5 4
↑ ↑
i=1 j=2

4 5
j=1 i=2

4 5
k=1
j=1

RS(A, 1, 1, 1)
Ans: A[1] = 4
(larger than 3 other elements)
1, 2, 3

Avg. Case

Randomized-Partition produces a partition where probability of occurence (for each element to be chosen as pivot) = $\frac{1}{n}$ (for n elements)

If we assume that in Randomized-Select, the i^{th} element is ~~always~~ present in larger side of partition, we get:

$$T(n) \le \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n)$$

for partition procedure

$$T(n) \le \frac{1}{n}\left(T(n-1) + \sum_{k=1}^{\lceil n/2 \rceil - 1} T(n-k) + \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

as $\max(1, n-1) = n-1$

and for $\max(k, n-k)$

for $k \ge n-k$

$2k \ge n$

$k \ge \frac{n}{2}$

for $k < n-k$

$2k < n$

$k < \frac{n}{2}$

ie $\max(k, n-k) = \begin{cases} k & \text{if } k \ge \lceil n/2 \rceil \\ n-k & \text{if } k < \lceil n/2 \rceil \end{cases}$

eg. $f(n) = 2n^2 + 3n + 1$

$\leq 2n^2 + 3n^2 + n^2$

$\leq 6n^2$

$$T(k) \leq \frac{1}{n}\left(T(n-1) + \sum_{k=1}^{\lceil n/2 \rceil - 1} T(n-k) + \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n)$$

as if $f(n) = \sum_{k=1}^{\lceil n/2 \rceil - 1} T(n-k) + \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)$

$$\leq 2 \sum_{k=\lceil n/2 \rceil}^{n-1} T(k)$$

$$T(k) \leq \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n) + O(n)$$

$$= \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

We solve by substitution

we assume

Selection in worst-case linear time

To get a Select algorithm which takes $O(n)$ time in worst-case. The idea behind this algo is to guarantee a good split when array is partitioned. Select algo uses the Partition algo from Quicksort.

Select algo has following steps:

(1) Divide the n elements of the input array into $\lceil n/5 \rceil$ groups of 5 elements each and atmost one group made up of remaining $n \bmod 5$ elements.

(2) Find median of each of $\lceil n/5 \rceil$ groups by insertion sorting and taking middle element (If the group has even no. of elements, take larger of 2 medians)

(3) Use Select recursively to find median x of the $\lceil n/5 \rceil$ medians found in step 2.

(4) Partition the input array around the median of medians x using a modified version of PARTITION. Let k be no. of elements on low side of partition, so that $n-k$ be no. of elements on high side.

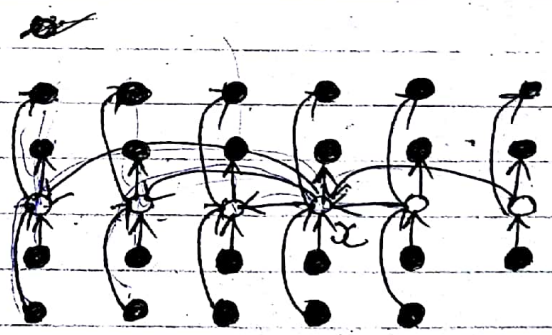
(5) Use Select recursively to find the i^{th} smallest element on low side if $i \le k$ or $(i-k)^{th}$ smallest element on high side if $i > k$.

---

eg. if $n = 29$

$$\frac{n}{5} = \left\lceil \frac{29}{5} \right\rceil = \lceil 5.8 \rceil = 6$$

$29 \bmod 5 = 4$ ← last gp has 4 elements



n elements are represented by small circles & each gp occupies a column. The medians of gps are whitened and median of medians x is labeled.

3 or of every gp. to right of x are $\geq x$
x 3 " " " to left " $\leq x$

Arrows are drawn from larger to smaller elements. Atleast half of $\lceil n/5 \rceil$ gps contribute 3 elements that are greater than x, except for one group that has fewer than 5 elements (if 5 does not divide n exactly) and one group containing x itself.
ie no of elements greater than x is atleast

$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right)$

$\geq \frac{3n}{10} - 6$

$\geq x$

$\frac{7n}{10}+6$ x $\frac{3n}{10}-6$

Total = n

$n - \frac{3n}{10} + 6$

$= \frac{10n - 3n}{10} + 6$

$= \frac{7n}{10} + 6$

So, in worst case, Select is called recursively on atmost $\frac{7n}{10}+6$ elements in step 5.

