Dinamičko programiranje

Dinamičko programiranje rešava probleme kombinovanjem rešenja potproblema. Za razliku od pristupa "podeli i osvoji", dinamičko programiranje se primenjuje kod problema čiji potproblemi nisu nezavisni, već potproblemi imaju neke zajedničke potprobleme. U takvim slučajevima bi "podeli i osvoji" algoritmi bespotrebno više puta rešavali iste potprobleme. Algoritmi razvijeni na principima dinamičkog programiranja rešavaju svaki potproblem samo jednom i čuvaju njegovo rešenje u tabeli. Na taj način oni izbegavaju ponovno rešavanje tog potproblema svaki put kada se on pojavi.

Dinamičko programiranje se obično koristi za rešavanje problema optimizacije. Ovakvi problemi mogu imati više rešenja. Svako rešenje ima neku vrednost, a cilj je pronaći rešenje sa optimalnom (minimalnom ili maksimalnom) vrednošću. Ovakvo rešenje nazivamo jednim od optimalnih rešenja, pošto može postojati više rešenja koje dostižu istu vrednost.

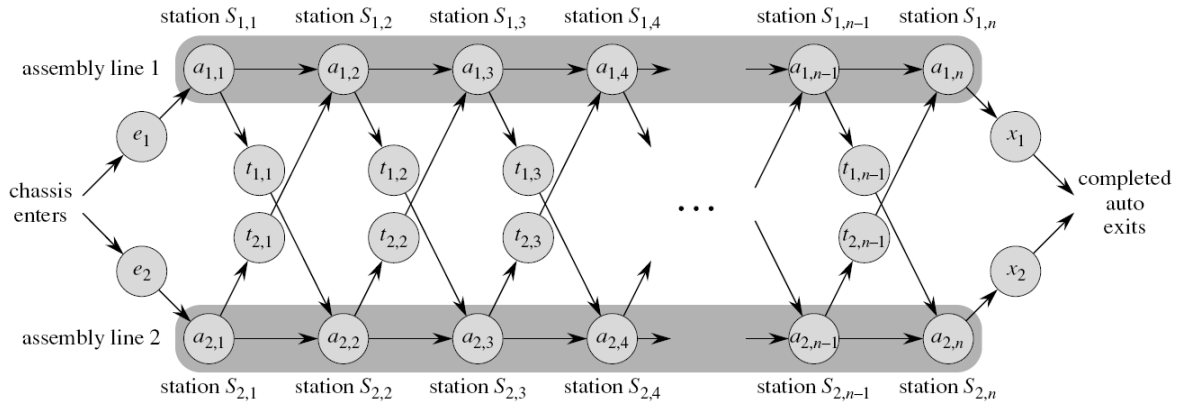
Razvoj algoritma dinamičkog programiranja se može razložiti na četiri koraka:

1. Odrediti strukturu optimalnog rešenja
2. Rekurzivno definisati vrednost optimalnog rešenja
3. Izračunati vrednost optimalnog rešenja po principu "odozdo na gore"
4. Konstruisati optimalno rešenje na osnovu izračunatih informacija

Koraci 1-3 čine osnovu rešavanja problema korišćenjem dinamičkog programiranja. Korak 4 se može preskočiti ukoliko je tražena samo vrednost optimalnog rešenja. Da bismo izvršili korak 4, ponekad je potrebno sačuvati i neke dodatne informacije tokom izračunavanja u koraku 3, kako bismo lakše konstruisali optimalno rešenje.

Organizovanje linija za sklapanje automobila

Prvi primer na kome ćemo proučiti dinamičko programiranje rešava problem u proizvodnji automobila. Pretpostavimo da fabrika automobila ima dve linije za sklapanje automobila, kao što je prikazano na Slici ###.



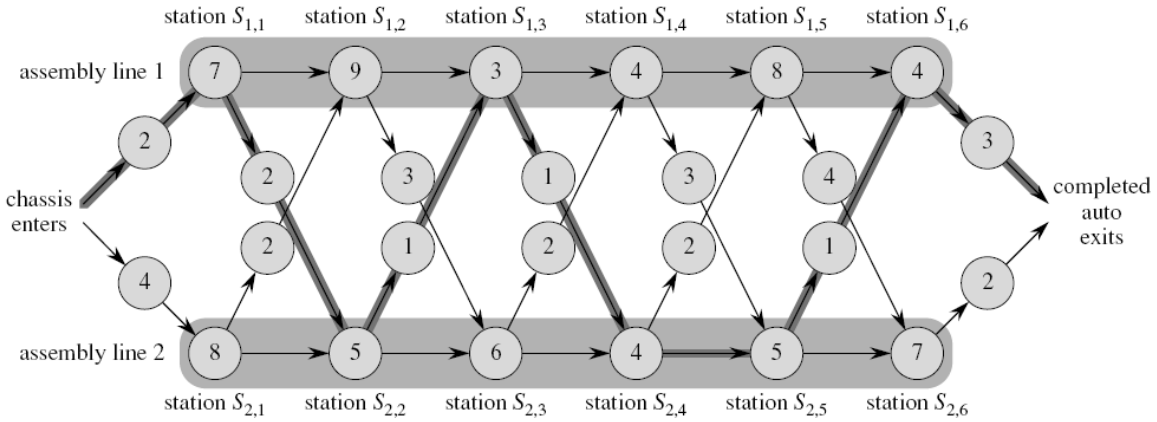
Slika ###. Problem pronalaženja najbržeg puta u proizvodnji

