ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Εισαγωγή

Η μέθοδος του Δυναμικού Προγραμματισμού, όπως και η μέθοδος του "Διαίρει-και-Βασίλευε", επιλύει προβλήματα συνδυάζοντας λύσεις σε υποπροβλήματα.

Η σημαντική διαφορά μεταξύ των δύο μεθόδων είναι ότι η μέθοδος "διαίερει-και-βασίλευε"
(α) διαχωρίζει ένα πρόβλημα σε ανεξάρτητα υποπροβλήματα,
(β) λύνει αυτά τα προβλήματα αναδρομικά, και
(γ) συνδυάζει τις λύσεις για να λύσει το αρχικό πρόβλημα.

Σε αντίθεση, η μέθοδος του δυναμικού προγραμματισμού είναι εφαρμόσιμη εκεί όπου τα υποπροβλήματα δεν είναι ανεξάρτητα, αλλά "επικαλύπτονται" και έχουν αρκετά υποπροβλήματα.

Σε αυτές τις περιπτώσεις, η μέθοδος "διαίρει-και-βασίλευε" θα έκανε πολύ περισσότερη δουλειά από όσο χρειαζόταν, αφού θα έλυνε τα κοινά αυτά υποπροβλήματα ξανά και ξανά.

Ένας αλγόριθμος δυναμικού προγραμματισμού λύνει κάθε υποπρόβλημα ακριβώς μία φορά, καταχωρεί τη "λύση" σε ένα πίνακα, και αποφεύγει τον υπολογισμό της "λύσης" κάθε φορά που το υποπρόβλημα συναντάται. !!!

Φάσεις στη Σχεδίαση ενός Αλγορίθμου Δυναμικού Προγραμματισμού

1. Χαρακτήρισε τη δομή μιάς βέλτιστης λύσης.
2. Ορισε αναδρομικά την τιμή μιάς βέλτιστης λύσης.
3. Υπολόγισε την βέλτιστη λύση και την τιμή της "από κάτω προς τα πάνω".

ΔΠ εφαρμογή σε Optimization problems
• Σε τέτοια προβλήματα έχουμε : πολλές δυνατές λύσεις
• κάθε λύση έχει μια τιμή, και θέλουμε αυτή που είναι βέλτιστη (min, max).

Πολλαπλασιασμός Ακολουθίας από Πίνακες

Το πρόβλημα του πολλαπλασιασμού μιάς ακολουθίας (αλυσίδας) από πίνακες έχει ως εξής:

ΕΙΣΟΔΟΣ : Μιά ακολουθία $A_1, A_2, ..., A_n$ από ("συμβατούς") πίνακες
ΕΞΟΔΟΣ : Το γινόμενο $A_1, A_2, ..., A_n$ κάνοντας τον ελάχιστο αριθμό βαθμωτών πολλαπλασιασμών.

Υπάρχουν πολλοί τρόποι υπολογισμού του γινομένου $A_1, A_2, ..., A_n$, οι οποίοι οδηγούν μεν στο ίδιο αποτέλεσμα λόγω του ότι ο πολλαπλασιασμός πινάκων είναι πράξη επιμεριστική.

Όμως έχουν διαφορετικά κόστη (παίρνουμε σαν κόστος το πλήθος των βαθμωτών πολλαπλασιασμών).

Γενικά το κόστος πολλαπλασιασμού δύο πινάκων A, B διαστάσεων $p \times q$ και $q \times r$ αντίστοιχα, είναι p q r, όπως φαίνεται από τον παρακάτω ψευδοκώδικα, όπου C = A B:

```
for i ← 1 to p do
    for j ← 1 to r do
        C[i, j] ← 0
        for k ← 1 to q do C[i, j] ← C[i, j] + A [i, k] . B [k, j]
```

Παράδειγμα: Έστω η αλυσίδα A_1, A_2, A_3 με διαστάσεις $\underbrace{10x100}_{A_1}$ $\underbrace{100x5}_{A_2}$, και $\underbrace{5x50}_{A_3}$, αντίστοιχα.

