★ Dynamic Programming:

→ n jobs
each with a positive profit
s_i & f_i given
Pick mutually compatible jobs that maximize the profit.

★ DP Pipeline

① Naive Solution

② Optimal Substructure : Optimal Solution to a problem contains within it optimal solution to its subproblems.

③ Overlapping Subproblems: When recursive algorithm visits the same subproblem.

④ DP Recurrence (based on Optimal Substructure)

⑤ Memoization (Top-down)

⑥ Tabulation (Bottom-up) (Iterative)

⑦ Backtracking the Solution.

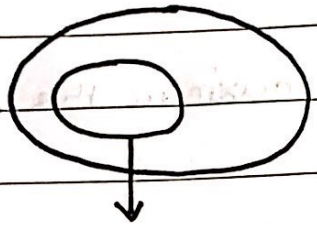
★ Linear Ordering: Order all the subproblems w.r.t to their sizes. ↓ That are to be solved

★ Independent Subproblems: Given any split position k $(A_i \ldots A_k)$, $(A_{k+1} \ldots A_j)$ can be solved independently of each other. i.e. no overlap between 2 subproblems

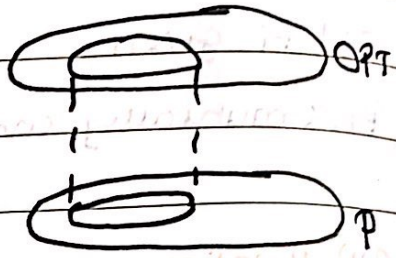
① Weighted Interval Scheduling

① Naive : $O(2^n)$

② Optimal Substructure:



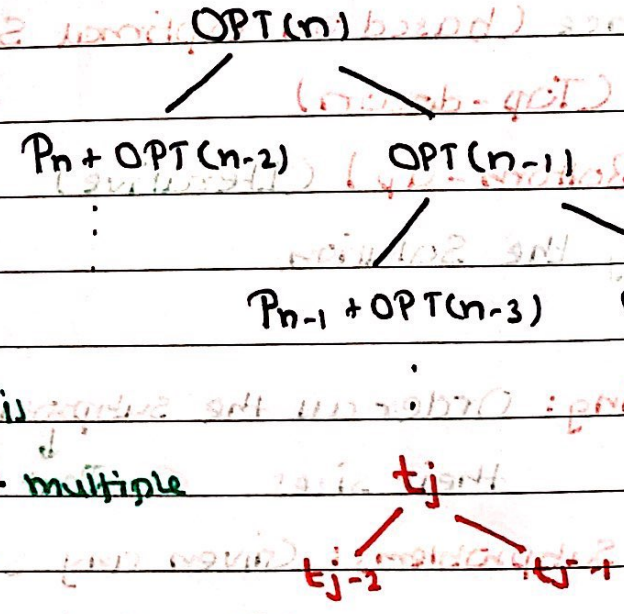
This must be optimal for subproblem of P



③ Overlapping Subproblems:

Every j^{th} job is compatible with $(j-2)^{th}$ job

★ Recursion Tree



→ We can see same subproblem is solved multiple times.

$t_j = 1 + t_{j-1} + t_{j-2}$

$\Rightarrow (t_j + 1) = (t_{j-1} + 1) + (t_{j-2} + 1)$

$f_j = f_{j-1} + f_{j-2}$

$\therefore$ Time required $t_n = f_{n+1} - 1 \approx O(1.63^n)$

→ Linear Ordering: Sort by finish times

p_j: profit of j^{th} job

$OPT(\{1, 2, \ldots, j\}) = OPT(j) \quad 2 \le j \le n$

$OPT(j) = \max(p_j + OPT(\text{compatible}(j)), OPT(j-1)) \quad j \ge 1$

$OPT(0) = 0$

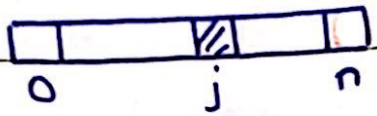
→ Sort jobs by finish times

→ compatible(j) can be found by Binary search, where it represents the index of job i ∋ j-i is minimum & j > i ∋ it is compatible with job j.