Karoserija automobila ulazi na liniju, na različitim stanicama na karoseriju se dodaju delovi i na kraju kompletiran auto izlazi sa linije. Svaka linija ima n stanica, označenih sa j=1,2,...,n. Označimo sa $S_{i,j}$ j-tu stanicu na i-toj liniji (pri čemu i može biti 1 ili 2). Stanica j na liniji 1 ($S_{i,j}$) obavlja istu operaciju kao i stanica j na liniji 2 ($S_{i,j}$). Međutim, stanice su izgrađene u različito vreme i različitim tehnologijama, tako da je vreme potrebno za obavljanje operacije na svakoj stanici različito, čak i za stanice na istim pozicijama ali na različitim linijama. Označimo vreme potrebno za obavljanje operacije na stanici $S_{i,j}$ sa $a_{i,j}$. Kao što se može videti na Slici ###, karoserija ulazi na stanicu 1 na jednoj od linija i kreće se dalje od stanice do stanice.

Takođe, postoji vreme e_i potrebno da karoserija uđe na liniju i, kao i vreme x_i potrebno da kompletiran automobil izađe sa linije i.

Kada karoserija uđe na jednu liniju, ona se kreće samo po toj liniji. Vreme potrebno da karoserija pređe od jedne stanice do druge je zanemarivo. Međutim, desilo se da naručilac zahteva što bržu isporuku automobila. U tom slučaju karoserija i dalje mora da prođe svih n stanica, ali radnici mogu da prebacuju nepotpuno sklopljen automobil sa jedne linije na drugu, nakon bilo koje stanice. Vreme potrebno za prebacivanje karoserije sa linije i nakon što je prošla kroz stanicu $S_{i,j}$ je $t_{i,j}$, pri čemu je i=1,2,...,n-1 (zato što je nakon n-te stanice automobil već kompletiran. Problem je odrediti kroz koje stanice na liniji 1 i koje stanice na liniji 2 karoserija treba da prođe da bi se minimizovalo vreme potrebno za proizvodnju jednog automobila. U primeru na Slici ###a, najkraće vreme se dobija izborom stanica 1, 3 i 6 sa linije 1 i stanica 2, 4 i 5 sa linije 2.

Očigledno, primena "sirove sile" za minimizaciju vremena potrebnog za proizvodnju automobila nije moguća u slučajevima kada postoji veliki broj stanica. Ukoliko imamo listu stanica kroz koje karoserija prolazi na liniji 1 i listu stanica na liniji 2, lako je izračunati u vremenu $O(n)$ koliko traje prolazak kroz fabriku. Na žalost, postoji 2^n mogućih načina da se izaberu stanice, tako da bi za određivanje najbržeg puta bilo potrebno $O(2^n)$ vremena, što je za veliko n neprihvatljivo.



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

(b)

Slika ###. a) Primer problema linija za sklapanje automobila. b) Vrednosti za $f_i[j]$, f^*, $l_i[j]$ i l^* za primer dat pod a).

Korak 1: Struktura najbržeg puta kroz fabriku

Prvi korak u paradigmi dinamičkog programiranja je određivanje strukture optimalnog rešenja. Za problem organizovanja linija za sklapanje automobila ovaj korak možemo uraditi na sledeći način. Razmotrimo najbrži način da karoserija sa početne tačke prođe kroz stanicu $S_{1,j}$. Ako je j=1, postoji samo jedan put kojim karoserija može da prođe, tako da je jednostavno odrediti koliko je potrebno da karoserija prođe kroz stanicu $S_{1,j}$. Međutim, za j=2,3,...,n postoje dva izbora. Karoserija može doći sa stanice $S_{1,j-1}$ i direktno otići do stanice $S_{1,j}$, pri čemu smo naglasili da je vreme za prelazak sa jedne stanice na drugu zanemarivo. Drugi način je da karoserija stigne sa stanice $S_{2,j-1}$ i da se zatim prebaci na stanicu $S_{1,j}$, za šta je potrebno utrošiti $t_{2,j-1}$ vremena. Razmotrićemo ova dva slučaja odvojeno, pri čemu ćemo uočiti da oni imaju dosta toga zajedničkog.

