ***graf potproblema***
- Na slici je prikazano kako on izgleda u slučaju problema sečenja cevi za $n = 4$
- Potproblemi x i y su povezani samo ako se potproblem y posmatra tokom rešavanja potproblema x
- Ovaj graf se može posmatrati kao skupljeno, kondenzovano, rekurzivno stablo za top-down pristup



… GRAF POTPROBLEMA

- Kod bottom-up pristupa, veza ukazuje na to koje potprobleme moramo rešiti prvo, pre nego što pristupimo rešavanju nekog konkretnog potproblema
- Graf potproblema može se iskoristiti za analizu vremena izvršavanja algoritma
  - Kako se svaki potproblem rešava samo jednom, ukupno vreme izvršavanja jednako je vremenu potrebnom da se reši svaki potproblem
  - Tipično vreme potrebno da se reši neki potproblem proporcionalan je broju izlaznih linija odgovarajućeg čvora



REKONSTRUKCIJA REŠENJA ...

- Oba algoritma za rešavanje problema sečenja cevi imaju istu osobinu u pogledu rezultata koji vraćaju
- Koja je to osobina?
- Vraćaju vrednost (kolika je zarada od) optimalnog rešenja, ali ne i izbor koji je doveo do optimalnog rešenja

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2       return r[n]
3  if n == 0
4       q = 0
5  else q = -∞
6       for i = 1 to n
7            q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

… REKONSTRUKCIJA REŠENJA …

- Proširenje bottom-up pristupa
- Za svaku cev dužinu j čuva se, ne samo maksimalna zarada r_n, nego i optimalna dužina s_j, prvog dela koji se seče
- U algoritam se dodaje niz $s[0 \ldots n]$, gde se na odgovarajućoj poziciji čuva dužina prvog odsečenog dela za optimalno rešenje

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

- Na osnovu niza s se lako da rekonstruisati na koje delove se seče cev u optimalno rešenju

… REKONSTRUKCIJA REŠENJA

- Sledeći algoritam uzima niz cena i niz koji čuva dužinu prvog dela na koji je cev sečena i ispisuje optimalnu dekompoziciju cevi dužine ***n***