④ DP recurrence:

```
OPTVAL(j)
 if j == 0
    return 0
return max(pj + OPT(compatible(j)), OPT(j-1))
```

⑤ Memoization

T: 

(Fill initially with -1)

```
OPTVAL(j)
    if j==0 return 0
    if T[j] != -1 return T[j]
    return T[j] = max(Pj + OPTVAL(compatible(j)),
                      OPTVAL(j-1));
```

TC : O(nlogn)

Workspace : O(n) (Stack + Table)

⑥ Tabulation

⑦ Backtracking

```
Find_Sol(j)
if j==0 then
    output nothing
else
    If Pj + T[compatible(j)] ≥ T[j-1]
    print j;
    Find_Sol(compatible(j));
    else
    Find_sol(j-1);
return
```

② Matrix Chain Multiplication

$A_1 \; A_2 \; \ldots \; A_n$

$p_0 \times p_1 \quad p_1 \times p_2 \quad \ldots \quad p_{n-1} \times p_n$

$((A_1A_2)A_3)$: $p_0p_1p_2 + p_0p_2p_3$

$(A_1(A_2A_3))$: $p_0p_1p_3 + p_1p_2p_3$

} Total scalar multiplications

★ C_n : No. of expressions containing n pairs of parentheses which are correctly matched

$$C_n = \frac{\binom{2n}{n}}{n+1}$$

$$\& \; C_n \sim \frac{4^n}{n^{3/2}\sqrt{\pi}}$$

∴ To calculate min. no. of multiplications (scalar)

① Naive : $\Omega(4^n/n^{3/2})$

② Optimal Substructure :

$(A_1 A_2 \ldots A_k)(A_{k+1} \ldots A_n)$

A B

To compute optimal solution A & B must be optimal

if there was less costly way to parenthesize A our solution is not optimal

③ Overlapping Subproblem

$A_i \ldots A_j$ is computed both in $A_{i-1} \ldots A_j$ & $A_i \ldots A_{j+1}$

④ DP recurrance / Memoization

m[i,j] : no. of scalar multiplications required to multiply

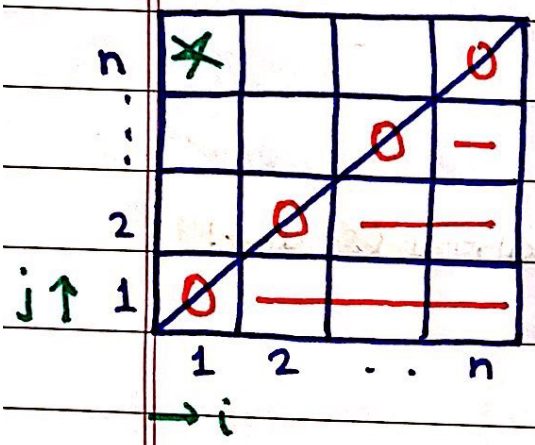
$A_i \ldots A_j \quad (1 \le i \le j \le n)$

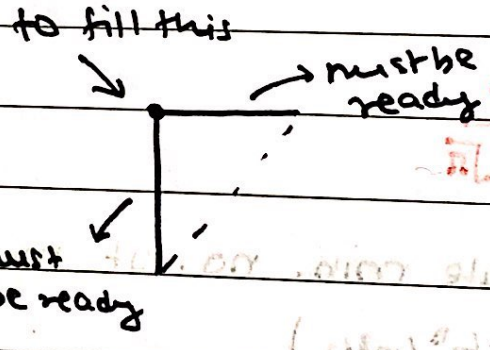
$A_i \ldots A_k A_{k+1} \ldots A_j$

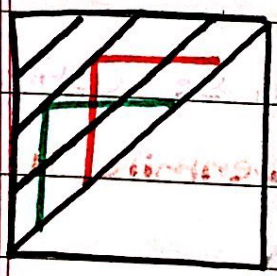
$$m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} p_k p_j)$$

$$m[i,i] = 0$$

⑤ Tabulation







* Space complexity : $O(n^2)$

T.C. : $O(n^3)$

Fill diagonals first

⑥ Backtracking

Store the split points in each cell s[i,j]