Pretpostavimo prvo da je najbrži put do stanice $S_{1,j}$ kroz stanicu $S_{1,j-1}$. Ključno je uočiti da je karoserija morala doći najbržim putem od početne tačke do stanice $S_{1,j-1}$. Zašto? Da je postojao brži način da karoserija stigne do stanice $S_{1,j-1}$, onda bi taj put omogućio i brži put do stanice $S_{1,j}$, što je u suprotnosti sa pretpostavkom.

Slično, pretpostavimo da je najbrži put do stanice $S_{1,j}$ kroz stanicu $S_{2,j-1}$. Sada uočavamo da je karoserija morala doći najbržim putem od početne tačke do stanice $S_{2,j-1}$. Kao i u prethodnom slučaju, zaključujemo slično: da je postojao brži način da karoserija stigne do stanice $S_{2,j-1}$, onda bi taj put omogućio i brži put do stanice $S_{1,j}$, što je opet u suprotnosti sa pretpostavkom.

U opštem slučaju možemo reći da za proizvodne linije optimalno rešenje problema (nalaženje najbržeg puta do stanice S_{ij}) sadrži u sebi optimalna rešenja potproblema (nalaženje najbržeg puta do stanica $S_{1,j-1}$ i $S_{2,j-1}$). Ovu osobinu nazivamo **optimalna podstruktura**, koja je ključna za primenu dinamičkog programiranja.

Koristimo optimalnu podstrukturu da bismo pokazali kako možemo formirati optimalno rešenje problema na osnovu optimalnih rešenja potproblema. U slučaju proizvodnih linija razmišljamo na sledeći način. Ako posmatramo najbrži put do stanice $S_{1,j}$, on mora da vodi preko stanice j-1 ili na linji 1 ili liniji 2. Onda je najbrži put do stanice $S_{1,j}$ ili

- najjbrži put do stanice $S_{1,j-1}$, a zatim direktno do stanice $S_{1,j}$ ili
- najjbrži put do stanice $S_{2,j-1}$, a zatim prebacivanjem sa linije 2 na liniju 1 do stanice $S_{1,j}$.

Korišćenjem simetričnog pristupa, najbrži put do stanice $S_{2,j}$ je ili

- najjbrži put do stanice $S_{2,j-1}$, a zatim direktno do stanice $S_{2,j}$ ili
- najjbrži put do stanice $S_{1,j-1}$, a zatim prebacivanjem sa linije 1 na liniju 2 do stanice $S_{2,j}$.

Da bismo rešili problem nalaženja najbržeg puta do stanice j na bilo kojoj liniji, rešavamo potprobleme nalaženja najbržih puteva do stanica j-1 na obe linije. Na ovaj način možemo formirati optimalno rešenje problema, formiranjem optimalnih rešenja za potprobleme.

Korak 2: Rekurzivno rešenje

Drugi korak u paradigmi dinamičkog programiranja je rekurzivno određivanje vrednosti nekog od optimalnih rešenja u funkciji od optimalnih rešenja potproblema. U slučaju organizovanja proizvodnih linija, za potprobleme uzimamo određivanje najbržeg puta to stanice j na obe linije, za j=1,2,...,n. Neka je $f_i[j]$ najkraće moguće vreme da karoserija stigne od startne pozicije do stanice $S_{i,j}$.

Naš primarni cilj je da odredimo najkraće vreme da karoserija prođe ceo put kroz fabriku, koje ćemo označiti sa f^*. Karoserija treba da prođe kroz svih n stanica bilo na liniji 1 ili na liniji 2 i da zatim napusti fabriku. Samim tim, brži od ovih puteva je i najbrži put kroz celu fabriku, pa imamo da je

$$f^* = \min\left(f_1[n] + x_1, f_2[n] + x_2\right)$$

Takođe je lako doneti zaključak o $f_1[1]$ i $f_2[1]$. Da bi se prošlo kroz prvu stanicu na bilo kojoj traci, karoserija direktno ulazi na tu stanicu, pa je odatle

$$f_1[1] = e_1 + a_{1,1}$$
$$f_2[1] = e_2 + a_{2,1}$$

Razmotrimo sada kako izračunati $f_i[j]$ za j=2,3,...n i i=1,2.