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n − s[n]
```

- U primeru korišćenom sa početka, ovaj algoritam će ispisati:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

- Da li se iz ovog može zaključiti kolika je veličina preostalih delova na koji se seče cev? Kako?

DINAMIČKO PROGRAMIRANJE, PRIMER – MNOŽENJE LANCA MATRICA

MNOŽENJE LANCA MATRICA …

- Algoritam koji pronalazi optimalno rešenje za množenje niza (lanca) $\langle A_1, A_2, \ldots, A_n \rangle$ matrica, $A_1 \cdot A_2 \cdot \cdots \cdot A_n$
- Standardni metod za množenje matrica se može primeniti, samo se mora voditi računa o tome koji se par matrica u lancu kada množi
- Rešenje je da se koriste zagrade kojima se reguliše koje se dve matrice (uključujući i rezultate) kada množe, tj. kojim redosledom
- Proizvod matrica je potpuno okružen zagradama ***(fully parenthesized)*** samo ako se radi o jednoj matrici ili proizvodu matrica potpuno okruženih zagradama
- Potpuna okruženost zagradama u množenju lanca matrica nije jedinstvena i što je više matrica, to je i više kombinacija
- Na primer, za lanac od 4 matrica, $\langle A_1, A_2, A_3, A_4 \rangle$, postoji 5 različitih kombinacija

… MNOŽENJE LANCA MATRICA …

- Potpuna okruženost zagradama u množenju lanca matrica nije jedinstvena i što je više matrica, to je i više kombinacija
- Na primer, za lanac od 4 matrica, $\langle A_1, A_2, A_3, A_4 \rangle$, postoji 5 različitih kombinacija

$$\left(A_1 \cdot \left(A_2 \cdot (A_3 \cdot A_4)\right)\right)$$

$$\left(A_1 \cdot \left((A_2 \cdot A_3) \cdot A_4\right)\right)$$

$$\left((A_1 \cdot A_2) \cdot (A_3 \cdot A_4)\right)$$

$$\left(\left(A_1 \cdot (A_2 \cdot A_3)\right) \cdot A_4\right)$$

$$\left(\left((A_1 \cdot A_2) \cdot A_3\right) \cdot A_4\right)$$

- Zašto je ovo tako važno? Zar nije sve jedno u kojem redosledu ih množimo?

… MNOŽENJE LANCA MATRICA …

- Nije!
- Redosled množenja matrica može imati dramatičan efekat na vreme izvršavanja algoritma
- Algoritam za množenje dve matrice:

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

- Matrice moraju biti kompatibilne (broj kolona od A i broj vrsta od B)

… MNOŽENJE LANCA MATRICA …

- Zašto nije?
- Primer: neka bude niz od 3 matrice $\langle A_1, A_2, A_3 \rangle$, dimenzija: ***10 x 100***, ***100 x 5***, ***5 x 50***
- Postoje 2 kombinacije $((A_1 \cdot A_2) \cdot A_3)$ i $(A_1 \cdot (A_2 \cdot A_3))$
- Ako se množi u kombinaciji $((A_1 \cdot A_2) \cdot A_3)$, dobija se $10 \cdot 100 \cdot 5 = 5000$ skalarnih množenje kako bi se izračunalo ***10 x 5*** matričnih proizvoda za A_1 i A_2, i još $10 \cdot 5 \cdot 50 = 2500$ skalarnih množenje kako bi se izračunalo ***10 x 50*** matričnih proizvoda za A_3 što ukupno daje ***7500*** skalarnih množenje
- Ako se množi u kombinaciji $(A_1 \cdot (A_2 \cdot A_3))$, dobija se $100 \cdot 5 \cdot 50 = 25000$ skalarnih množenje kako bi se izračunalo ***100 x 50*** matričnih proizvoda za A_2 i A_3, i još $10 \cdot 100 \cdot 50 = 50000$ skalarnih množenje kako bi se izračunalo ***10 x 50*** matričnih proizvoda za A_1 što ukupno daje ***75000*** skalarnih množenje

… MNOŽENJE LANCA MATRICA …

- Ako se množi u kombinaciji $((\boldsymbol{A_1} \cdot \boldsymbol{A_2}) \cdot \boldsymbol{A_3})$, dobija se $\mathbf{100} \cdot \mathbf{100} \cdot \mathbf{5} = \mathbf{5000}$ skalarnih množenje kako bi se izračunalo ***10 x 5*** matričnih proizvoda za $\boldsymbol{A_1}$ i $\boldsymbol{A_2}$, i još $\mathbf{10} \cdot \mathbf{5} \cdot \mathbf{50} = \mathbf{2500}$ skalarnih množenje kako bi se izračunalo ***5 x 50*** matričnih proizvoda za $\boldsymbol{A_3}$ što ukupno daje ***7500*** skalarnih množenje
- Ako se množi u kombinaciji $(\boldsymbol{A_1} \cdot (\boldsymbol{A_2} \cdot \boldsymbol{A_3}))$, dobija se $\mathbf{100} \cdot \mathbf{5} \cdot \mathbf{50} = \mathbf{25000}$ skalarnih množenje kako bi se izračunalo ***100 x 50*** matričnih proizvoda za $\boldsymbol{A_2}$ i $\boldsymbol{A_3}$, i još $\mathbf{10} \cdot \mathbf{100} \cdot \mathbf{50} = \mathbf{50000}$ skalarnih množenje kako bi se izračunalo ***10 x 50*** matričnih proizvoda za $\boldsymbol{A_1}$ što ukupno daje ***75000*** skalarnih množenje
- ***7500*** spram ***75000*** skalarnih množenja, čak ***10*** puta!!!
- Problem množenja lanca matrica: ako imamo lanac od $\boldsymbol{n}$ matrica $\langle \boldsymbol{A_1}, \boldsymbol{A_2}, \dots, \boldsymbol{A_n} \rangle$, gde za $\boldsymbol{i} = \mathbf{0}, \mathbf{1}, \dots, \boldsymbol{n}$, matrica $\boldsymbol{A_i}$ ima dimenzije $\boldsymbol{p_{i-1}} \times \boldsymbol{p_i}$, odrediti potpuna okruženost zagradama tako da se minimizuje broj skalarnih množenja

… MNOŽENJE LANCA MATRICA …

- Kako odrediti potpunu okruženost zagradama tako da se minimizuje broj skalarnih množenja?
- Brutalni pristup, neka je vreme određivanja potpune okruženosti zagradama $\boldsymbol{P(n)}$
- Za $\boldsymbol{n = 1}$, jasno je da postoji samo jedna kombinacija
- Kada je $\boldsymbol{n \geq 2}$, očigledno je da se problem potpune okruženosti zagradama može svesti na problem potpune okruženosti zagradama $\boldsymbol{2}$ podskupa (levog i desnog) originalnog skupa matrica, pri čemu deoba originalno skupa matrica može da nastane bilo gde između $\boldsymbol{k}$-tog i $\boldsymbol{(k+1)}$-og elementa skupa matrica, gde je $\boldsymbol{k = 1, 2, \dots, n-1}$

… MNOŽENJE LANCA MATRICA …

- Očigledno je da je rešenje problema određivanja potpune okruženosti zagradama opisano sledećom rekurentnom jednačinom:

$$P(n) = \begin{cases} 1 & za\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & za\ n \geq 2 \end{cases}$$

- Rešenje ove rekurentne jednačine (bez dokaza) je:

$$\Omega(2^n)$$

- Rešenje je znači eksponencijalno!

- Rešenje problema određivanja potpune okruženosti zagradama može se rešiti i dinamičkim programiranjem (gle čuda)
- Primenićemo četiri koraka opisana na početku

… MNOŽENJE LANCA MATRICA …

1. **Definisanje strukture optimalnog rešenja**
   - Usvojimo prvo da oznaka $\boldsymbol{A_{i\ldots j}}$, gde je $\boldsymbol{i \leq j}$, predstavlja rezultat množenja matrica $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$
   - Problem okruživanja matrica zagradama nije trivijalan, jer da bi se za $\boldsymbol{i \leq j}$ matrični proizvod $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$ okružio zagradama, potrebno je podeliti proizvod između $\boldsymbol{A_k}$ i $\boldsymbol{A_{k+1}}$ tako da je $\boldsymbol{k}$ u opsegu $\boldsymbol{i \leq k < j}$
   - U tom slučaju nastaju dva proizvoda $\boldsymbol{A_{i\ldots k}}$ i $\boldsymbol{A_{k+1\ldots j}}$ koje posle treba da množimo
   - Cena okruživanja zagradama jednaka je sumi cena da se odrede matrice $\boldsymbol{A_{i\ldots k}}$ i $\boldsymbol{A_{k+1\ldots j}}$ plus cena njihovog množenja

… MNOŽENJE LANCA MATRICA …

1. Definisanje strukture optimalnog rešenja (nastavak)

– Optimalna podstruktura problema okruživanja zagradama je sledeća:

- Pretpostaviti da je optimalnim rešenjem proizvod $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$ podeljen i zaokružen zagradama između $\boldsymbol{A_k}$ i $\boldsymbol{A_{k+1}}$
- Onda podlanac matrica $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_k}$ unutar optimalno zaokruženog lanca $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$ takođe mora biti optimalno rešenje
- Zašto?
