LECTURE 10

Last Class: Kruskal's, Prim's algorithms for MST.

CUT PROPERTY: If you have a weighted undirected graph with all edge weights distinct, then ∀ cut S, the edge of least weight that crosses the cut must be part of the MST.

Analysis of Kruskal's: (Add least weight edge that does not create cycle).

Lemma 1: Every edge the algorithm adds is a "safe" choice; ie; the edge is part of the MST.

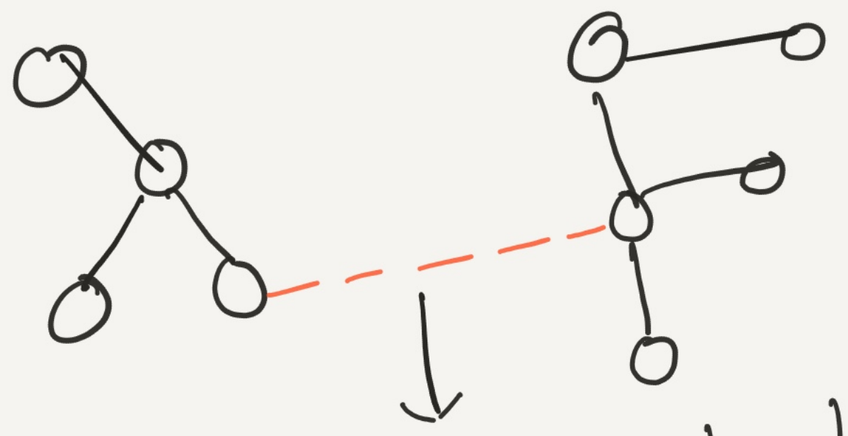
Lemma 2: The algorithm ends with a spanning tree.

2 together $\Rightarrow$ Kruskal's is correct!

Proof of lemma 2: ① Definitely no cycles in the graph T computed by the algorithm

(2) Graph T is connected (all vertices have paths between them).

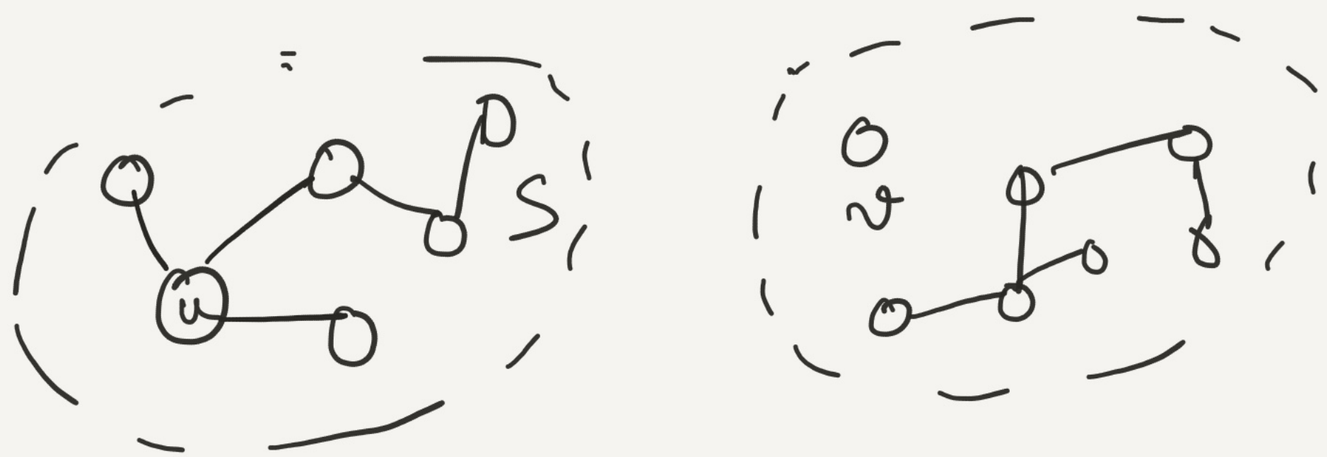
Suppose the graph T is not connected.



there must be an edge crossing from left to right.

Suppose $u, v \in G$ are not connected in the graph T.

Let $S =$ set of all vertices reachable from u in the graph T.



$S \neq \emptyset$ and $v \notin S$, ie; $S \neq V$.