"παρενθεσιοποίηση"	"κόστος"
$((A_1 . A_2) . A_3)$	10x100x5 + 10x5x50 = 7,500
$(A_1 . (A_2 . A_3))$	10x100x50 + 100x5x50 = 75,000

Όσο το μήκος μιάς αλυσίδας μεγαλώνει, τόσο μεγαλώνει και ο αριθμός των δυνατών παρενθεσιοποίησεων. Π.χ., για την αλυσίδα A1, A2, A3, A4 υπάρχουν 5 δυνατές παρενθεσιοποίησσεις:

(A1. (A2 . (A3 . A4)))
(A1. ((A2 . A3) A4))
((A1. A2) . (A3 . A4))
((A1. (A2 . A3) . A4))
(((A1. A2) . A3) . A4)

$(A_1 | A_2) | (- - | A_n)$

Η εξέταση όλων των δυνατών παρενθεσιοποιήσεων δεν οδηγεί σε γρήγορο αλγόριθμο:

Έστω P(n) ο αριθμός των παρενθεσιοποιήσεων μιάς αλυσίδας μήκους n.

Αφού η αλυσίδα μπορεί να "σπάσει" μεταξύ της k και της k+1 θέσης για οποιοδήποτε k μεταξύ 1 και n-1, και οι προκύπτουσες "μικρές" αλυσίδες να παρενθεσιοποιηθούν αναδρομικά και ανεξάρτητα, θα έχουμε:

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

$n=3 \quad A_1 A_2 A_3$

$P(3) = \underset{1}{P(1)} \cdot \underset{1}{P(2)} + \underset{1}{P(2)}\,\underset{1}{P(1)}$
$= 2$

$\Rightarrow \begin{Bmatrix} A_1 \cdot (A_2 \cdot A_3) \\ (A_1 \cdot A_2) \cdot A_3 \end{Bmatrix}$

Μπορεί να δειχθεί ότι η λύση της παραπάνω αναδρομικής εξίσωσης είναι:

$$P(n) = \frac{1}{n} \binom{2(n-1)}{n-1} \in \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Γενικό πρόβλημα: $\underbrace{P_0 \times P_1}_{A_1} \cdot \underbrace{P_1 \times P_2}_{A_2} \cdots \underbrace{P_{n-1} \times P_n}_{A_n}$

Αλυσίδα $\langle A_1, A_2, .., A_n \rangle$, όπου A_i $(1 \leq i \leq n)$

έχουμε $P_{i-1} \times P_i$
η διάσταση του A_i

Ιδιότητα βέλτιστης παρενθεσιοποίησης

Έστω ότι η βέλτιστη παρενθεσιοποίηση "σπάζει" την αλυσίδα μεταξύ των θέσεων k και k+1.

Τότε οι παρενθεσιοποιήσεις των υποαλυσίδων $A_1, A_2, \ldots, A_k$ και $A_{k+1}, A_{k+2} \ldots, A_n$, είναι και αυτές βέλτιστες.

(Αλλοιώς, μιά καλύτερη παρενθεσιοποίηση της υποαλυσίδας θα οδηγούσε σε παρενθεσιοποίηση της αλυσίδας καλύτερης από τη βέλτιστη).

(1)

(Χαρακτηρισμός δομής βέλτιστης λύσης)

αναδρομικός ορισμός

Συμβολισμός: $A_{i..j} = A_i \,.\, A_{i+1} \,.\, \ldots \,.\, A_j$, όπου $1 \le i \le j \le n$.

(2) Προχωράμε στη 2η φάση της σχεδίασης ενός αλγορίθμου δυναμικού προγραμματισμού που είναι ο ορισμός της τιμής μιάς βέλτιστης λύσης συναρτήσει των τιμών των (βελτίστων) λύσεων σε υποπροβλήματα.

$(A_i \; A_{i+1})(A_{i+2} \; \text{---} \; A_{j-1} \; A_j)$

Έστω **m[i, j]** το ελάχιστο πλήθος βαθμωτών πολλαπλασιασμών για τον υπολογισμό του πίνακα $A_{i..j}$.

Τότε, m[i, j] είναι ίσο με το ελάχιστο κόστος υπολογισμού των υποπροβλημάτων υπολογισμού των $A_{i..k}$ και $A_{k+1..j}$, συν το κόστος πολλαπλασιασμού των δύο αυτών πινάκων. !