- Pa ako postoji drugi, optimalan, način zaokruživanja $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_k}$ unutar optimalno zaokruženog lanca $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$, to znači da se inicijalno zaokruženi $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_k}$ lanac mogao zameniti tim drugim što bi dovelo do drugačijeg zaokruživanja lanca $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$ sa manjom cenom, što je kontradikcija
- Isti zaključak može se sprovesti i za podlanac $\boldsymbol{A_{k+1} \cdot A_{k+2} \cdot \cdots \cdot A_j}$

… MNOŽENJE LANCA MATRICA …

1. **Definisanje strukture optimalnog rešenja (nastavak)**
   - Dalje proces ide tako što se optimalno rešenje problema konstruiše iz optimalnih rešenja njegovih potproblema
   - Očigledno se problem rešava, tako što se originalni problem deli na 2 potproblema, pa se traži optimalno rešenje tih potproblema, nakon čega se vrši njihova kombinacija
   - Suština je da se odredi optimalna pozicija za deljenje originalnog lanca matrica, tj. da se provere sve pozicije kako bi se pronašla optimalna pozicija

… MNOŽENJE LANCA MATRICA …

2. Rekurzivno rešenje

- Kao potproblem se uzima određivanje minimalne cene za okruživanje zagradama lanca $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$ za $\boldsymbol{1 \leq i \leq j \leq n}$
- Neka je $\boldsymbol{m[i,\ j]}$ predstavlja minimalni broj skalarnih operacija množenja potrebnih da se odredi matrica $\boldsymbol{A_{i\ldots j}}$; za ceo problem to znači da je $\boldsymbol{m[1,\ n]}$ minimum za $\boldsymbol{A_{1\ldots n}}$
- $\boldsymbol{m[i,\ j]}$ se rekurzivno određuje na sledeći način:
  - Ako je $\boldsymbol{i = j}$ problem je trivijalan, $\boldsymbol{A_{i\ldots j} = A_i}$, tako da je $\boldsymbol{m[i,\ i] = 0}$ za svako $\boldsymbol{i = 1, 2, \ldots, n}$
  - U slučaju da je $\boldsymbol{i < j}$, za izračunavanje $\boldsymbol{m[i,\ j]}$ se može osloniti na optimalno rešenje iz prvog koraka (definisanje strukture optimalnog rešenja)

2. Rekurzivno rešenje (nastavak)

- Ako je za optimalno rešenje potrebno da podelimo lanac $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$ između $\boldsymbol{A_k}$ i $\boldsymbol{A_{k+1}}$, gde je $\boldsymbol{i \leq k < j}$, onda je $\boldsymbol{m[i,\ j]}$ jednaka minimalnoj ceni izračunavanja $\boldsymbol{A_{i\ldots k}}$ i minimalnoj ceni $\boldsymbol{A_{k+1\ldots j}}$ plus cena množenja te dva matrice
- Ako su dimenzije svake matrice tipa $\boldsymbol{p_{i-1} \times p_i}$, onda je za množenje matrica $\boldsymbol{A_{i\ldots k} \cdot A_{k+1\ldots j}}$ je potrebno $\boldsymbol{p_{i-1} \cdot p_k \cdot p_j}$ skalarnih operacija množenja, što dovodi do sledeće jednačine:
  $$\boldsymbol{m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j}$$
- Naravno, ova jednačina podrazumeva da znamo $\boldsymbol{k}$ za optimalno rešenje, što naravno nije još tačno, pa bi trebalo proveriti za svako $\boldsymbol{k}$
- Srećom, nema puno kombinacija, samo $\boldsymbol{j - i}$, za $\boldsymbol{k = i, i+1, \ldots, j-1}$
- Na taj način dolazi se do rekurente jedanačine:

$$\boldsymbol{P(n)} = \begin{cases} \boldsymbol{1} & \boldsymbol{za\ i = j} \\ \min\limits_{\boldsymbol{i \leq k < j}} \{\boldsymbol{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j}\} & \boldsymbol{za\ i < j} \end{cases}$$

… MNOŽENJE LANCA MATRICA …

2. Rekurzivno rešenje (nastavak)

$$\boldsymbol{P(n)} = \begin{cases} \mathbf{1} & \boldsymbol{za\ i = j} \\ \min\limits_{\boldsymbol{i \le k < j}} \{\boldsymbol{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j}\} & \boldsymbol{za\ i < j} \end{cases}$$

- Ova rekurentna jednačina je rešenje određivanja optimalnog okruženja zagradama za lanac matrica $\boldsymbol{A_i \cdot A_{i+1} \cdot \cdots \cdot A_j}$
- U ovom konkretnom primeru dinamičkog programiranja, minimalna cena da se formira okruženost zagradama, $\boldsymbol{m[i,j]}$, nije dovoljna, jer je potrebno i odrediti kako pozicionirati same zagrade
- Tako da pored $\boldsymbol{m[i,j]}$, pamti se $\boldsymbol{k}$ po kojem je vršena optimalna deoba lanca matrica, ovo se čuva unutar strukture $\boldsymbol{s[i,j]}$

3. Izračunavanje optimalne cene

- Lako se može napraviti rekurzivni algoritam koji bi implementirao na osnovu rekurentne jednačina:

$$P(n) = \begin{cases} 1 & za\ i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\} & za\ i < j \end{cases}$$

- Međutim, kao i brutalni pristup, ovaj algoritam bi imao eksponencijalno vreme izvršavanja, jer bi ispitivao svaku granu stabla rekurzije, iako se pojedini delovi ponavljaju
- Primetiti da postoji relativno mali broj potproblema za problem $A_i \cdot A_{i+1} \cdot \cdots \cdot A_n$ i za i i j koji zadovoljavaju uslov $1 \le i \le j \le n$, svodi se na problem kombinatorike $\binom{n}{2} + n = \Theta(n^2)$
- Imamo optimalnu strukturu + preklapanje potproblema ukazuju na potrebu za dinamičkim programiranjem

… MNOŽENJE LANCA MATRICA …

3. Izračunavanje optimalne cene (nastavak)

- Bottom-up rešenje za potpuno okruživanje zagradama množenja lanca matrica

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

3. Izračunavanje optimalne cene (nastavak)

- Pretpostaviti da je svaka matrica $\boldsymbol{A_i}$ dimenzija $\boldsymbol{p_{i-1} \times p_i}$, za $\boldsymbol{i = 1, 2, \dots, n}$
- Ulaz algoritma je sekvenca dimenzija matrica koje čine lanac množenja, $\boldsymbol{p = \langle p_0, p_1, \dots, p_n \rangle}$ pri čemu važi:
  $$\boldsymbol{p.length = n + 1}$$
- Rezultati rada procedura smeštaju se u dve matrice: $\boldsymbol{m[1 \dots n, 1 \dots n]}$ u kojoj se čuva cena i $\boldsymbol{s[1 \dots n-1, 2 \dots n]}$ u kojoj se čuva pozicija na kojoj je izvršeno optimalno deljenje lanca matrica prilikom računanja $\boldsymbol{m[i, j]}$

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

- Potrebno je odrediti kojim se sve elementima matrice $\boldsymbol{m[1 \dots n, 1 \dots n]}$ pristupa kada se računa $\boldsymbol{m[i, j]}$

3. Izračunavanje optimalne cene (nastavak)

- Potrebno je odrediti kojim se sve elementima matrice $\boldsymbol{m[1 \ldots n, 1 \ldots n]}$ pristupa kada se računa $\boldsymbol{m[i,j]}$
- Kada se računa cena $\boldsymbol{m[i,j]}$ onda postoji lanac matrica koje se množe dužine $\boldsymbol{j-i+1}$
- u svakom slučaju cena za $\boldsymbol{m[i,j]}$ će se računati na lancu matrica čija dužina je manja od $\boldsymbol{j-i+1}$
- Tako da za $\boldsymbol{k = i, i+1, \ldots, j-1}$, matrica $\boldsymbol{A_{i \ldots k}}$ je proizvod $\boldsymbol{k-i+1 < j-i+1}$ matrica i matrica $\boldsymbol{A_{k+1 \ldots j}}$ je proizvod $\boldsymbol{j-k < j-i+1}$

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

- Tako da algoritam koji popunjava tabelu $\boldsymbol{m[i,j]}$ trebalo bi to da radi u rastućem redosledu
- Tako da se problem optimizacije $\boldsymbol{A_{ij}}$ svodi na rešavanje $\boldsymbol{j-i+1}$ potproblema

3. Izračunavanje optimalne cene (nastavak)

- Algoritam prvo računa $\boldsymbol{m[i,i]=0}$ za svako $\boldsymbol{i=1,2,\ldots,n}$, minimalna cena lanac dužine $\boldsymbol{1}$, linije 3-4
- U sledećem koraku se računa cena za svaki lanac dužine $\boldsymbol{l=2}$, $\boldsymbol{m[i,i+1]}$ za svako $\boldsymbol{i=1,2,\ldots,n-1}$, prva iteracija ***for*** petlje, linija 5-13
- U drugom koraku se računa cena za svaki lanac dužine $\boldsymbol{l=3}$, $\boldsymbol{m[i,i+2]}$ za svako $\boldsymbol{i=1,2,\ldots,n-2}$, prva iteracija ***for*** petlje, linija 5-13, i tako dalje

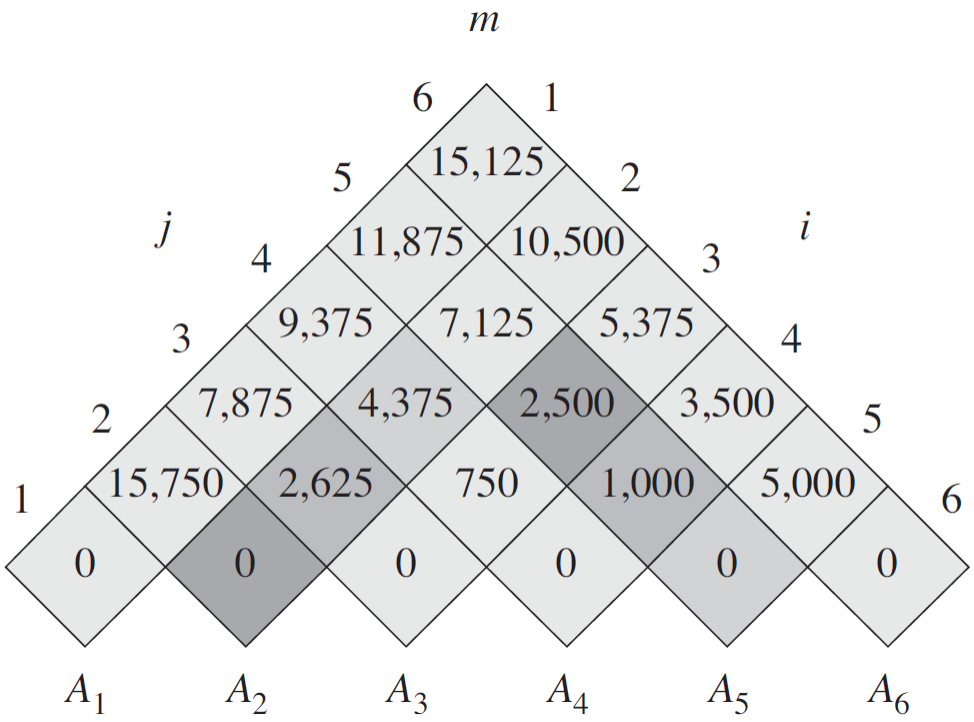
```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