Step 1, 2, and 4 take $O(n)$ time
Step 3 takes $T(\lceil n/5 \rceil)$ time

Step 5 takes $T\left(\frac{7n}{10}+6\right)$ time

$$T(n) \leq T\left(\left\lceil \frac{n}{5}\right\rceil\right) + T\left(\frac{7n}{10}+6\right) + O(n)$$

We prove it by substitution
Assume that $T(n) \leq cn$

$$T(n) \leq c\left(\left\lceil \frac{n}{5}\right\rceil\right) + c\left(\frac{7n}{10}+6\right) + O(n)$$

$$\leq \frac{cn}{5} + c + \frac{c7n}{10} + 6c + O(n)$$

$$\leq \frac{9cn}{10} + 7c + O(n)$$

$$\leq cn$$

$T(n) = O(n)$

Dynamic Programming

- It is like divide and conquer method, it solves problems by combining solutions to subproblems.

Diff

(1) But divide and conquer partition problem into subproblems, solve the subproblems recursively and then combine their solutions to solve original problem.

- But D.P. is applicable when subproblems are not independent ie when subproblems share subsubproblems.

(2) A divide & conquer algo does more work than necessary repeatedly solving the common subsubproblems.

- But D.P. algo solves every subsubproblem just once & then saves its answer in a table, thereby avoiding work of recomputing the answer every time the subsubproblem is encountered.

3) Divide and conquer is top-down approach.
But D.P is bottom up technique. We start with smallest subproblems and combining their solutions, until finally we arrive at solution of original problem.

- D.P is typically applied to optimization problems where there are many possible solution and each solution has a value and we wish to find an optimal solution (with min or max value).

For D.P technique to be applicable a problem must have:

1) Optimal substructure: optimal solution to the problem contains within it optimal solution to subproblems.

ex. in MCM, the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits product b/w A_k and A_{k+1} contains within it optimal solution to problem of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} \cdots A_j$

(2) Overlapping subproblem:
When a recursive algorithm revisits the same problem over & over again, we say that the problem has overlapping subproblem.

- It
- m
- co
- Dif
- (1) Bu
- p
- th
- H
- o
- F
- A

(1) So, solves each subproblem once & then looks up solution which can be looked up when needed.
eg, of MCM, it looks up the solution to subproblems & saves some time when solving subproblems of higher levels.

4. m[3,4] is referenced when m[2,4], m[1,4] -- are computed

(2)

X

Proof of Master Theorem

The 1st part of proof of master theorem analyzes the master recurrence:-

$T(n) = aT(n/b) + f(n)$

under assignment that n is an exact power of $b > 1$.

Lemma 1

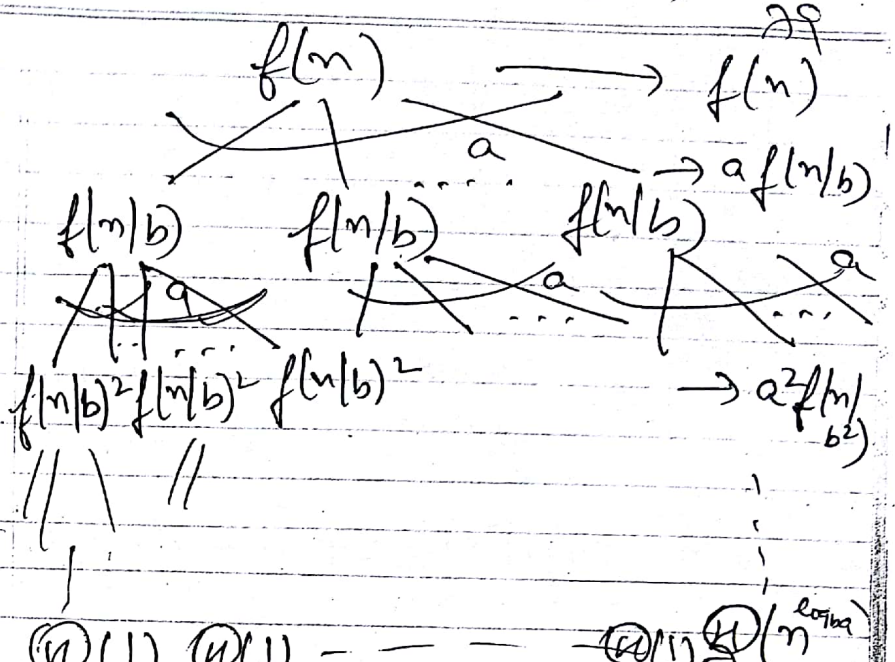
Let $a \geq 1$ and $b > 1$ be constants & let $f(n)$ be a non negative function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