* Matrix-Chain-Order

```
for i = 1 to n
    m[i,i] = 0
for l = 2 to n  (Chain length)
  for i = 1 to n-l+1
    j = i+l-1
    m[i,j] = ∞
        for k = i to j-1
            q = m[i,k] + m[k+1,j] + pi-1 pk pj
            if q < m[i,j]
                m[i,j] = q
                s[i,j] = k
return m and s
```

③ Sequence Alignment

X: $x_1 x_2 \ldots x_m$

Y: $y_1 y_2 \ldots y_n$

X CGT

Y ACTC

* ACTG

gap penalty: 5

C _ G T

A C T C

1 5 1 1 = 8

_ C G T

A C T C

5 0 1 1 = 7

_ _ _ _ C G T

A C T C _ _ _

5 5 5 5 5 5 = 35

* Minimise the total penalty

① Naive: $O(\binom{n+m}{n})$

↳ # m+n places choose n

Here repetitions are not considered

$x_1 x_2 \ldots x_i$

$y_1 y_2 \ldots y_j$

① x_i, y_j aligned

② x_i, gap aligned

③ gap, y_j aligned

* $OPT(x_1 x_2 \ldots x_i, y_1 y_2 \ldots y_j)$

$= OPT(i,j)$

$= \min \begin{cases} f(x_i, y_j) + OPT(i-1, j-1) \\ f(x_i, -) + OPT(i-1, j) \\ f(-, y_j) + OPT(i, j-1) \end{cases}$

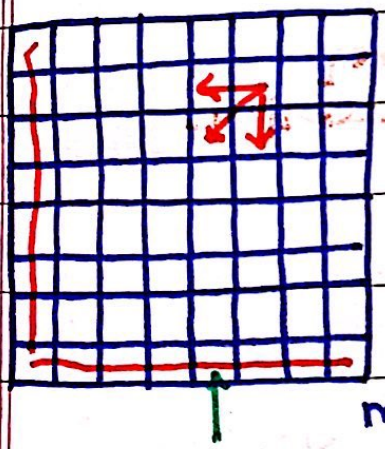
→ $OPT(\phi, y_1 \ldots y_j) = \sum_{k=1}^{j} f(-, y_k)$

$OPT(x_1 \ldots x_i, \phi) = \sum_{k=1}^{i} f(x_k, -)$

$OPT(\phi, \phi) = 0$

} Base Case

② Tabulation



Base Case

* filing (i,j) requires

(i-1, j)

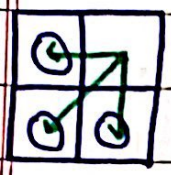
(i, j-1)

(i-1, i-1) to be filled already

* fill entries in O(m+n)

prefix sums

* Proof of Correctness (Use optimal substructure)



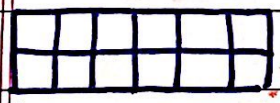
Optimal solution
⇔ the three cells are filled optimally.

Space: O(mn) : To calculate the optimal solution i.e. to print it.

2	
1	3

→ store 1|2|3 in (i,j) to backtrack

O(min(n,m)) to calculate just the optimal value / penalty

2 { 

m (if m ≤ n)

(# only two rows req required for recursion)

④ LCS

C[i,j] be length of LCS for seq x_i, y_j then

$$C[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ C[i-1,j-1]+1 & i,j>0 \ \& \ x_i = y_j \\ \max(C[i-1,j], C[i,j-1]) & i,j>0 \ \& \ x_i \neq y_j \end{cases}$$

⑤ Knapsack Problem (0-1 Knapnsack)

items	weights	value
i_1	w_1	v_1
i_2	w_2	v_2
⋮	⋮	⋮
i_n	w_n	v_n

→ Maximise the value of items taken out if $\sum w_k \leq W$ (Capacity of Knapsack)

→ Fractional Knapsack
(Sort in ↓ order of v_i/w_i. Proof by exchange argument.)