Ako se usresredimo na $f_1[j]$, prisetimo se da je najbrži put do stanice $S_{1,j}$ ili najbrži put do stanice $S_{1,j-1}$, a zatim direktno u $S_{1,j}$ ili najbrži put do stanice $S_{2,j-1}$, a zatim prebacivanje sa linije 2 na liniju 1 u stanicu $S_{1,j}$. U prvom slučaju imamo da je $f_1[j] = f_1[j-1] + a_{1,j}$, a u drugom slučaju $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$. Odatle je

$f_1[j] = \min\left(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j} \right)$, za j=2,3,...,n.

Simetrično imamo i da je

$f_2[j] = \min\left(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j} \right)$, za j=2,3,...,n.

Kombinovanjem prethodnih jednačina dobijamo rekurzivne jednačine

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & , j = 1 \\ \min\left(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j} \right) & , j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & , j = 1 \\ \min\left(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j} \right) & , j \geq 2 \end{cases}$$

Na Slici ###b su prikazane izračunate vrednosti $f_i[j]$ za primer prikazan na Slici ###a, kao i vrednost f^*.

Vrednosti $f_i[j]$ daju vrednosti optimalnih rešenja potproblema. Da bismo mogli da konstruišemo optimalno rešenje, definisaćemo $l_i[j]$ kao brojeve linija (1 ili 2) čija je stanica j-1 iskorišćena za najbrži dolazak do stanice $S_{i,j}$, pri čemu je i=1,2 i j=2,3,...,n. Definisaćemo takođe i l^* kao liniju čija je stanica n iskorišćena za najbrži put kroz celu fabriku. Korišćenjem vrednosti l^* i $l_i[j]$ prikazanih na Slici ###b, odredićemo najbrži put kroz fabriku prikazanu na Slici ###a, na sledeći način. Polazeći od $l^* = 1$, uzimamo stanicu $S_{1,6}$. Sada očitavamo $l_1[6]$ čija je vrednost 2 i uzimamo stanicu $S_{2,5}$. Dalje, uzimamo $l_2[5] = 2$ (koristimo stanicu $S_{2,4}$), $l_2[4] = 1$ (stanica $S_{1,3}$), $l_1[3] = 2$ (stanica $S_{2,2}$) i na kraju $l_2[2] = 1$ (stanica $S_{1,1}$).

Korak 3: Računanje najkraćeg vremena

Najkraće vreme prolaska karoserije kroz celu fabriku bi se sada moglo lako izračunati primenom prethodno opisanih rekurzivnih funkcija. Međutim, problem sa ovakvim rekurzivnim algoritmima je u tome što njihovo vreme izvršavanja eksponencijalno zavisi od n.

Mnogo bolji način je ukoliko računamo $f_i[j]$ u suprotnom pravcu od rekurzivnog načina. Primetimo da za svako $j \geq 2$, vrednost $f_i[j]$ zavisi samo od $f_1[j-1]$ i $f_2[j-1]$. Računanjem vrednosti $f_i[j]$ u pravcu povećanja rednog broja stanica j (sa leva na desno na Slici ###b, možemo izračunati najbrži put kroz fabriku i vreme koje on oduzima sa kompleksnošću $O(n)$.

U nastavku je dat pseudokod algoritma za pronalaženje najbržeg puta:

```
FastestWay(a, t, e, x, n)
    f1[1] = e1 + a1,1
    f2[1] = e2 + a2,1
   for j = 2, n
      if  f1[j−1] + a1,j ≤ f2[j−1] + t2,j−1 + a1,j
          f1[j] = f1[j−1] + a1,j
          l1[j] = 1
      else
          f1[j] = f2[j−1] + t2,j−1 + a1,j
          l1[j] = 2
      if  f2[j−1] + a2,j ≤ f1[j−1] + t1,j−1 + a2,j
          f2[j] = f2[j−1] + a2,j
          l2[j] = 2
      else
          f2[j] = f1[j−1] + t1,j−1 + a2,j
          l2[j] = 1
   if  f1[n] + x1 ≤ f2[n] + x2
       f* = f1[n] + x1
       l* = 1
   else
       f* = f2[n] + x2
       l* = 2
```

Korak 4: Konstruisanje najkraćeg puta

Kada imamo izračunate vrednosti $f_i[j]$, f^*, $l_i[j]$ i l^* možemo je da konstruišemo niz stanica kroz koje prolazi karoserija na putu kroz fabriku. Sledeća procedura štampa stanice u redosledu suprotnom od rednih brojeva stanica.

```
PrintStations(l, n)
   i = l*
   print "line", i, "station", n
   for j = n, 2, -1
      i = li[j]
      print "line", i, "station", j-1
```