As the original graph was connected, it must have an edge e that crosses the cut S.

$\Rightarrow$ Adding edge e to T would not have created a cycle.

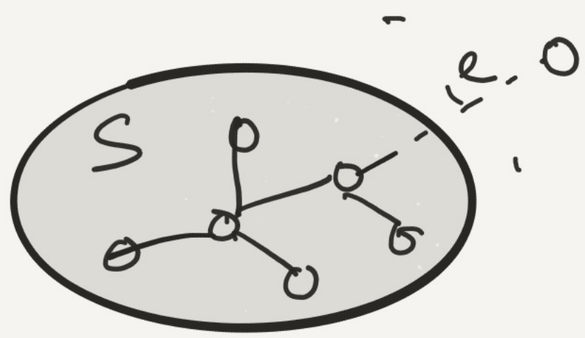
Analyzing Prim's Algorithm for MST.

↓

"Add vertex $v \notin S$ with least attachment cost to S".

Lemma 1: Every edge added by the algorithm is "safe", i.e. is part of the MST.

Proof:



Cut S, Then the edge added by Prim's is the minimum weight edge crossing the cut.

$\Rightarrow$ e is part of the MST (by the cut Property).

Lemma 2: Prim's algorithm returns a MST.

Proof: ① No cycles: Each edge connects to a as yet unseen ("new") vertex.

② T is Connected: We only stop when $S = V$ and we add a vertex by an edge connecting to a previous vertex.

Conclusion: Prim's algorithm returns the MST

Summary of Greedy Algorithms:

① Start with a simple solution, refine it iteratively by a "greedy" rule.

② Coming up with the right rule can be deceptive and tricky.

③ MST:

Dynamic Programming

→ One of the sledge hammer of algorithm design.

→ Be warned: no easy way to teach or learn how to come up with the algorithms than to practice them.

Example 1: Weighted Interval Scheduling Problem

INPUT: Jobs 1: (s_1, f_1) v_1

2: (s_2, f_2) v_2

. .

. .

. .

n: (s_n, f_n) v_n

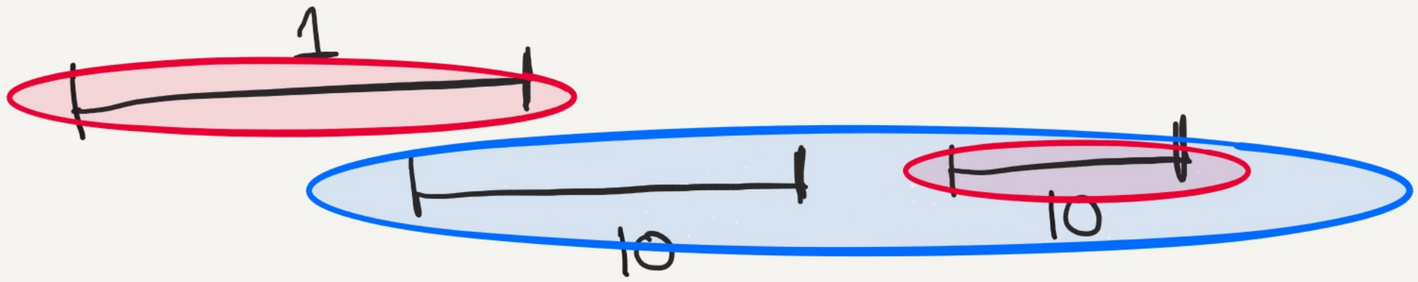
} values

OUTPUT: A subset $S \subseteq \{1, 2, \ldots, n\}$ of non-conflicting jobs with maximum value $\left(\text{i.e., } \sum_{i \in S} v_i\right)$.

Ex: If all values v_i were equal, then this is same as "interval scheduling".

Earliest Finish Time Fails!