where i is the integer, then:

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

$T(n) = aT(n/b) + f(n)$ X



For no. of levels

$$\Rightarrow T\left(\frac{n}{b^i}\right) = T(1) = 1$$

ie $\frac{n}{b^i} = 1$

$n = b^i$

$i = \log_b n$

$(0, 1 - - - \log_b n)$

$$T(n) = f(n) + aT(n/b)$$

$$= f(n) + af(n/b) + a^2 f(n/b^2) + \cdots$$

$$\Theta(n^{\log_b a}) \quad \| \quad T(1)$$

$$T(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) + \Theta(n^{\log_b a})$$

as $T(1) = \Theta(1)$

Thus proved

Lemma 2

If $a \geq 1$ and $b > 1$ be constants and let $f(n)$ be a non-ve function defined on exact powers of b. A function $g(n)$ defined over exact powers of b by:

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j) \quad \text{---} (2)$$

$$= n^{\log_b a - \epsilon} \sum_{j=0}^{\log_b n - 1} (b^\epsilon)^j$$

$$= n^{\log_b a - \epsilon} \left(\frac{b^{\epsilon \log_b n} - 1}{b^\epsilon - 1} \right)$$

$$= n^{\log_b a - \epsilon} \left(\frac{n^\epsilon - 1}{b^\epsilon - 1} \right)$$

$$= n^{\log_b a - \epsilon} O(n^\epsilon)$$

$$= O(n^{\log_b a}) \quad \text{--- (4)}$$

Sub^n (4) in (3), yields

$$g(n) = O(n^{\log_b a})$$

Case 1 is proved

Case 2

$$f(n) = \Theta(n^{\log_b a})$$

$$f(n/b^j) = \Theta((n/b^j)^{\log_b a})$$

Sub^n in (2), we get

$$\Rightarrow g(n) = \Theta\left(\sum_{j=0}^{\log_b n - 1} a^j \left(\frac{n}{b^j} \right)^{\log_b a} \right)$$

$af(n/b) \le cf(n)$

$c<1$ and all $n \ge b$

we have $a^i f(n/b^i) \le c^i f(n)$ from (5)

$$g(n) \le \sum_{j=0}^{\log_b n - 1} a^i f(n/b^i)$$

$$\le \sum_{j=0}^{\log_b n - 1} c^i f(n)$$

$$\le f(n) \sum_{i=0}^{\infty} c^i$$

$$= f(n)\left(\frac{1}{1-c}\right) \quad \left[\frac{a}{1-b}\right]$$

$g(n) = O(f(n))$ — (6)

we have

$f(n) = \Theta(f(n))$

Lemma 3

Let $a \ge 1$ and $b > 1$ be constants and let $f(n)$ be a non-ve function defined on exact powers of b. Define $T(n)$ on exact powers of b by the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n = b^i \end{cases}$$

where i is a +ve integer. Then $T(n)$ can be bounded asymptotically for exact powers of b as follows:-

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

(2) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$

(3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all large n, then $T(n) = \Theta(f(n))$