Lančano množenje matrica

Sledeći primer dinamičkog programiranja je algoritam koji rešava problem lančanog množenja matrica. Neka je dat niz (lanac) od n matrica $(A_1, A_2, ..., A_n)$ čiji proizvod

$$A_1 A_2 \cdots A_n$$

treba izračunati. Kada zagradama grupišemo matrice u parove, prethodni proizvod možemo izračunati višestrukim pozivanjem standardnog potprograma za množenje dve matrice. Proizvod matrica je **u potpunosti grupisan zagradama** ukoliko je on samo jedna matrica ili ako je proizvod dva potpuno grupisana proizvoda. Množenje matrica je u potpunosti asocijativna operacija, tako da svako grupisanje

zagradama daje isti proizvod. Na primer, ukoliko je lanac matrica $(A_1, A_2, ..., A_n)$, proizvod $A_1 A_2 \cdots A_n$ može biti u potpunosti grupisan matricama na pet različitih načina:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

Način na koji se lanac matrica može grupisati ima ogroman uticaj na vreme izračunavanja proizvoda. Posmatrajmo prvo množenje dve matrice. Standardan algoritam je dat sledećim pseudokodom, pri čemu su atributi *kolone* i *redovi* brojevi kolona i redova matrice.

```
MatrixMultiply(A,B)
   if kolone(A) = redovi(B)
      for i = 1, redovi(A)
         for j = 1, kolone(B)
            C[i,j] = 0
            for k = 1, kolone(A)
               C[i,j] = C[i,j]+A[i,k]*B[k,j]
   else
      print "Neodgovarajuce dimenzije"
   return(C)
```

Dve matrice A i B možemo pomnožiti samo ako su one **kompatibilne**, što znači da broj kolona matrice A mora biti jednak broju redova matrice B. Tako, ako je A matrica dimenzija $p \times q$, a B matrica dimenzija $q \times r$, onda je rezultujuća matrica C dimenzija $p \times r$. Vreme potrebno za izračunavanje matrice C je najviše određeno skalarnim proizvodom u pretposlednjoj liniji pseudokoda, koje se ponavlja $p \cdot q \cdot r$ puta. Iz tog razloga ćemo vreme izvršavanja izražavati u funkciji broja skalarnih množenja.

Da bismo pokazali kako vreme izvršavanja zavisi od izbora načina na koji su matrice grupisane zagradama, posmatrajmo problem množenja lanca od tri matrice (A_1, A_2, A_3). Pretpostavimo da su dimenzije matrica 10×100, 100×5 i 5×50, redom. Ukoliko matrice množimo u skladu sa grupisanjem $((A_1A_2)A_3)$, moraćemo da izvršimo $10 \cdot 100 \cdot 5 = 5000$ skalarnih množenja da bismo izračunali proizvod A_1A_2, a zatim još $10 \cdot 5 \cdot 50 = 2500$ skalarnih množenja da bismo dobijenu matricu pomnožili matricom A_3, tako da konačno imamo 7500 skalarnih množenja. Ako bismo matrice množili u skladu sa grupisanjem $(A_1(A_2A_3))$, onda moramo da obavimo $100 \cdot 5 \cdot 50 = 25000$ skalarnih množenja za izračunavanje proizvoda A_2A_3, a zatim još $10 \cdot 100 \cdot 50 = 50000$ skalarnih množenja, da bismo prethodnu matricu pomnožili sa A_1, što je u zbiru 75000 skalarnih množenja. Odavde vidimo da je množenje matrica na prvi način 10 puta brže.

Problem množenja lanca matrica se može definisati na sledeći način: Dat je lanac od n matrica $(A_1, A_2, ..., A_n)$, gde je $i = 1, 2, ..., n$, a matrica A_i ima dimenzije $p_{i-1} \times p_i$. U potpunosti grupisati zagradama proizvod $A_1A_2 \cdots A_n$, tako da broj skalarnih množenja bude minimalan.

Primetimo da u problemu lančanog množenja matrica mi zapravo ne množimo matrice. Naš cilj je da odredimo redosled množenja matrica tako da vreme izvršavanja bude minimalno. Vreme potrebno za određivanje optimalnog redosleda množenja je najčešće znatno manje od vremena koje će se uštedeti prilikom množenja matrica.

Naravno, isprobavanje svih mogućih načina grupisanja nije efikasan algoritam jer bi njegova kompleksnost eksponencijalno zavisila od n. Iz tog razloga ćemo ovaj problem rešiti korišćenjem dinamičkog programiranja.