Ex:

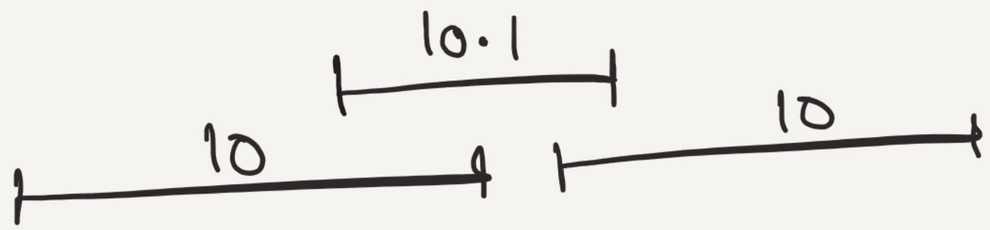


Total value of EFT is 11

Best is 20.

"No natural greedy algorithm is known".

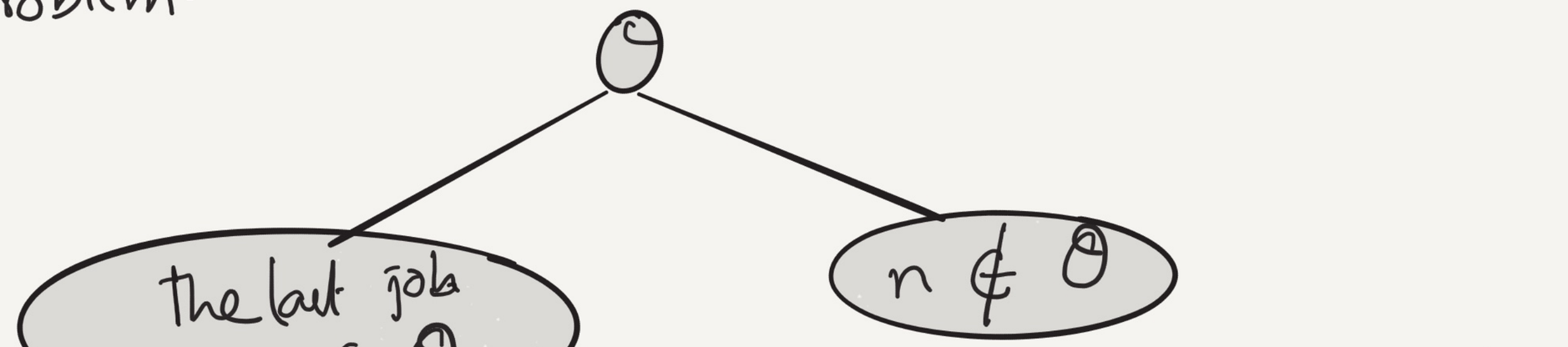
Ex: Sort jobs in decreasing values



Sort the jobs in increasing finish time order.

→ Assume that $f_1 \leq f_2 \leq f_3 \leq \dots \leq f_n$.

→ Consider an optimal solution O for the problem.



⇓

all jobs conflicting with the n^{th} are not part of O.

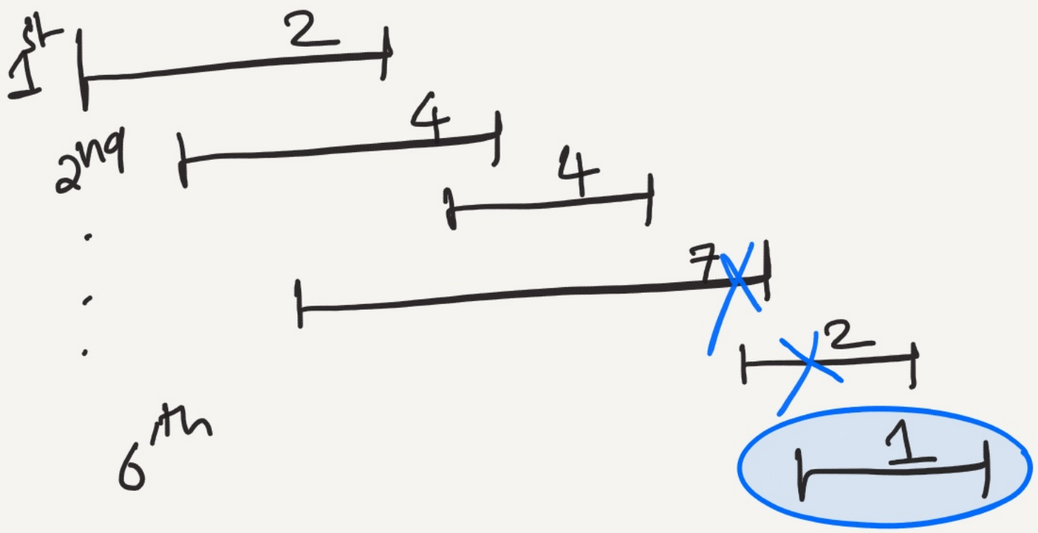
⇓

O is the best for jobs 1 through $n-1$.