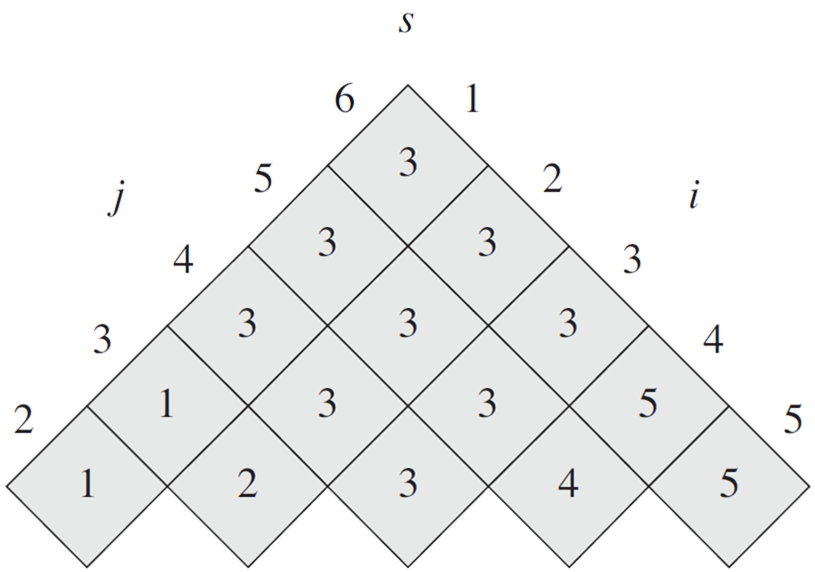
- U svako koraku se cena $\boldsymbol{m[i,j]}$ računa samo na osnovu zapisa u tabeli $\boldsymbol{m[i,k]}$ i $\boldsymbol{m[k+1,j]}$, koji su već izračunati
- Kako izgleda ceo postupak za niz od 6 matrica