Korak 1: Struktura optimalnog grupisanja zagradama

Prvi korak u paradigmi dinamičkog programiranja je određivanje optimalne podstrukure i njeno korišćenje u cilju konstruisanja optimalnog rešenjenja na osnovu optimalnih rešenja potproblema. Za lančano množenje matrica, ovaj korak možemo obaviti na sledeći način. Zbog pogodnosti, matricu koja je rezultat proizvoda $A_i \cdot A_{i+1} \cdots A_j$ predstavljaćemo kao $A_{i..j}$, pri čemu je $i \le j$. Primetimo da ako je problem netrivijalan, t.j. ako je $i < j$, onda svako grupisanje proizvoda $A_i \cdot A_{i+1} \cdots A_j$ mora deliti proizvod između A_k i A_{k+1} za neko celobrojno k u opsegu $i \le k \le j$. To znači da za neku vrednost k, prvo računamo matrice $A_{i..k}$ i $A_{k+1..j}$, a zatim ih međusobno množimo da bismo dobili konačni proizvod $A_{i..j}$. Cena množenja pri ovakvom grupisanju je odatle zbir cene računanja matrice $A_{i..k}$, cene računanja matrice $A_{k+1..j}$ i cene množenja ove dve matrice.

Optimalna podstruktura ovog problema je data u nastavku. Pretpostavimo da optimalno grupisanje proizvoda $A_i \cdot A_{i+1} \cdots A_j$ deli proizvod između A_k i A_{k+1}. Onda grupisanje levog podlanca $A_i \cdot A_{i+1} \cdots A_k$ unutar ovog optimalnog grupisanja lanca $A_i \cdot A_{i+1} \cdots A_j$ mora biti optimalno grupisanje proizvoda $A_i \cdot A_{i+1} \cdots A_k$. Zašto? Ukoliko bi postojalo jeftinije grupisanje proizvoda $A_i \cdot A_{i+1} \cdots A_k$, zamenom tog grupisanja u optimalnom grupisanju proizvoda $A_i \cdot A_{i+1} \cdots A_j$ bi dalo jeftinije grupisanje tog proizvoda, što je kontradiktorno. Slično razmišljanje važi i za desni podlanac $A_{k+1} \cdot A_{k+2} \cdots A_j$, koji unutar optimalnog grupisanja lanca $A_i \cdot A_{i+1} \cdots A_j$ mora takođe imati optimalno grupisanje.

Sada možemo iskoristiti optimalnu podstrukturu da bismo konstruisali optimalno rešenje na osnovu optimalnih rešenja potproblema. Videli smo da rešenje svakog netrivijalnog problema lančanog množenja matrica zahteva deljenje proizvoda, tako da svako optimalno rešenje u sebi sadrži optimalna rešenja potproblema. Iz toga sledi da optimalno rešenje problema možemo konstruisati tako što problem podelimo na dva potproblema (optimalno grupisanje podlanaca $A_i \cdot A_{i+1} \cdots A_k$ i $A_{k+1} \cdot A_{k+2} \cdots A_j$), pronađemo optimalna rešenja tih potproblema, a zatim kombinujemo ta dva rešenja. Prilikom traženja odgovarajućeg mesta za deljenje proizvoda, moramo proveriti sva moguća mesta, kako bismo bili sigurni da smo odredili optimalno rešenje.

Korak 2: Rekurzivno rešenje

Definišimo sada cenu optimalnog rešenja rekurzivno u funkciji optimalnih rešenja potproblema. U slučaju problema lančanog množenja matrica, za potprobleme biramo probleme određivanja minimalne cene grupisanja proizvoda $A_i \cdot A_{i+1} \cdots A_j$ za $1 \le i \le j \le n$. Neka je $m[i, j]$ minimalan broj skalarnih množenja potrebnih da bi se izračunala matrica $A_{i..j}$. Za ceo problem, cena najjeftinijeg načina za izračunavanje matrice $A_{1..n}$ bi onda bila $m[1, n]$.

Vrednost $m[i, j]$ možemo rekurzivno definisati na sledeći način. Ako je $i = j$, problem je trivijalan, jer se lanac sastoji samo od jedne matrice $A_{i..i} = A_i$, tako da za računanje proizvoda nije potrebna nijedno skalarno množenje. To znači da je $m[i, i] = 0$ za svako $i = 1, 2, ..., n$. Za računanje $m[i, j]$ kada je $i < j$, koristimo se strukturom optimalnog rešenja iz Koraka 1. Pretpostavimo da optimalno grupisanje deli proizvod $A_i \cdot A_{i+1} \cdots A_j$ između A_k i A_{k+1}, pri čemu je $i \le k < j$. Iz toga sledi da je $m[i, j]$ jednaka minimalnoj ceni izračunavanja potproizvoda $A_{i..k}$ i $A_{k+1..j}$ uvećanoj za cenu međusobnog množenja ove dve matrice. Ako se podsetimo da svaka matrica A_i ima dimenzije $p_{i-1} \times p_i$, dobijamo da je za računanje proizvoda $A_{i..k} A_{k+1..j}$ potrebno izvršiti $p_{i-1} p_k p_j$ skalarnih množenja. Odatle dobijamo