Proof: We use bounds in lemma 2 to evaluate summations from lemma 1

Case 1

$T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a})$

$= \Theta(n^{\log_b a})$

Case 2

$T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \lg n)$

$= \Theta(n^{\log_b a} \lg n)$

Case 3

$T(n) = \Theta(n^{\log_b a}) + \Theta(f(n))$

$= \Theta(f(n))$

Matrix-chain multiplication

MCM problem can be stated as:- Given a chain $(A_1, A_2 \ldots A_n)$ of n matrices where $i = 1, 2, 3 \ldots n$ Matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1, A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Total number of parenthesizations $P(n)$ for a sequence of n matrices is given by:-

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

eg. for 4 matrices (we can have 5 ways)

$$P(4) = \sum_{k=1}^{3} P(k) P(4-k)$$

$= P(1) P(3) + P(2) P(2) + P(3) P(1)$

$= 2 + 1 + 2 = 5$

$$P(3) = \sum_{k=1}^{2} P(k) P(3-k)$$

$= P(1) P(2) + P(2) P(1)$

$= 2$

$$P(2) = \sum_{k=1}^{1} P(k)\, P(2-k)$$

$$= P(1)\, P(1)$$

$$= 1$$

Let there be 4 matrices A_1, A_2, A_3, A_4, to compute it, it can be done in 5 ways:-

1. $(A_1(A_2(A_3 A_4)))$
2. $((A_1 A_2)(A_3 A_4))$
3. $(((A_1 A_2) A_3) A_4)$
4. $((A_1 (A_2 A_3)) A_4)$
5. $(A_1 ((A_2 A_3) A_4))$

Note: If $A_1 - x \times y$

$A_2 - y \times z$

$A_1 A_2 - x \times z$

Total no. of scalar multiplications $= xyz$

eg. We have to compute matrix chain product $A_1 A_2 A_3$

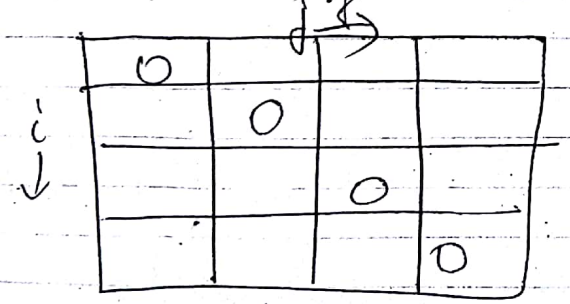
Using Dynamic Programming to solve MCM problem:-

(1) We have to find optimal parenthesization of $A_1, A_2, \ldots A_n$. An optimal parenthesization of $A_1, A_2 \ldots A_n$ splits product between A_k and A_{k+1} for some k between $1 \le k < n$ i.e first of all compute optimal parenthesization of $A_1, A_2 \ldots A_k$, then compute optimal parenthesization of $A_{k+1} \ldots A_n$ and finally multiply $A_{1 \ldots k}$ and $A_{k+1 \ldots n}$ to produce the final product $A_{1 \ldots n}$.

(2) Recursively define value of an optimal solution. Let $m[i,j]$ be the minimum no. of S.M. needed to compute $A_{i \ldots j}$. (eg. $m[2,5]$ is min no. of S.M. needed to compute $A_{2 \ldots 5}$ i.e $A_2 A_3 A_4 A_5$.

If $i = j$, then chain consists of just one matrix $A_{i \ldots i} = A_i$.

$m[i,i] = 0$ for $i = 1, 2, 3 \ldots n$



Matrix m

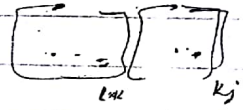
i will be always less than or equal to j.

When $i < j$, let the optimal parenthesization splits product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1} where $i \le k < j$

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

Because k can be $i, i+1 \ldots j-1$

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



$m[i,j]$ gives the minimum no. of scalar multiplications required to find $A_i A_{i+1} \ldots A_j$.

To keep track of how to construct an optimal solution we use array $s[1 \ldots n][1 \ldots n]$

We store value k at $s[i,j]$ at which we can split product $A_i A_{i+1} \ldots A_j$.

