PAGE No.
DATE / /

DAA Week 7 Notes

Approach to DP

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a Bottom up Fashion.
4. Construct an optimal solution from computed information.

(*) weighted interval scheduling problem.

- we have 1 resource
- we have n requests labeled 1...N
- Each req has Start Time s_i
  Finish Time f_i
  weight w_i
- Goal: subset $S \subseteq \{1..N\}$ of mutually compatible intervals so as to maximize $\sum_{i \in S} w_i$

There are 2 CASES either a req is in the soln set or not in the soln set.

CASE 1: req i belongs to soln set

OPT SOLN = w_i + OPT SOLN (All req Compatible with i)

CASE 2: req i not in soln set

OPT SOLN = OPT SOLN (All req with job i)

PAGE No.
DATE / /

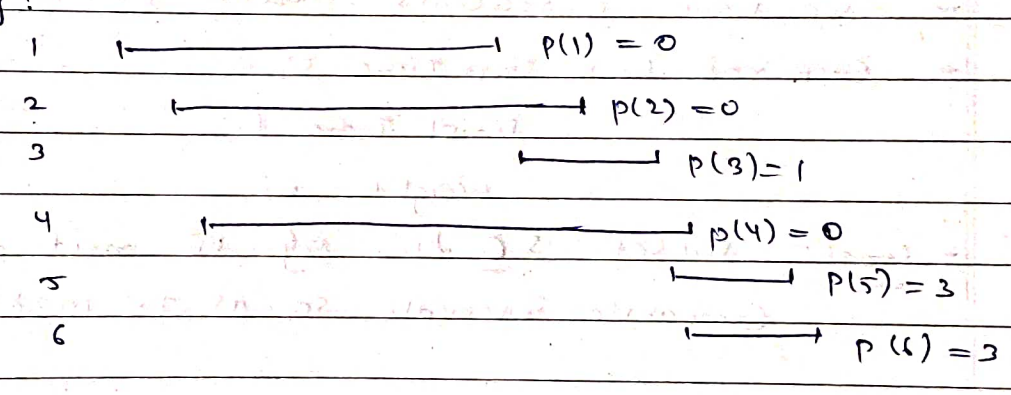
SOLUTION:

Sort req in order of non-decreasing Finish Time
$f_1 \leq f_2 \leq \ldots \leq f_n$.

P(j) → For an interval j to be largest index i<j such that interval i & j are disjoint.

i.e i is the leftmost interval that ends before j begins.

Eg:



O(j) → OPT SOLN TO problem consisting of requests {1...j}

let OPT(j) denote value of Oj

O(3) = {1, 3} OPT(3) = 6

$$OPT(j) = \max \begin{cases} w_j + OPT(P(j)) & \rightarrow j \in O_j \\ OPT(j-1) & \rightarrow j \notin O_j \end{cases}$$

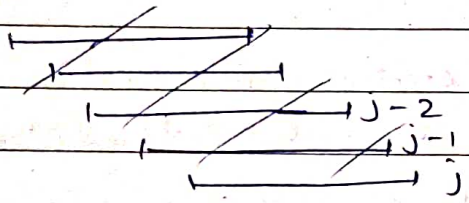
Compute_opt(j)

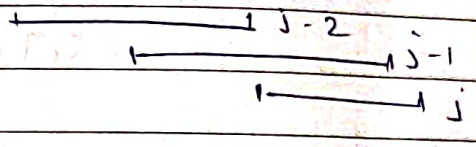
if j==0 ret 0

else

return max(w_j + compute-opt(p(j)) , compute_opt(j-1))

Worst case





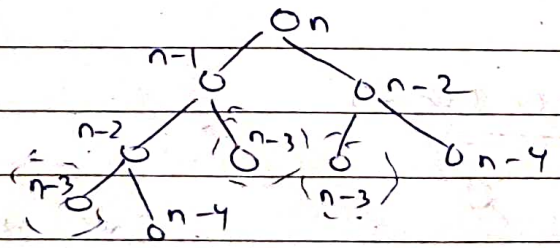
$T(n) = T(n-1) + T(n-2)$

$j \notin o_j$: we reduce j by 1

$j \in o_j$: $p(j) = j-2$

Worst case if each req only overlaps with one req above it and one req below it.

exponential time complexity.



There are various subproblems that repeat in the tree. And we are calculating them again and again.

Instead we should calculate it once and store, and just retrieve when called again

MEMOIZATION

PAGE No.
DATE / /

```
Compute-opt(j):
    if j == 0  return 0

    elif m[j] is not empty
        return m[j]

    else
        m[j] = max( compute_opt(j-1),
                    wj + compute_opt(p(j)) )
        return m[j]
```

O(n) As every subproblem called only once and it is saved when called again.

Initial sorting O(n log n)

~~Make~~ making p[] O(n log n)

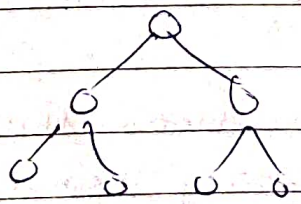
USING BINARY SEARCH TO FIND THE p[] For n elements

n * log n

↑ element ↑ each Binary search Count

Compute-opt O(n)

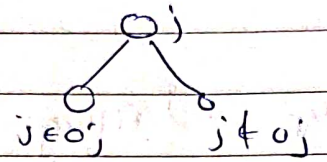
O(n log n)



We do Bottom up to Find optimal value

Then do Top Down to Find the optimal path that was taken.