3. Izračunavanje optimalne cene (nastavak)

– Kako izgleda ceo postupak za niz od 6 matrica

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25



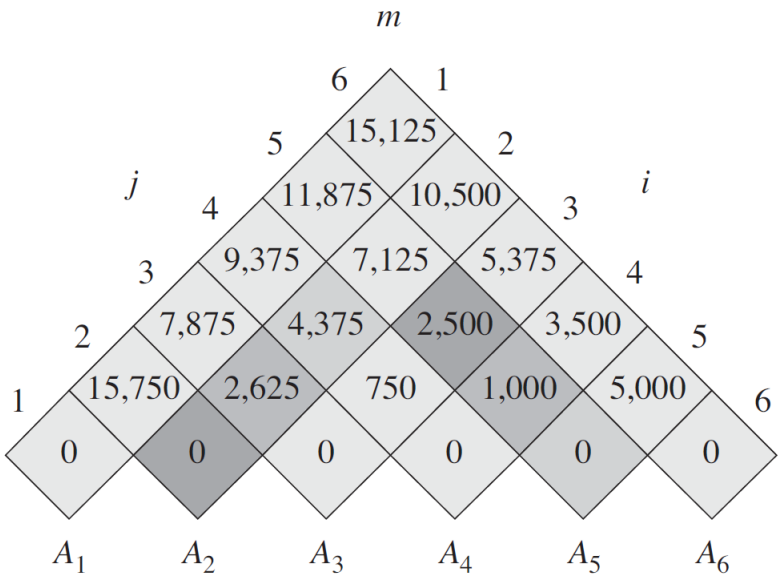


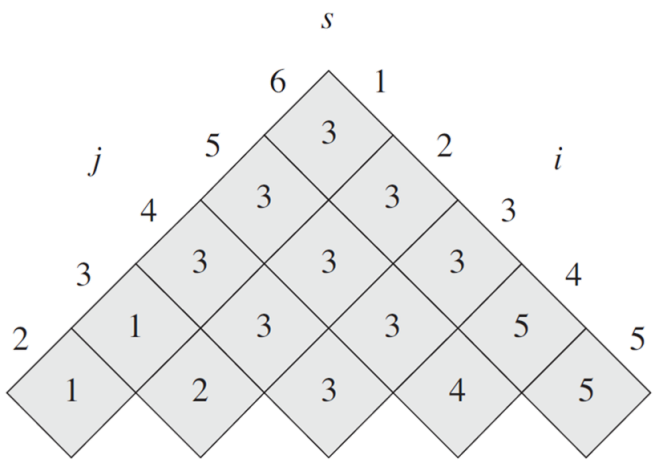
$$\min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j \}$$

3. Izračunavanje optimalne cene (nastavak)

– Kako izgleda ceo postupak za niz od 6 m.