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

Rekurzivno rešenje podrazumeva da znamo vrednost k, koje mi zapravo ne znamo. Međutim, postoji samo $j-i$ mogućih vrednosti za k ($k = i, i+1, ..., j-1$). S obzirom da optimalno grupisanje mora da koristi samo jednu vrednost za k, moramo da proverimo sve moguće vrednosti da bismo utvrdili koje od njih je najbolje. Tako naša rekurzivna definicija minimalne cene grupisanja proizvoda $A_i \cdot A_{i+1} \cdots A_j$ postaje

$$m[i,j] = \begin{cases} 0 & \text{ako je } i = j \\ \min\left(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\right) & \text{ako je } i < j \end{cases}$$

Vrednost $m[i,j]$ daje cenu optimalnog rešenja potproblema. Da bismo mogli kasnije da konstruišemo optimalno rešenje, definišimo $s[i,j]$ koje će da čuva vrednost k za deljenje proizvoda $A_i \cdot A_{i+1} \cdots A_j$ sa kojim se dobija optimalno grupisanje. To zapravo znači da je $s[i,j]$ jednako vrednosti k takvoj da je $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$.

Korak 3: Računanje optimalne cene

Sada je jednostavno napisati rekurzivni algoritam baziran na prethodnom razmatranju, kako bi se izračunala minimalna cena $m[1,n]$ za množenje $A_1 \cdot A_2 \cdots A_n$. Međutim, takvo rekurzivno rešenje bi imalo eksponencijalnu kompleksnost, što je neprihvatljivo.

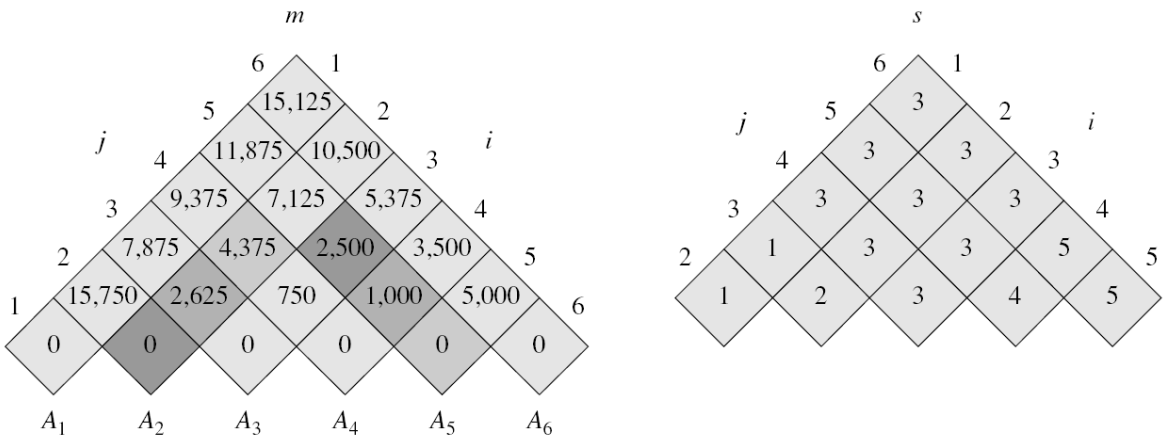
Umesto toga, Korak 3 ćemo rešiti korišćenjem paradigme dinamičkog programiranja, tako što ćemo optimalnu cenu izračunati korišćenjem tabelarnog pristupa "odozdo na gore". Sledeći pseudokod pretpostavlja da matrica A_i ima dimenzije $p_{i-1} \times p_i$ za $i = 1, 2, ..., n$. Ulaz u algoritam je niz $p = (p_0, p_1, ..., p_n)$, gde je $length(p) = n+1$. Procedura koristi pomoćnu tabelu $m[1..n, 1..n]$ za čuvanje cena $m[i,j]$ i pomoćnu tabelu $s[1..n, 1..n]$ koja pamti vrednost k za koje je postignuto optimalno $m[i,j]$. Tabelu s ćemo kasnije iskoristiti za konstruisanje optimalnog rešenja.

Da bismo korektno implementirali pristup "odozdo na gore", moramo odrediti koja polja u tabeli se koriste za računanje $m[i,j]$. Iz prethodnih jednačina se moglo videti da cena $m[i,j]$ računanja proizvoda lanca od $j-i+1$ matrica zavisi samo od računanja prozvoda lanca koji sadrži manje od $j-i+1$ matrica. To znači da je za $k = i, i+1, ..., j-1$, matrica $A_{i..k}$ proizvod $k-i+1 < j-i+1$ matrica, a matrica $A_{k+1..j}$ je proizvod $j-k < j-i+1$ matrica. Iz tog razloga, algoritam će popunjavati tabelu m na način koji odgovara rešavanju problema množenja matrica u lancu koji se povećava.

```
MatrixChainOrder(p)
   n = length(p)-1
   for i = 1, n
      m[i,i] = 0
   for l = 2, n    // l je duzina lanca
      for i = 1, n-l+1
         j = i+l-1
         m[i,j] = ∞
         for k = i, j-1
            q = m[i,k]+m[k+1,j]+p_{i-1}p_kp_j
            if q < m[i,j]
               q = m[i,j]
               s[i,j] = k
   return(m, s)
```