We compute optimal solution in bottom up fashion i.e first we compute $m[i,j]$ for $i=j$.
Then we compute $m[i,i+1]$ for $i=1,2,3 \ldots n-1$.

eg. We have 6 matrices $A_1 \ldots A_6$ given as follows:

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

30	35	15	5	10	20	25
P_0	P_1	P_2	P_3	P_4	P_5	P_6

$m[1,2] = m[1,1] + m[2,2] + P_0P_1P_2$

$k=1$

$\Rightarrow 0+0+30\times35\times15$

$\Rightarrow 15750$

$m[2,3] = m[2,2] + m[3,3] + P_1P_2P_3$

$k=2$

$= 0+0+35\times15\times5$

$= 2625$

$m[3,4] = m[3,3] + m[4,4] + P_2P_3P_4$

$k=3$

$= 0+0+15\times5\times10 = 750$

$m[4,5] = m[4,4] + m[5,5] + P_3P_4P_5$

$k=4$

$= 0+0+5\times10\times20$

$= 1000$

$k=5$

$m[5,6] = m[5,5] + m[6,6] + P_4P_5P_6$

$= 0+0+10\times20\times25 = 5000$

$m[1,3] = \min \begin{cases} m[1,1]+m[2,3]+P_0P_1P_3, \\ m[1,2]+m[3,3]+P_0P_2P_3 \end{cases}$

$1 \le k < 3$

$k=1$

$k=2$

$= \min \begin{cases} 0+2625+30\times35\times5 \\ 15750+0+30\times15\times5 \end{cases}$

$= \min \{7875, 18000\}$

$= 7875$

$m[2,4] = \min \begin{cases} m[2,2]+m[3,4]+P_1P_2P_4, \\ m[2,3]+m[4,4]+P_1P_3P_4 \end{cases}$

$\Rightarrow \min \begin{cases} 0+750+35\times15\times10, \\ 2625+0+35\times5\times10 \end{cases}$

$\Rightarrow \min \{6000, 4375\} = 4375$

$m[3,5] = \min \{ m[3,3] + m[4,5] + p_2 p_3 p_5,\ m[3,4] + m[5,5] + p_2 p_4 p_5 \}$

$= \min \{ 0 + 1000 + 15 \times 5 \times 20,\ 750 + 0 + 15 \times 10 \times 20 \}$

$= \min \{ 2500, 3750 \}$

$= 2500$

$m[4,6] = \min \{ m[4,4] + m[5,6] + p_3 p_4 p_6,\ m[4,5] + m[6,6] + p_3 p_5 p_6 \}$

$\Rightarrow \min \{ 0 + 5000 + 5 \times 10 \times 25,\ 1000 + 0 + 5 \times 20 \times 25 \}$

$\Rightarrow \min \{ 6250, 3500 \}$

$\Rightarrow 3500$

$m[1,4] = \min \{ m[1,1] + m[2,4] + p_0 p_1 p_4,\ m[1,2] + m[3,4] + p_0 p_2 p_4,\ m[1,3] + m[4,4] + p_0 p_3 p_4 \}$

$\Rightarrow \min \{ 0 + 4375 + 30 \times 35 \times 10,\ 1575 + 750 + 30 \times 15 \times 10,\ 7875 + 30 \times 5 \times 10 \}$

$\Rightarrow \min \{ 14875, 21000, 9375 \}$ $\quad k = 3$

$\Rightarrow 9375$

$m[2,5] = \min \{ m[2,2] + m[3,5] + p_1 p_2 p_5,\ m[2,3] + m[4,5] + p_1 p_3 p_5,\ m[2,4] + m[5,5] + p_1 p_4 p_5 \}$

$= 7125$

$k = 3$

Minimum no. of S.M. reqd. to compute $A_{1 \ldots 6} = 15125$ & optimal parenthesization is:-

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

$A_{16} \rightarrow k = 3$

$A_{13} \rightarrow k = 1$

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

$A_{46} \rightarrow k = 5$

$[(A_1)(A_2 A_3))((A_4 A_5)(A_6))]$

m[i, j+1]

m[6,6] =

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	(15125)
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

Array m

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	(5)
5						5
6						

Array s ↓ we store values of k

Print_optimal_paren (s, i, j)
if i = j
then print "A"i
else print "("
Print_optimal_paren (s, i, s[i,j])
Print_optimal_paren (s, s[i,j]+1, j)
Print ")"

Eg. (5, 1, 6)