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





$$\min_{i \le k < j} \{ m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j \}$$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

… MNOŽENJE LANCA MATRICA …

3. Izračunavanje optimalne cene (nastavak)

- Vreme izvršavanja algoritma je $\boldsymbol{O(n^3)}$ što je daleko bolje od eksponencijalnog vremena koje zahteva brutalni pristup
- Zašto?
- Koliko ima ***for*** petlji?
- Takođe je i najkraće vreme izvršavanja $\Omega(\boldsymbol{n^3})$
- Zašto?
- Da li je moguće izbeći ili smanjiti vreme izvršavanja bilo koje od petlji?
- Tako da je vreme izvršavanja algoritma $\Theta(\boldsymbol{n^3})$

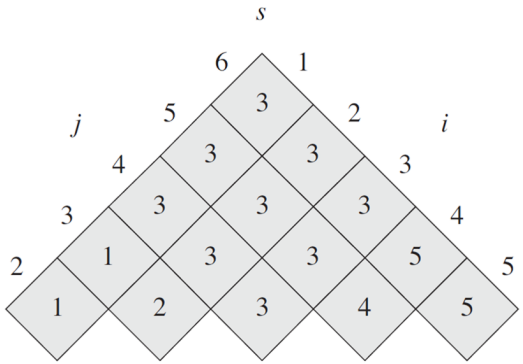
```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

… MNOŽENJE LANCA MATRICA …

4. Ispisivanje optimalnog rešenja

- Prethodni algoritam izračunava cenu i mesto na kojem treba da se podeli lanac matrica koje se množe, ali ne ispisuje rešenje sa potpunom okruženošću zagradama
- Kako se to može uraditi? Pravljenjem dodatnog rekurzivnog algoritma za ispis na osnovu matrice **s**

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2        print "A"_i
3  else print "("
4        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6        print ")"
```

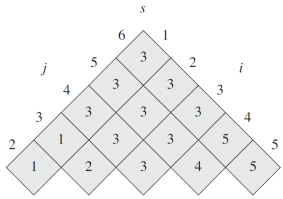


- Zašto ovo radi?

4. Ispisivanje optimalnog rešenja (nastavak)

- Matrica $s[1 \ldots n-1, 2 \ldots n]$ sadrži sve informacije potrebne da se restaurira optimalno rešenje
- Zato što svaki zapis $s[i,j]$ čuva k vrednosti u kojoj dolazi do podele lanca matrica koje se množe

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2        print "A"_i
3  else print "("
4        PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5        PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6        print ")"
```



- To znači da se inicijalno lanac matrica deli na $A_{i \ldots s[i,j]} A_{s[i,j]+1 \ldots n}$ i tako dalje
- Ostatak mesta na kojima dolazi do deljenja može se odrediti rekurzivno, jer je zagrada određena deobom $s[1,n]$ ujedno i poslednja koja bi trebala da se ispiše (osim one jedne trivijalne)
- U primeru:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

rešenje je $\big((A_1 \cdot (A_2 \cdot A_3))((A_4 \cdot A_5) \cdot A_6)\big)$

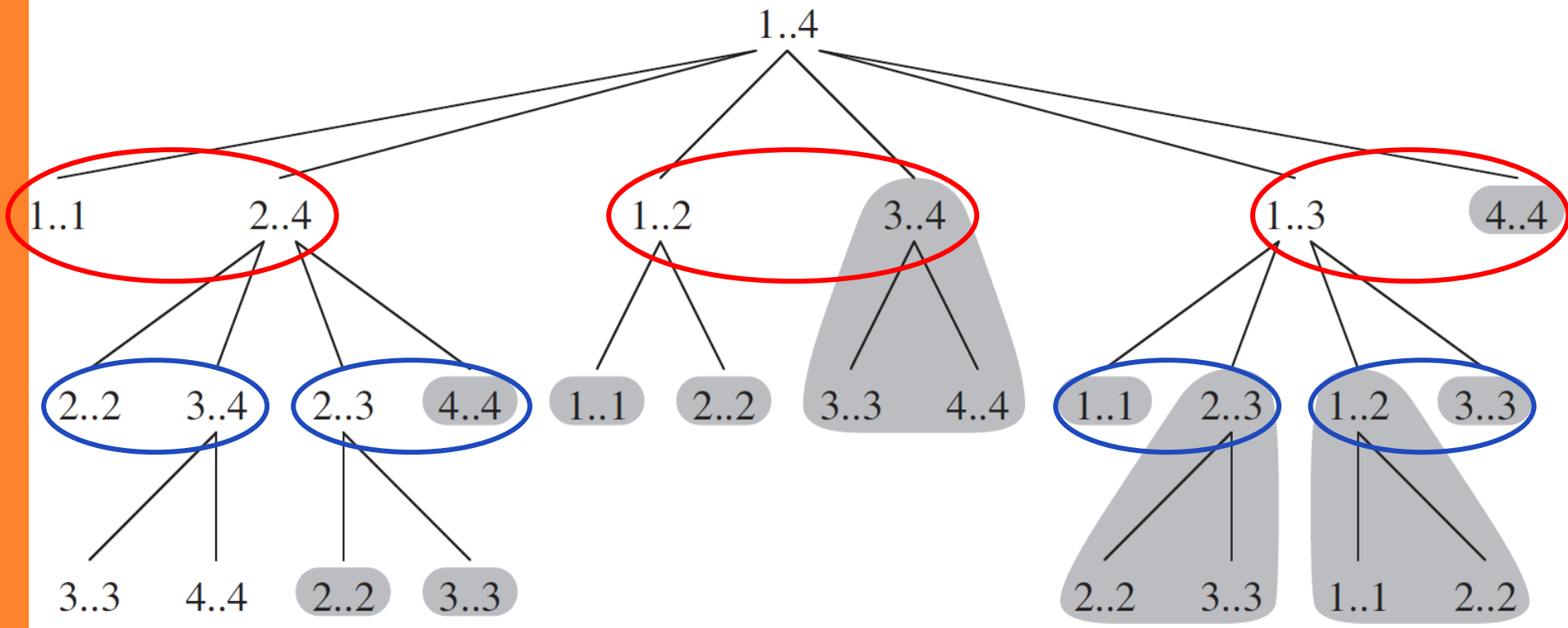
MNOŽENJE LANCA MATRICA – REKURZIJA …

- Da bi se u lancu množenje matrica odredila cena $\boldsymbol{m}[\boldsymbol{i},\boldsymbol{j}]$ potrebno je odrediti cene i za potprobleme
- Rekurzivni algoritam koji je suštinski neefikasan ima sledeći izgled

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2        return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5        q = RECURSIVE-MATRIX-CHAIN(p, i, k)
                 + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
                 + p_{i−1} p_k p_j
6        if q < m[i, j]
7              m[i, j] = q
8  return m[i, j]
```