$\forall \; j = 1, \ldots, n$

Let $p(j) =$ largest index $i < j$ that <u>does not</u> conflict with j^{th} job

Example:



$p(1) = 0$

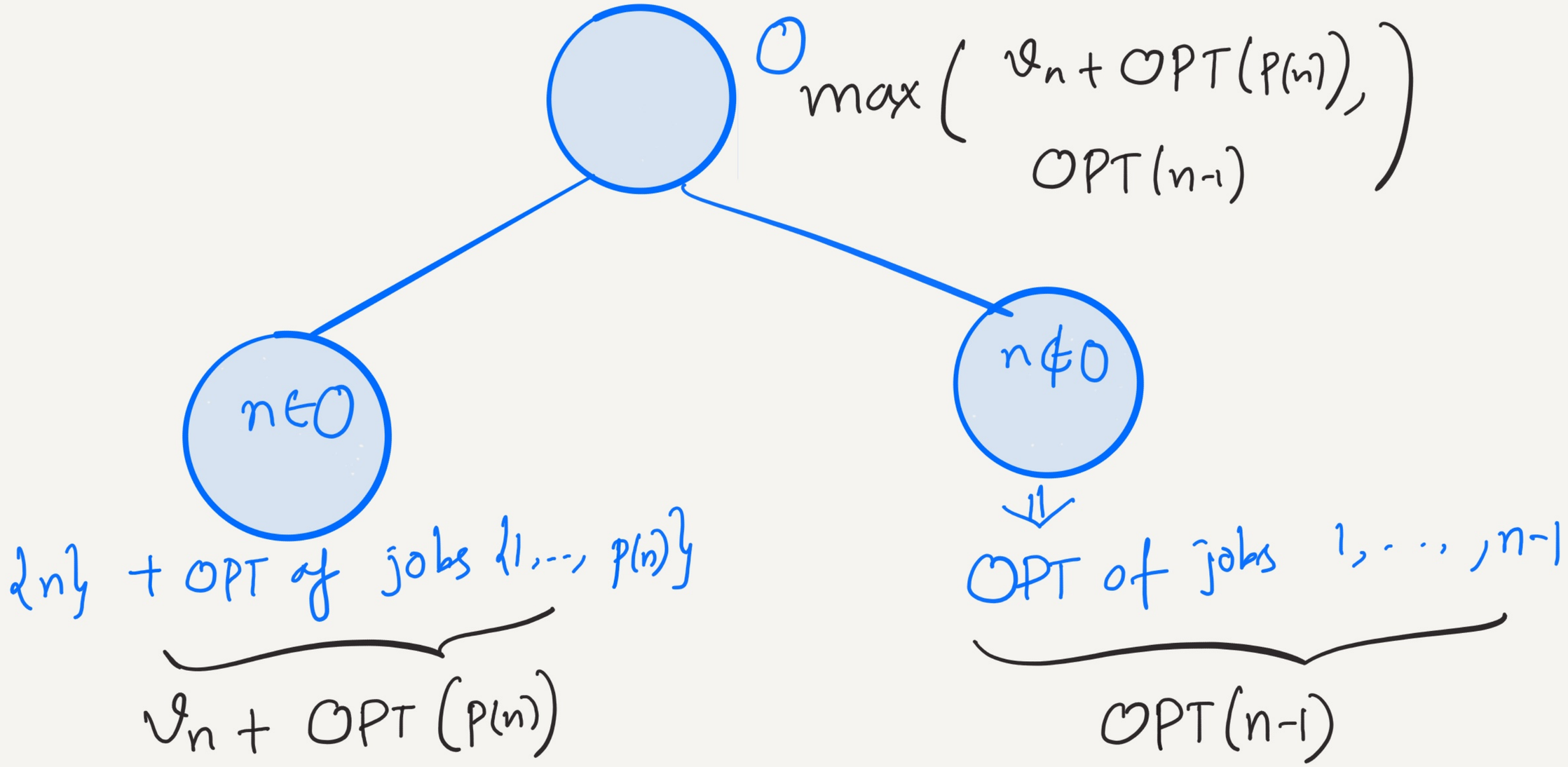
$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$