Ans- ((5, 1, 3) + (5, 4, 6)

(5, 1, 3)

Print ((A1 (5, 1, 1) + (5, 2, 3)

(5, 1, 1)

Print ((A1

(5, 2, 3)

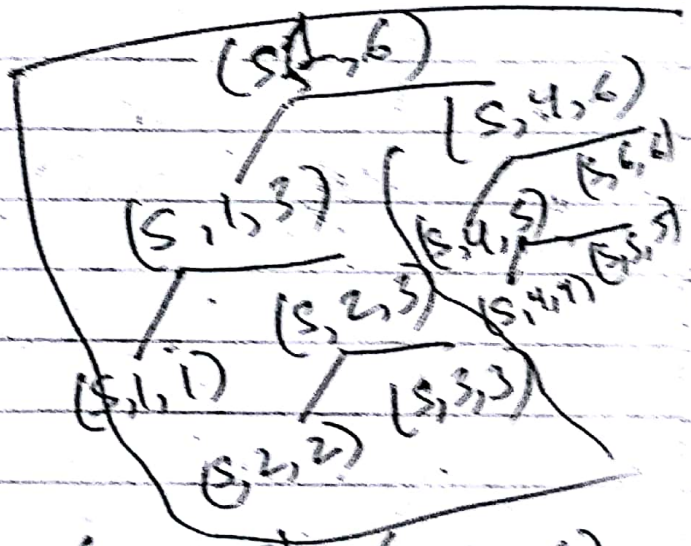
Print ((A1((5, 2, 2) + (5, 3, 3)

(5, 2, 2)

Print ((A1(A2

(5, 3, 3)

Print ((A1(A2A3))



(5, 4, 6).