★ $OPT(j, W') = (v_j + OPT(j-1, W' - w_j), OPT(j-1, W'))$
max of these two

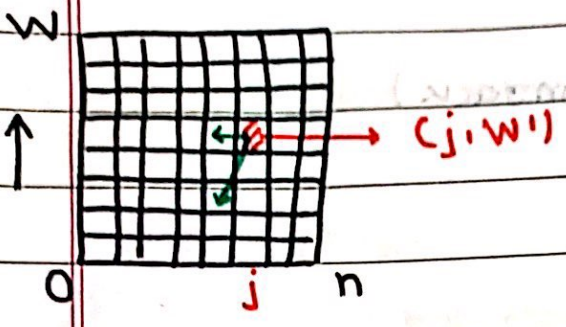
if $W' \geq w_j$

else $OPT(j, W') = OPT(j-1, W')$

$OPT(0, W') = 0$

$OPT(j, 0) = 0$ 1 2 ... j ... n

★ No independent subproblems



* Store 0 or 1 in each cell of new table M.

0: exclude

1: include

T.C. $O(nW)$

$O(n2^{\log W})$: Psuedo Polynomial Time

Hard Problem

→ OPTIMAL SOLUTION given M can be found in $O(n)$

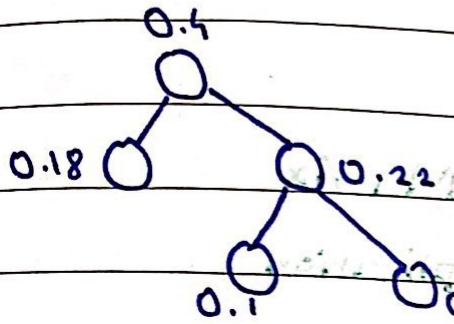
⑥ Offline BST

$\{k_1, k_2, \ldots, k_n\}$ $\quad \sum_{i=1}^{n} p_i = 1$

$p_1 \quad p_2 \quad \ldots \quad p_n$

→ Expected search cost to be minimized

∴ ↑ prob key should be closer to the root



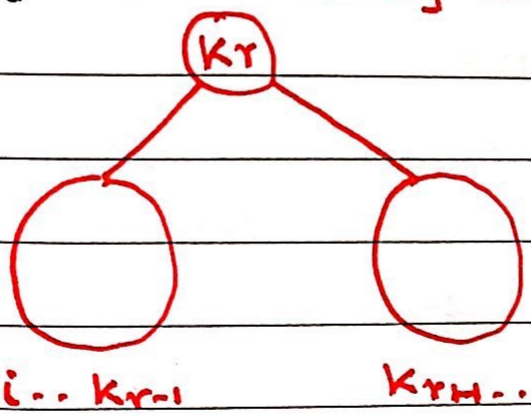
$(0.4 + 2(0.18 + 0.22) + 3(0.1 + 0.1))$

∴ expected search cost

$= \sum_{i=1}^{n} p(k_i)(d(k_i) + 1)$

→ Organise

$k_i \ldots k_r \ldots k_j$ (Sorted) (Key values)



★ Cut & paste argument.

★ Independent Subproblems ✓

(Subtrees solved independently)

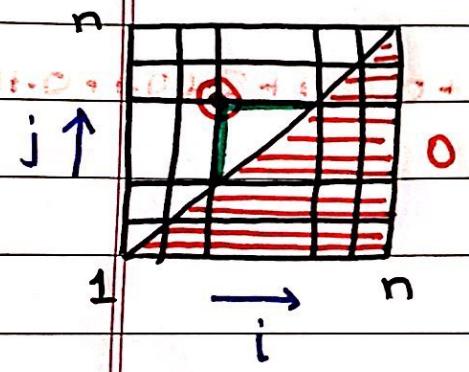
$$\star\ OPT(i,j) = OPT(K_i \ldots K_j)$$

$$= \min_{i \le r \le j} \left\{ OPT(K_i \ldots K_{r-1}) + OPT(K_{r+1} \ldots K_j) + \sum_{u=i}^{j} P(K_u) \right.$$

$$= P_i \text{ if } (i=j)$$

$$= 0 \text{ if } j<i$$

$$\# \text{ Cost} = \sum P(k)(d(k)+1)$$



★ Similar to Matrix Chain Multiplication

T.C. $O(n^3)$