Ο υπολογισμός του γινομένου $A_{i..k} \times A_{k+1..j}$, απαιτεί $p_{i-1}\, p_k\, p_j$ βαθμωτούς πολλαπλασιασμούς, και επομένως έχουμε

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

[όπου ο πίνακας A_i έχει διαστάσεις $p_{i-1} \times p_i$]

$$\left.\begin{array}{l} A_i \Rightarrow P_{i-1} \cdot P_i \\ A_k \Rightarrow P_{k-1} \cdot P_k \\ A_j \Rightarrow P_{j-1} \cdot P_j \end{array}\right\} P_{i-1} \cdot P_k \cdot P_j \; ?$$

Επομένως, ο αναδρομικός ορισμός για το ελάχιστο κόστος παρενθεσιοποίησης του γινομένου $A_i\ A_{i+1} \ldots A_j$, γίνετε:

$$m[i, j] = \begin{cases} 0, & \text{άν } i = j \\ \min\limits_{i \le k \le j} \{m[i, k] + m[k+1, j] + p_{i-1}\, p_k\, p_j\}. & \text{αν } i < j \end{cases}$$

Σε αυτό το σημείο φαίνεται προφανής ο παρακάω αναδρομικός αλγόριθμος που υπολογίζει το m[1, n] αλλά και το βέλτιστο "σπάσιμο" (παρενθεσιοποίηση):

Αλγόριθμ

> άν n = 1 τερμάτισε και επίστρεψε 0,
> αλλοιώς, για κάθε k μεταξύ 1 και n, υπολόγισε το
> $m[1, k] + m[k+1, n] + P_0 P_k P_n$ αναδρομικά και
> διάλεξε k_0 το οποίο το ελαχιστοποιεί
> */αυτό το k_0 υπαγορεύει και την αντίστοιχη παρενθεσιοποίηση/*

Πόσο χρόνο παίρνει ο παραπάνω αλγόριθμος υπολογισμού του m[1, n];

Έστω T (n) ο χρόνος εκτέλεσης του παραπάνω αλγορίθμου υπολογισμού m[1, n] (και της βέλτιστης παρενθεσιοποίησης). Θα έχουμε:

$$T(1) \ge 1$$

$$T(n) \ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1), \quad n > 1$$

Κάθε όρος από τους T (1), T(2), ..., T (n-1) εμφανίζεται δύο φορές (ο όρος T(i) εμφανίζεται όταν k = i και όταν n - k = i). Συνεπώς:

$$T(n) \ge 1 + 2 \sum_{k=1}^{n-1} T(k) + \underbrace{n-1} = n + 2 \sum_{k=1}^{n-1} T(k)$$

$T(1) \geq 1 = 1^1$
$T(2) \geq 2 + 2 \cdot 1 = 4 \geq 2^2$
$T(3) \geq 3 + 2 \cdot (T(1) + T(2)) = 3 + 2 \cdot 5 = 13 \geq 2^3$
⋮
$T(4) \geq 2^4$

Επιλύουμε την παραπάνω αναδρομική ανισότητα με την μέθοδο της αντικατάστασης. Προβλέπουμε ότι $T(n) \in \Omega(2^n)$.

Υποθέτουμε ότι $T(n) \geq 2^{n-1}$ για κάθε $n \geq 1$. Αποδεικνύουμε επαγωγικά την υπόθεσή μας. Για n =1, ισχύει $T(1) = 1 \geq 2^{i-1}$.

Για $n > 1$,

$$
\begin{aligned}
T(n) &\geq n + 2 \sum_{k=1}^{n-1} T(k) && \text{(από αναδρομική ανισότητα)} \\
&\geq n + 2 \sum_{k=1}^{n-1} 2^{k-1} && \text{(από επαγωγική υπόθεση)} \\
&= n + 2 \sum_{k=0}^{n-1} 2^{k} \\
&= n + 2 \frac{2^{n-1} - 1}{2 - 1} \\
&\geq 2^{n-1}
\end{aligned}
$$

όπως χρειάζεται.

Συνεπώς, ο προφανής αναδρομικός αλγόριθμος δεν είναι καλός.

Προχωράμε στην 3η φάση της σχεδίασης ενός αλγόριθμου δυναμικού προγραμματισμού που συνίσταται στον υπολογισμό της λύσης "από κάτω προς τα πάνω".

Πόσα υποπροβλήματα υπάρχουν;

Οσα και τα ζεύγη i και j, $1 \leq i \leq j \leq n$, δηλαδή $\binom{n}{2} + n \in \theta(n^2)$.

Θα υπολογίσουμε το m[1, n] επιλύοντας κάθε υποπρόβλημα μόνο μία φορά, αρχίζοντας από "κάτω" και πηγαίνοντας προς τα "πάνω".