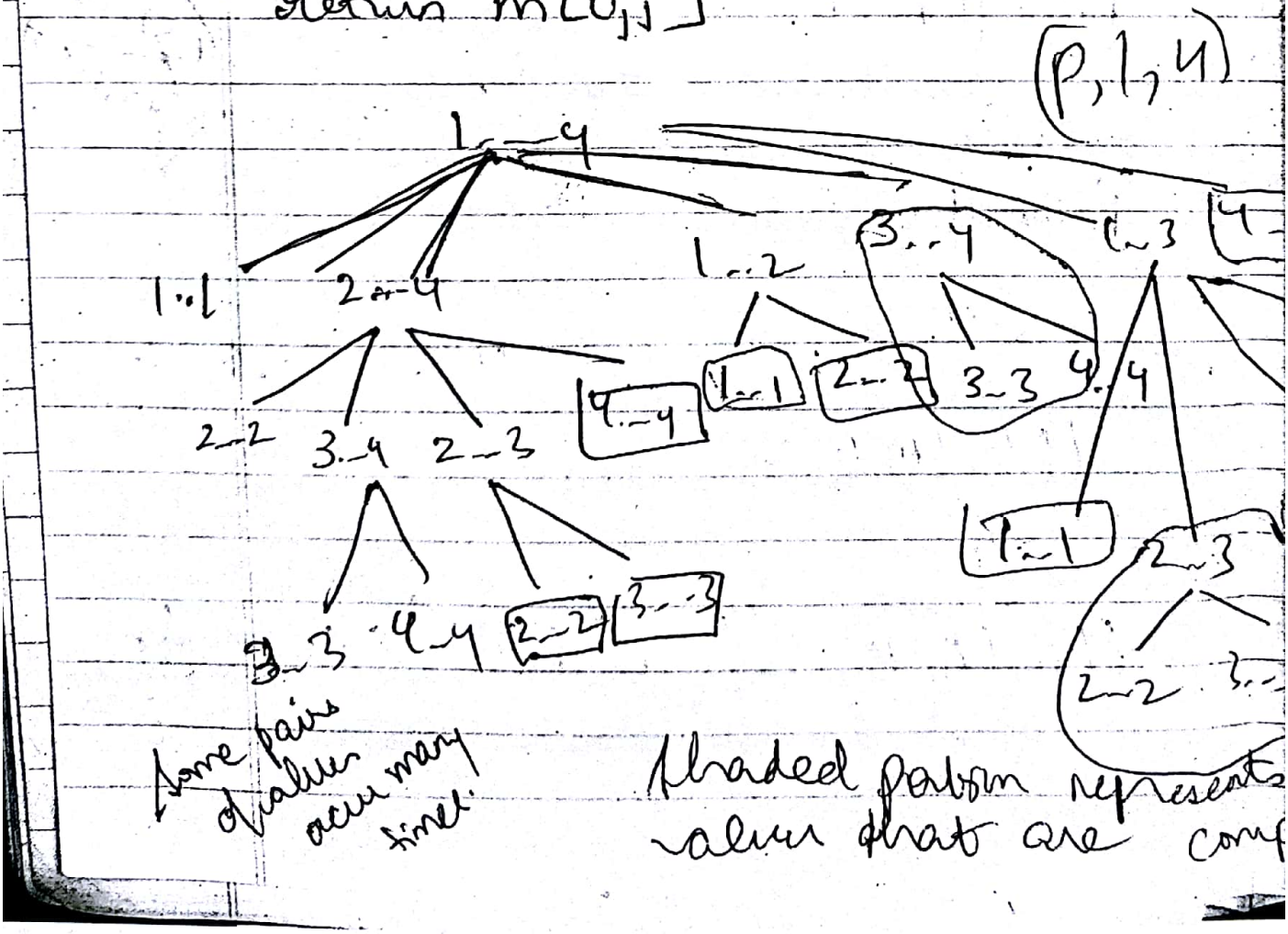
((A1(A2A3)) C

(5, 4, 5) (5, 4, 5) + (5, 6, 6)

((A1(A2A3))((A4A5)A6)) (5, 4, 4) + (5, 5, 6)

If every entry has to be recomputed rather than looked up, it would increase running time dramatically. To see this, consider following recursive procedure:-

```
Recursive - Matrix Chain (p,i,j)
if c=j
then return 0
m[i,j] ← ∞
for k ← i to j-1
  do q ← R-M-C (p, i, k) + R-M-C
  (p, k+1, j) + pi-1 pk pj
if q < m[i,j]
 then m[i,j] ← q
return m[i,j]
```



Memoization

* It is variation of dynamic programming
* It is a top-down technique but problem must have optimal substructure and overlapping subproblems property.
* It solves every subproblem just once & saves its answer in a table to avoid recomputing answer every time the subproblem is encountered.
* Each table entry initially contains special value to indicate that the entry has yet to be filled in. When subproblem is first encountered during execution of recursive algorithm, its solution is computed and then stored in table.
* Each subsequent time that subproblem is encountered value stored in table is simply looked up and returned.

Solving MCM problem using Memoization

```
Memoized-Matrix Chain (p, n)
    for i ← 1 to n
        do for j ← i to n
            do m[i, j] ← ∞
    return look-up-chain (p, 1, n)
```

(we have not found entry, ∞ value is ∞)

Algo for look-up-chain

```
look-up-chain (p, i, j)
    if m[i, j] < ∞
    then return m[i, j]
    if i = j
    then m[i, j] ← 0
    else for k ← i to j - 1
    do q ← look-up-chain(p, i, k)+
look-up-chain (p, k+1, j) +
p[i-1] p[k] p[j]
    if q < m[i, j]
    then m[i, j] ← q
    return m[i, j]
```

T, BST

Optimal binary search tree

Suppose we are designing a program to translate text from English to French. One way to perform these lookup operations is to build a binary search tree. We want frequently occurring words in the text to be placed nearer the root and also words which have no French translation might not appear in BST at all. We need an OBST.

Given a sequence $K = (k_1, k_2 \ldots k_n)$ of n distinct keys in sorted order (ie $k_1 < k_2 \ldots < k_n$) and we wish to build a BST from these. Each key k_i has a prob. p_i. Some search may be for values not in k, so we have $(n+1)$ dummy keys $d_0, d_1, \ldots d_n$.

d_0 represents all values less than k_1.
d_n " " " " " k_n.
And for $i = 1, 2 \ldots n-1$, d_i represents all values between k_i and k_{i+1}.

For each d_i, we have prob. q_i.
k_i is an internal node and each dummy key d_i is a leaf.

Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \quad \text{---} (1)$$

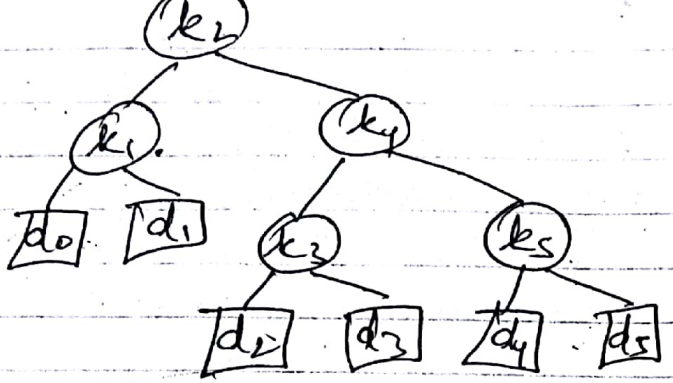
Let us assume that actual cost of search is the no. of nodes examined ie depth of node found by search in T, plus 1. Then the expected cost of search in T is:-

$$E[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

From (1), we have

$$\Rightarrow 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

depth_T: node's depth in tree T.



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(depth+1) × prob =

Node	depth	prob.	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40

This BST has Expected cost of 2.80

2.80

we want to find BST with lowest cost. Such a tree is OBST.

Structure of an optimal binary search tree

Consider any subtree of BST, which has keys $k_i \dots k_j$ for $1 \le i \le j \le n$, and has dummy keys $d_{i-1}, \dots d_j$

Given keys $k_i \dots k_j$, one of these keys k_r $(i \le r \le j)$ will be root.

(root) k_r

$k_i \dots k_{r-1}$ (left subtree) $\quad$ $k_{r+1} \dots k_j$ (rt. subtree)

Recursive Solution

Let us define $e[i,j]$ as expected cost of searching an OBST containing keys $k_i \dots k_j$

When keys are $j = i-1$, there are no actual keys, only dummy key d_{i-1}, then:

$$e[i, i-1] = q_{i-1}$$

$$j = i-1 \quad (\text{prob} \dots -1)$$

When $i \leq j$, we need to select a root k_r from among $k_i -- k_j$ and make OBST with keys $k_i -- k_{r-1}$ as lt-subtree + OBST with keys $k_{r+1} -- k_j$ as rt-subtree.

For subtree with keys $k_i -- k_j$, let us denote sum of prob. as

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

If k_r is root, then;

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)] + (e[r+1,j] + w(r+1,j))$$

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

we have

$$e[i,j] = e(i,r-1] + e[r+1,j] + w(i,j)$$

```
OPTIMAL-BST (p, q, n)
  for i ← 1 to n+1
    do e[i, i-1] ← q[i-1]
       w[i, i-1] ← q[i-1]
  for l ← 1 to n
    do for i ← 1 to n-l+1
         do j ← i+l-1
            e[i, j] ← ∞
            w[i, j] ← w[i, j-1]
                      + p[j] + q[j]
            for r ← i to j
              do t ← e[i, r-1] + e[r+1, j]
                   + w[i, j]
                 if t < e[i, j]
                 then e[i, j] ← t
                      root[i, j] ← r
  return e and root
```

Running time = $O(n^3)$

1st iteration $e[i, i]$ and $w(i, i)$ $l = 1$

and " $e[i, i+1]$ and $w(i, i+1)$ $l = 2$

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

$\Rightarrow \begin{Bmatrix} 0.05+0.40+0.45, \\ 0.45+0.05+0.45 \end{Bmatrix}$

$\Rightarrow \{0.90, 0.95\}$

$\Rightarrow 0.90$

for $i=1$

i \ j	0	1	2	3	4	5
1	0.05	0.30	0.45	0.55	0.70	1.00
2	—	0.10	0.25	0.35	0.50	0.80
3			0.05	0.15	0.30	0.60
4				0.05	0.20	0.50
5					0.05	0.35
6						0.10

Root

i \ j	1	2	3	4	5
1	1	1	2	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

To construct tree

$k_1 \text{ —— } k_5$ $\quad 1,5 \to 2$