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Stablo rekurzije ovog algoritma pokazuje jasno preklapanje potproblema



… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Vreme da se odredi $\boldsymbol{m[1,n]}$ je eksponencijalno
  - Izvršavanje algoritma u koracima 1-2 i 6-7, kao i množenje u 5 traju makar jednu jedinicu vremena, tako da nastaje rekurentna jednačina:

$$\boldsymbol{T(n) \geq 1},$$

$$\boldsymbol{T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)}$$

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

  - Kako se svako $\boldsymbol{i = 1, 2, \dots, n-1}$ u $\boldsymbol{T(i)}$ pojavljuje barem 2 puta, jednom kao $\boldsymbol{T(k)}$, drugi put kao $\boldsymbol{T(n-k)}$, i kad se na $\boldsymbol{n-1}$ jedinicu doda još ono 1 sa početka rekurentne jednačine, dobija se

$$\boldsymbol{T(n) \geq 2 \cdot \sum_{i=1}^{n-1} T(i) + n}$$

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Da bi se pokazalo da je vreme potrebno da se odredi $\boldsymbol{m[1,n]}$ eksponencijalno, treba dokazati da je $\boldsymbol{T(n) = \Omega(2^n)}$
- Zapravo se metodom supstitucije može dokazati da je $\boldsymbol{T(n) \geq 2^{n-1}}$, za $\boldsymbol{n \geq 1}$

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

$$\boldsymbol{T(1) \geq 1 = 2^0}$$

$$\boldsymbol{T(n) \geq 2 \cdot \sum_{i=1}^{n-1} 2^{i-1} + n = 2 \cdot \sum_{i=0}^{n-2} 2^i + n}$$

$$\boldsymbol{= 2 \cdot \left(2^{n-1} - 1\right) + n = 2^n - 2 + n \geq 2^{n-1}}$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

- Očigledno je da RECURSIVE-MATRIX-CHAIN(*p*, *1*, *n*) u najmanju ruku eksponencijalan u $\boldsymbol{n}$

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Za razliku od neefikasnog rekurzivnog pristupa, bottom-up rešenje rešava svaki od $\Theta(n^2)$ potproblema samo jednom
- Alternativa, u rekurzivni algoritam dodati beleženje već izračunatih vrednosti
- Algoritam ima dva dela:
  - Glavnu proceduru koja inicijalizuje početne vrednosti i pokreće rekurzivni algoritam sa beleženjem
  - Rekurzivni algoritam sa beleženjem
- I ovde se beleži cena određenog podlanca u matricu $m[i, j]$ samo se ovde inicijalno vrednosti postavljaju ∞
- ***Zašto?***
- Rekurzivni algoritam se modifikuje tako da se prvo proveri da li je vrednost izračunata, ako jeste samo se vrati ta vrednost nazad, ako ne izračuna se u datoj realizaciji

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Osnovni algoritam:

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4       for j = i to n
5            m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Rekurzivni algoritam sa beleženjem:

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2        return m[i, j]
3  if i == j
4        m[i, j] = 0
5  else for k = i to j − 1
6              q = LOOKUP-CHAIN(m, p, i, k)
                   + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7              if q < m[i, j]
8                    m[i, j] = q
9  return m[i, j]
```

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- I algoritam baziran na bottom-up pristupu izvršava se u $\boldsymbol{O(n^3)}$ vremena
- **Zašto?**
- Očigledno se 5 linija u glavnoj proceduri izvršava u $\Theta(\boldsymbol{n^2})$ vremena

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4       for j = i to n
5            m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2       return m[i, j]
3  if i == j
4       m[i, j] = 0
5  else for k = i to j − 1
6            q = LOOKUP-CHAIN(m, p, i, k)
                 + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7            if q < m[i, j]
8                 m[i, j] = q
9  return m[i, j]
```

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Što se tiče rekurzivnog dela:
  - ako je $\boldsymbol{m[i,j] = \infty}$, onda se izvršavaju linije od 3-9
  - ako je $\boldsymbol{m[i,j] < \infty}$, algoritam završava u 2.oj liniji
  - Prvi slučaj se izvršava $\boldsymbol{\Theta(n^2)}$ puta, jer toliko ima el. u tabeli
  - Svi pozivi drugog tipa su unutar rekurzivnih poziva prvog tipa
  - Kad god se desi rekurzivni poziv LOOKUP-CHAIN metode, desi $\boldsymbol{O(n)}$ takvih poziva
  - Ukupno, poziva drugog tipa ima $\boldsymbol{O(n^3)}$
  - Ukupno vreme čitavog top-down algoritma je, dakle, $\boldsymbol{O(n^3)}$

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

… MNOŽENJE LANCA MATRICA – REKURZIJA …

- Poređenje bottom-up i top-down pristupa:

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
                + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

… MNOŽENJE LANCA MATRICA – REKURZIJA

- Da li odabrati bottom-up ili top-down pristup, zavisi od toga da li treba proći kroz sve potprobleme ili se može desiti situacija da se ne mora proći kroz sve

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
                + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

DINAMIČKO PROGRAMIRANJE, ELEMENTI

KLJUČNI ELEMENTI PROBLEMA

- Da bi optimizacioni problem bio kandidat za rešavanja dinamičkim programiranjem, mora imati dva ključna elementa:
  - Da li postoji optimalna podstruktura, tj. da li se rešenje problema može naći unutar optimalnih rešenja njegovih potproblema
  - Potprobleme koji se preklapaju, tj. rešavanje osnovnog problema parcionisanjem na potprobleme dovodi do ponovljenog rešavanja nekog ili svih potproblema po nekoliko puta