Έχουμε τον εξής αλγόριθμο:

- Υπολόγισε πρώτα τα m[1, 1], m[2, 2], ..., m[n, n].
  */ ίσα με μηδέν */
- Υπολόγισε μετά τα m[1, 2], m[2, 3], ..., m[n-1, n]. χρησιμοποιώντας την αναδρομική έκφραση για τα m.
- Συνέχισε υπολογίζοντας τα m[1, 3], m[2, 4], ..., m[n-2, n], κ.ο.κ.

Με τον παραπάνω αλγόριθμο, το κάθε υποπρόβλημα επιλύεται μόνο μιά φορά και η λύση του καταχωρείται σε ένα πίνακα και χρησιμοποιείται κατ' ευθείαν κάθε φορά που το υποπρόβλημα "καλείται" από ένα μεγαλύτερο υποπρόβλημα.

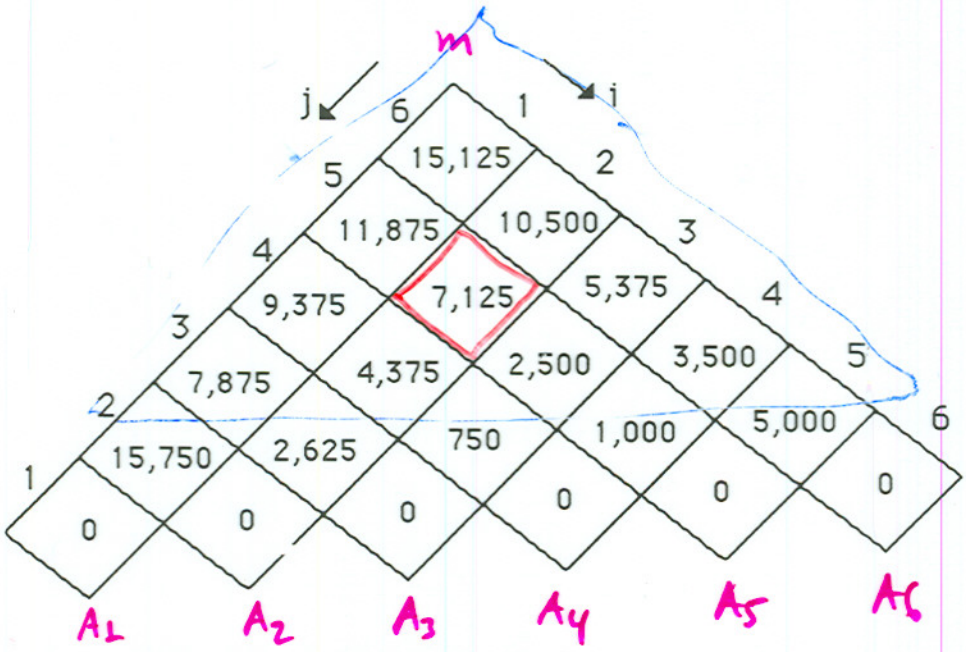
Επίδειξη του αλγορίθμου πάνω σε μιά αλυσίδα από 6 πίνακες:

πίνακας	διάσταση
A_1	30 x 35
A_2	35 x 15
A_3	15 x 5
A_4	5 x 10
A_5	10 x 20
A_6	20 x 25

m[1,1], m[2,2], ..., m[6,6]

m[1,2] m[2,3] ....

m[1,3] = min{m[1,2], m[2,3]}



Να, π.χ. πως υπολογίζεται η θέση m[2, 5]:

$$
m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + P_1P_2P_5 = 0 + 2{,}500 + 35.15.20 = 13{,}000 \\ m[2,3] + m[4,5] + P_1P_3P_5 = \ldots = 7{,}125 = 7{,}125 \\ m[2,4] + m[5,5] + P_1P_4P_5 = \ldots = 11{,}375 \end{cases}
$$