Compute an optimal soln.



$w_j + OPT(P(j)) \geq OPT(j-1)$

j is in optimal solution if and only if

Find_solution.

if $j > 0$ then

if $w_j + M[P(j)] \geq M[j-1]$ then.

output j together with the results of Find_solution (P(j))

else

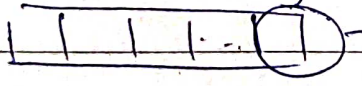
output the results of Find_solution (j-1)

endif

endif

O(n) worst case when subproblem size does not reduce much.

memoized array



→ optimal ANS

what if we are able to build memorized array without recursion.

Try to build memoized array using iteration.

PAGE No.
DATE / /

Top Down {Finding optimal path taken}

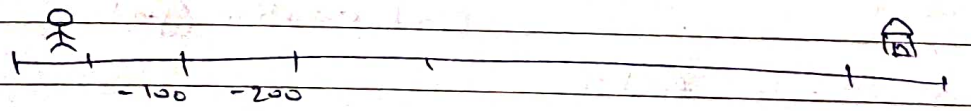
Bottom up {Filling memo}

For i = 1 to n:

$$m[i] = \max(m[i-1] + w_i + m[p(i)])$$

Divide & Conquer	DYNAMIC
• independent subproblems	• dependent subproblems {reduce overlapping subproblems}
• we find optimal soln for each subproblem & combine	• we find optimal value then from that optimal solution.

(*) Video Game Problem:



GOAL: get home at level 'n'

G_0 ← initial energy.

we loose e_i energy when we land on stage i

choices

		Cost
1.	walk into stage	50 units
2.	Jump over a stage	150 units
3.	Jump over 2 stages	350 units

Reach home with maximum energy left.

PAGE No.
DATE / /

We have 'n' unique subproblems.

Stage $0 \to 1$

$0 \to 2$

$0 \to 3$

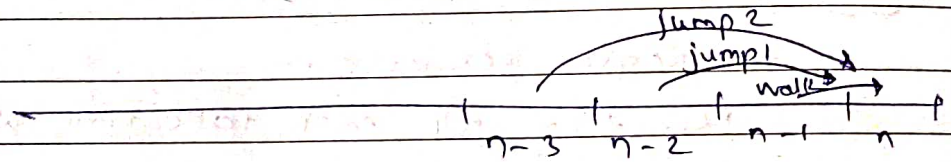
⋮

$0 \to n$.

$OPT(n)$ = opt level of energy when we reach stage n.

Formula (1)

$$OPT(n) = MAX \begin{pmatrix} OPT(n-1) - 50 - e_n, \\ OPT(n-2) - 150 - e_n, \\ OPT(n-3) - 350 - e_n \end{pmatrix}$$



$OPT(0) = E_0$

$OPT(1) = E_0 - 50 - e_1$

$OPT(2) = MAX (\cdot \cdot \cdot)$

For $i = 3$ to n :

Formula (1)

end For.

(*) Austrian coin determination.

Goal : gives less number of coins.

1, 5, 10, 20, 25

$OPT(n)$ = min. no. of coins to pay for n schillings.

Formula (2)

$$OPT(n) = MIN \begin{pmatrix} 1 + OPT(n-1), \\ 1 + OPT(n-5), \\ 1 + OPT(n-10), \\ 1 + OPT(n-20), \\ 1 + OPT(n-25) \end{pmatrix}$$

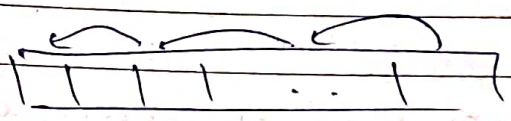
PAGE No.
DATE / /

Initialize OPT[1 . . 25]

For i = 25 to n:

Formula ②

End For.



Find OPT soln then Top Down to find number of each items.

④ 0-1 knapsack & subset sum.

→ single resource.

- Req {1 . . n} each take time w_i to process.
- can schedule jobs at any time between 0 to W

Objective: To schedule jobs such that we maximize the machine's utilization.

OPT(i, w) = value of the OPT solution using a subset of the items {1 . . i} with max allowed weight W

if $n \notin O$ $\quad OPT(n, w) = OPT(n-1, w)$

if $n \in O$ $\quad OPT(n, w) = w_n + OPT(n-1, w - w_n)$

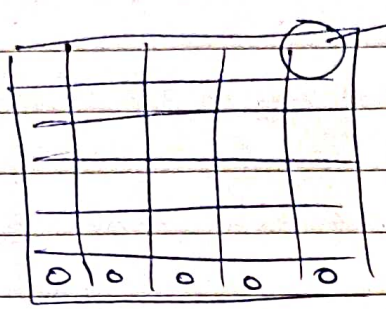
If $w < w_i$:

$$OPT(i, w) = OPT(i-1, w)$$

Formula ③

Else:

$$OPT(i, w) = MAX\begin{pmatrix} OPT(i-1, w) \\ , \\ w_i + OPT(i-1, w - w_i) \end{pmatrix}$$



For i = 1 to N

O(n)

For w = 0 to W

O(W)

use Rec formula (1)

O(1)

Pseudo Polynomial

$O(n \cdot W)$

number of req's

Int weight

Pseudo-Polynomial Time

An algorithm runs in pseudo-polynomial time if its running time is a polynomial time in the numeric value of input.

Polynomial Time

An algorithm runs in polynomial time if its running time is a polynomial in length of the input (or output).