OPTIMALNA PODSTRUKTURA …

- Gradimo optimalno rešenje problema kroz optimalna rešenja potproblema
  - Mora se voditi računa da se uključe svi potproblemi
- Do optimalne podstrukture se dolazi praćenjem nekoliko standardnih koraka

… OPTIMALNA PODSTRUKTURA …

- Do optimalne podstrukture se dolazi praćenjem nekoliko standardnih koraka:
  1. Rešenje problema zavisi od nekakvog izbora (gde preseći cev), što implicira da postoji jedan ili više potproblema.
  2. Pretpostavljamo da za dati problem, izbor dovodi do optimalnog rešenja (ne razmatramo kako se do njega dolazi)
  3. U skladu sa izborom, prepoznajemo koji potproblemi nastaju i kako na najbolji način okarakterisati prostor potproblema
  4. Pokazuje se da je rešenje potproblema (kojim se dolazi do optimalnog rešenja problema) takođe optimalno rešenje primenom isecanje-pa-ubacivanje tehnike – iseca neoptimalno rešenje i ubacuje se optimalno. Obično se dokazuje da je rešenje optimalno primenom kontradikcije

… OPTIMALNA PODSTRUKTURA …

- Prilikom karakterisanja prostora potproblema, važno je prostor držati što je moguće jednostavnim (jedno sečenje cevi), a proširivati ga samo kada je neophodno
  - Na primer, kod lanca matrica mogao se problem ograničiti na jednostavniji slučaj, da deoba lanca matrica uvek ide samo sa jednog kraja ($\boldsymbol{A_1 A_2 \dots A_j}$), međutim da bi ovo bilo zadovoljeno deoba bi trebala uvek da dovode do rešenja $\boldsymbol{A_1 A_2 \dots A_{j-1}}$ i $\boldsymbol{A_j}$, što ne garantuje optimalno rešenje, te zbog toga optimalno rešenje poptroblema ide sa obe strane $\boldsymbol{A_i A_{i+1} \dots A_j}$

… OPTIMALNA PODSTRUKTURA …

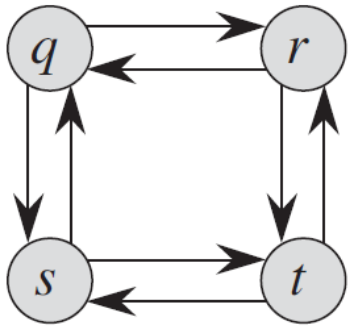
- Optimalna struktura se razlikuje od problema do problema u tome:
  - Koliko ima potproblema za optimalno rešenje problema
  - Koji broj izbora postoji u određivanju koje potprobleme koristiti u rešavanju optimalnog problema
- Kod sečenja cevi postoji samo jedan potproblem ($\boldsymbol{n} - \boldsymbol{i}$), ali $\boldsymbol{n}$ izbora kako odrediti $\boldsymbol{i}$
- Kod lanca matrica postoje dva potproblema (podela na dva podlanca, koji se svaki deli dalje) i $\boldsymbol{j} - \boldsymbol{i}$ izbora

… OPTIMALNA PODSTRUKTURA …

- Vreme izvršavanja algoritma (neformalno) zavisi od toga koliko imamo potproblema i koji je broj izbora
  - Kod sečenja cevi postojalo je sveukupno $\Theta(\boldsymbol{n})$ rešenja potproblema i najviše $\boldsymbol{n}$ izbora, što daje vreme izvršavanja od $\boldsymbol{O(n^2)}$
  - Kod lanca matrica postojalo je sveukupno $\Theta(\boldsymbol{n^2})$ rešenja potproblema i najviše $\boldsymbol{n-1}$ izbora, što daje vreme izvršavanja od $\boldsymbol{O(n^3)}$, tj. $\Theta(\boldsymbol{n^3})$
- U većini slučajeva graf potproblema daje dovoljno uvida u samo rešavanja problema i alternativan način da se sprovede analiza
  - Kod sečenja cevi bilo $\boldsymbol{n}$ čvorova i $\boldsymbol{n}$ linija (ivica), što znači ukupno $\boldsymbol{O(n^2)}$ čvorova i ivica
  - Kod lanca matrica (da smo ga crtali) stablo bi imalo $\Theta(\boldsymbol{n^2})$ čvorova i najviše $\boldsymbol{n-1}$ ivica, što daje ukupno $\boldsymbol{O(n^3)}$ čvorova i ivica

… OPTIMALNA PODSTRUKTURA

- Često se koristi bottom-up pristup, pa se prvo rešavaju potproblemi i računa se njihova cena izvršavanja, pa tek posle na osnovu njih optimalno rešenje poblema
- Treba biti pažljiv da se ne primeni dinamičko programiranje tamo gde nema optimalnog rešenja, već samo njegov privid
  - Primer *Unweighted shortest / longest simple path* iz ItA
  - Shortest je po svojoj prirodi optimalan, dok longest to nije zbog načina na koji se kombinuju čvorovi



PREKLAPANJE POTPROBLEMA

- Druga osobina algoritama dinamičkog programiranja je da uvek prostor potproblema takav da se potproblemi ponavljaju, tj. nije svaki potproblem drugačiji
- Najčešće je broj potproblema polinomijalan
- Kada rekurzivni algoritam rešava isti potproblem više puta, kaže se da on iskazuje preklapanje potproblema
- Kod divide-and-conqere algoritama, svaki novi korak rekurzije generiše novi potproblem