$A_i \rightarrow P_{i-1} P_i$

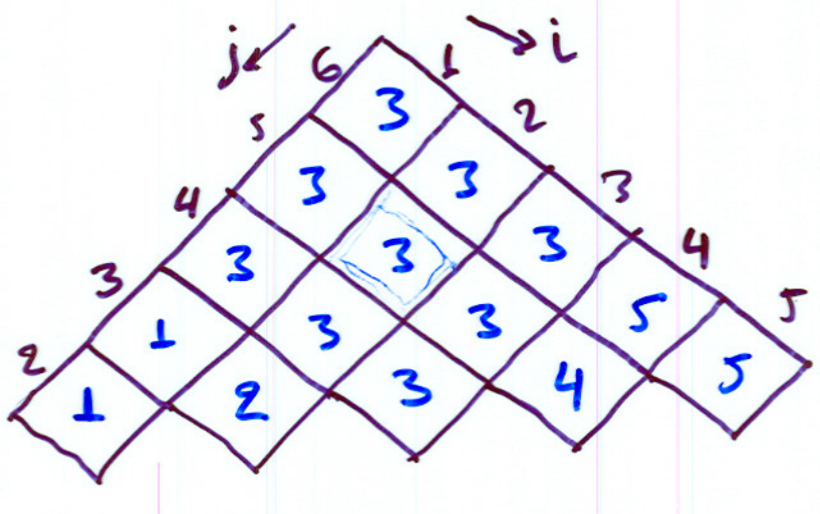
$m[1,k] + m[k+1,n] + P_0 \cdot P_k P_n$

Πόσο χρόνο παίρνει ο παραπάνω αλγόριθμος δυναμικού προγραμματισμού;

Υπάρχουν $\Theta(n^2)$ υποπροβλήματα και το καθένα παίρνει $O(n)$ χρόνο για να βρει το ελάχιστο στοιχείο $O(n)$ στοιχείων (κάθε στοιχείο αντιστοιχεί σε ένα σημείο "σπασίματος" της υποαλυσίδας που αντιστοιχεί στο υποπρόβλημα).

Άρα, συνολικά, $O(n^3)$ χρόνο.

Συνοψίζοντας, η μέθοδος δυναμικού προγραμματισμού είναι εύχρηστη εκεί όπου τα υποπροβλήματα "αλληλοκαλυπτονται" και οι λύσεις των υποπροβλημάτων αυτών είναι και αυτές βέλτιστες.



$A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5 \cdot A_6$

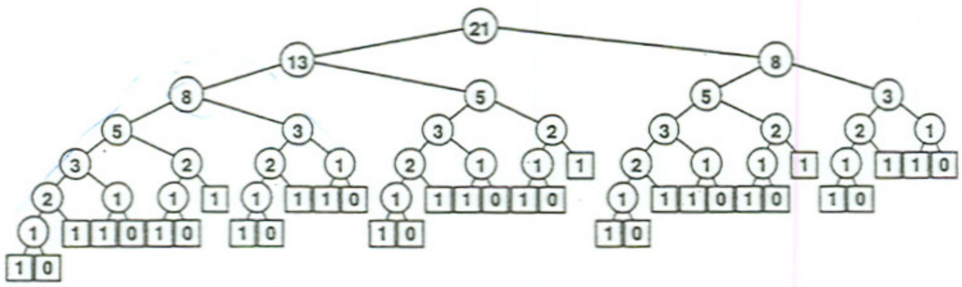
$(A_1 \cdot A_2 \cdot A_3) \cdot (A_4 \cdot A_5 \cdot A_6)$

$((A_1) \cdot (A_2 \cdot A_3)) \cdot ((A_4 \cdot A_5) \cdot A_6)$

Πρόγραμμα 5.10 Αριθμοί Fibonacci (αναδρομική υλοποίηση)

Αυτό το πρόγραμμα, αν και συμπαγές και κομψό, δεν είναι εύχρηστο επειδή απαιτεί εκθετικό χρόνο για τον υπολογισμό του F_N. Ο χρόνος εκτέλεσης για τον υπολογισμό του F_{N+1} είναι $\phi \approx 1{,}6$ φορές ο χρόνος που απαιτείται για τον υπολογισμό του F_N. Για παράδειγμα, αφού $\phi^9 > 60$, αν θεωρήσουμε ότι ο υπολογιστής μας χρειάζεται περίπου ένα δευτερόλεπτο για να υπολογίσει το F_N, γνωρίζουμε ότι θα χρειαστεί περισσότερο από ένα λεπτό για να υπολογίσει τον F_{N+9}, και περισσότερο από μία ώρα για να υπολογίσει τον F_{N+18}.

```
static int F(int i)
  {
    if (i < 1) return 0;
    if (i == 1) return 1;
    return F(i-1) + F(i-2);
  }
```