Algoritam prvo postavlja $m[i,i]=0$ za $i=1,2,...,n$ (minimalna cena za lance dužine 1). Zatim se tokom prvog izvršavanja petlje po l izračunava $m[i,i+1]$ za $i=1,2,...,n-1$ (minimalna cena za lance dužine $l=2$). Prilikom drugog prolaska kroz ovu petlju, računa se $m[i,i+2]$ za $i=1,2,...,n-2$ (minimalna cena za lance dužine $l=3$), i tako dalje. U svakom koraku, cena $m[i,j]$ zavisi samo od polja $m[i,k]$ i $m[k+1,j]$ koja su već izračunata.

Na Slici ### je prikazana ova procedura nad lancem od $n=6$ matrica. S obzirom da je $m[i,j]$ definisano samo za $i \le j$, koristi se samo deo matrice m iznad glavne dijagonale. Slika prikazuje tabelu zarotiranu tako da glavna dijagonala leži horizontalno. Lanac matrica je ispisan ispod tabele. Korišćenjem ovakvog prikaza, minimalna cena $m[i,j]$ za množenje podlanca matrica $A_i \cdot A_{i+1} \cdots A_j$ može se pronaći na preseku linija koje prolaze severoistočno od A_i i severozapadno od A_j. Svaki horizontalni red u tabeli sadrži polja za lance matrica iste dužine. Procedura *MatrixChainOrder* izračunava redove od dna ka vrhu i sa leva na desno u okviru svakog reda. Određeno polje $m[i,j]$ se izračunava korišćenjem proizvoda $p_{i-1}p_k p_j$ za $k=1,2,...,j-1$ i svih polja jugozapadno i jugoistočno od $m[i,j]$.



Slika ###. Tabele *m* i *s* dobijene u proceduri *MatrixChainOrder* za lanac od *n*=6 matrica, pri čemu su dimenzije matrica
A1 30 × 35
A2 35 × 15
A3 15 × 5
A4 5 × 10
A5 10 × 20
A6 20 × 25

Jednostavnom analizom strukture ugneždenih petlji unutar procedure *MatrixChainOrder* možemo zaključiti da je vreme izvršavanja algoritma $O(n^3)$, tako da je ovaj algoritam znatno efikasniji od eksponencijalnog algoritma koji bi ispitao sve moguće varijante grupisanja.

Korak 4: Konstruisanje optimalnog rešenja

Iako procedura *MatrixChainOrder* određuje optimalan broj skalarnih množenja potrebnih za izračunavanje proizvoda lanca matrica, ona ne pokazuje direktno kako je potrebno matrice množiti. Zahvaljujući informacijama u tabeli s, nije teško konstruisati optimalno rešenje. Svako polje $s[i,j]$ čuva vrednost k za koju se dobija optimalno deljenje proizvoda $A_i \cdot A_{i+1} \cdots A_j$ između A_k i A_{k+1}. Odatle znamo da se finalno računanje proizvoda $A_{1..n}$ optimalno može izvesti kao $A_{1..s[1,n]} \cdot A_{s[1,n]+1..n}$. Prethodna množenja matrica se mogu izvršiti rekurzivno, s obzirom da $s[1,s[1,n]]$ određuje poslednje množenje matrica pri računanju

$A_{1..s[1,n]}$, a $s\left[s\left[1,n\right]+1,n\right]$ određuje poslednje množenje matrica pri računanju $A_{s[1,n]+1..n}$. Naredna rekurzivna procedura štampa optimalno grupisanje lanca ($A_i, A_{i+1},..., A_j$), pri čemu joj se prosleđuje tabela s izračunata u proceduri *MatrixChainOrder*, kao i indeksi i i j. Početni poziv procedure *PrintOptimalParens*(*s*,1,*n*) štampa optimalno grupisanje lanca matrica $A_1, A_2,..., A_n$.

```
PrintOptimalParens(s,i,j)
   if i = j
      print "A",i
   else
      print "("
      PrintOptimalParens(s,i,s[i,j])
      PrintOptimalParens(s,s[i,j]+1,j)
      print ")"
```

Za primer na Slici ###, poziv *PrintOptimalParens*(*s*,1,6) štampa ((A1(A2A3))((A4A5)A6))