Εικόνα 5.14
Δομή αναδρομικού αλγορίθμου για τους αριθμούς Fibonacci

Αυτή η εικόνα των αναδρομικών κλήσεων που γίνονται για τον υπολογισμό του F_8 με βάση τον κλασικό αναδρομικό αλγόριθμο διασαφηνίζει τον τρόπο με τον οποίο η αναδρομή με επικαλυπτόμενα υποπροβλήματα μπορεί να οδηγήσει σε εκθετικό κόστος. Σε αυτή την περίπτωση, η δεύτερη αναδρομική κλήση αγνοεί τους υπολογισμούς που έγιναν κατά την πρώτη, κάτι που έχει ως αποτέλεσμα τη μαζική εκτέλεση των ίδιων υπολογισμών επειδή το αποτέλεσμα πολλαπλασιάζεται αναδρομικώς. Δίπλα παρουσιάζονται οι αναδρομικές κλήσεις για τον υπολογισμό του $F_6 = 8$ (που φαίνονται στο δεξιό υποδένδρο της ρίζας και το αριστερό υποδένδρο του αριστερού υποδένδρου της ρίζας).

```
8 F(6)
  5 F(5)
    3 F(4)
      2 F(3)
        1 F(2)
          1 F(1)
          0 F(0)
        1 F(1)
      1 F(2)
        1 F(1)
          0 F(0)
    2 F(3)
      1 F(2)
        1 F(1)
        0 F(0)
      1 F(1)
  3 F(4)
    2 F(3)
      1 F(2)
        1 F(1)
        0 F(0)
      1 F(1)
    1 F(2)
      1 F(1)
        0 F(0)
```

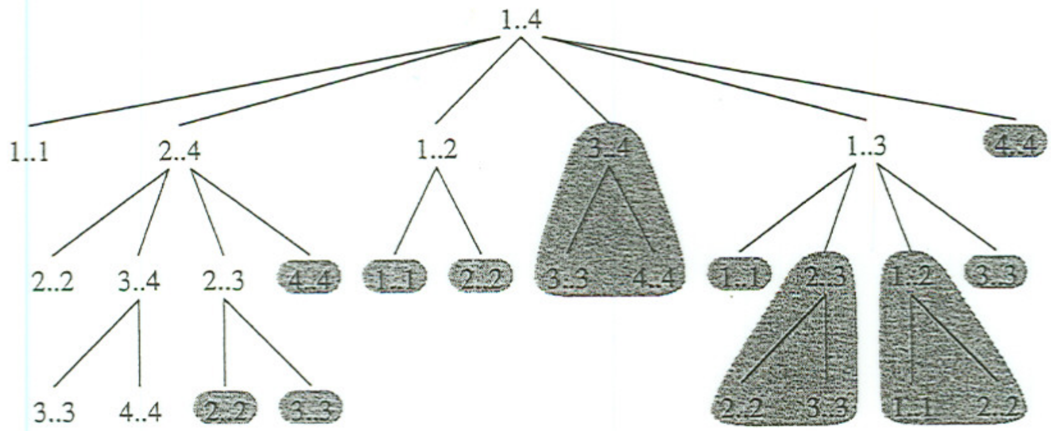


Figure 15.5 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN$(p, 1, 4)$.

observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, entry $m[3, 4]$ is referenced 4 times: during the computations of $m[2, 4]$, $m[1, 4]$, $m[3, 5]$, and $m[3, 6]$. If $m[3, 4]$ were recomputed each time, rather than just being looked up, the increase in running time would be dramatic. To see this, consider the following (inefficient) recursive procedure that determines $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A_{i..j} = A_i A_{i+1} \cdots A_j$. The procedure is based directly on the recurrence (15.12).

RECURSIVE-MATRIX-CHAIN(p, i, j)

if $i = j$
  then return 0
$m[i, j] \leftarrow \infty$
for $k \leftarrow i$ **to** $j - 1$
  do $q \leftarrow$ RECURSIVE-MATRIX-CHAIN(p, i, k)
      $+$ RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$
      $+ p_{i-1} p_k p_j$
    if $q < m[i, j]$
      then $m[i, j] \leftarrow q$
return $m[i, j]$

Figure 15.5 shows the recursion tree produced by the call RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node is labeled by the values of the parameters i and j. Observe that some pairs of values occur many times.

In fact, we can show that the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. Let $T(n)$ denote the time taken by RECURSIVE-