OPT(n) = max(v_n + OPT(p(n)), OPT(n-1))
n ∈ O
n ∉ O
{n} + OPT of jobs {1, ..., p(n)}
v_n + OPT(p(n))
OPT of jobs 1, ..., n-1
OPT(n-1)

Let O_j denote an optimal solution for jobs $\{1, \dots, j\}$.

and $OPT(j) =$ value of this solution

$$OPT(n) = \max\left(v_n + OPT(p(n)),\ OPT(n-1)\right).$$

$$OPT(j) = \max\left(v_j + OPT(p(j)),\ OPT(j-1)\right)$$

1st algorithm to compute OPT

Compute-Opt(j)

→ If j=0

RETURN 0

→ ELSE RETURN MAX(v_j + COMPUTE-OPT(P(j)),

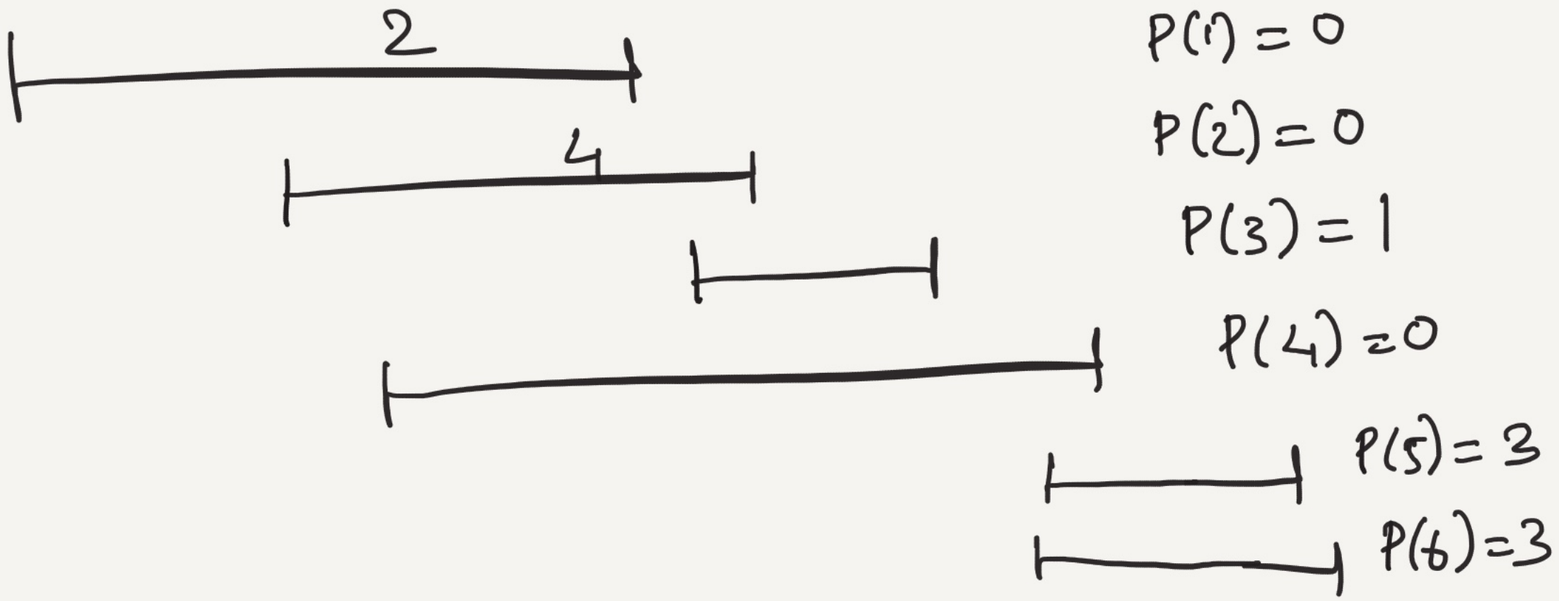
COMPUTE-OPT(j-1)) .

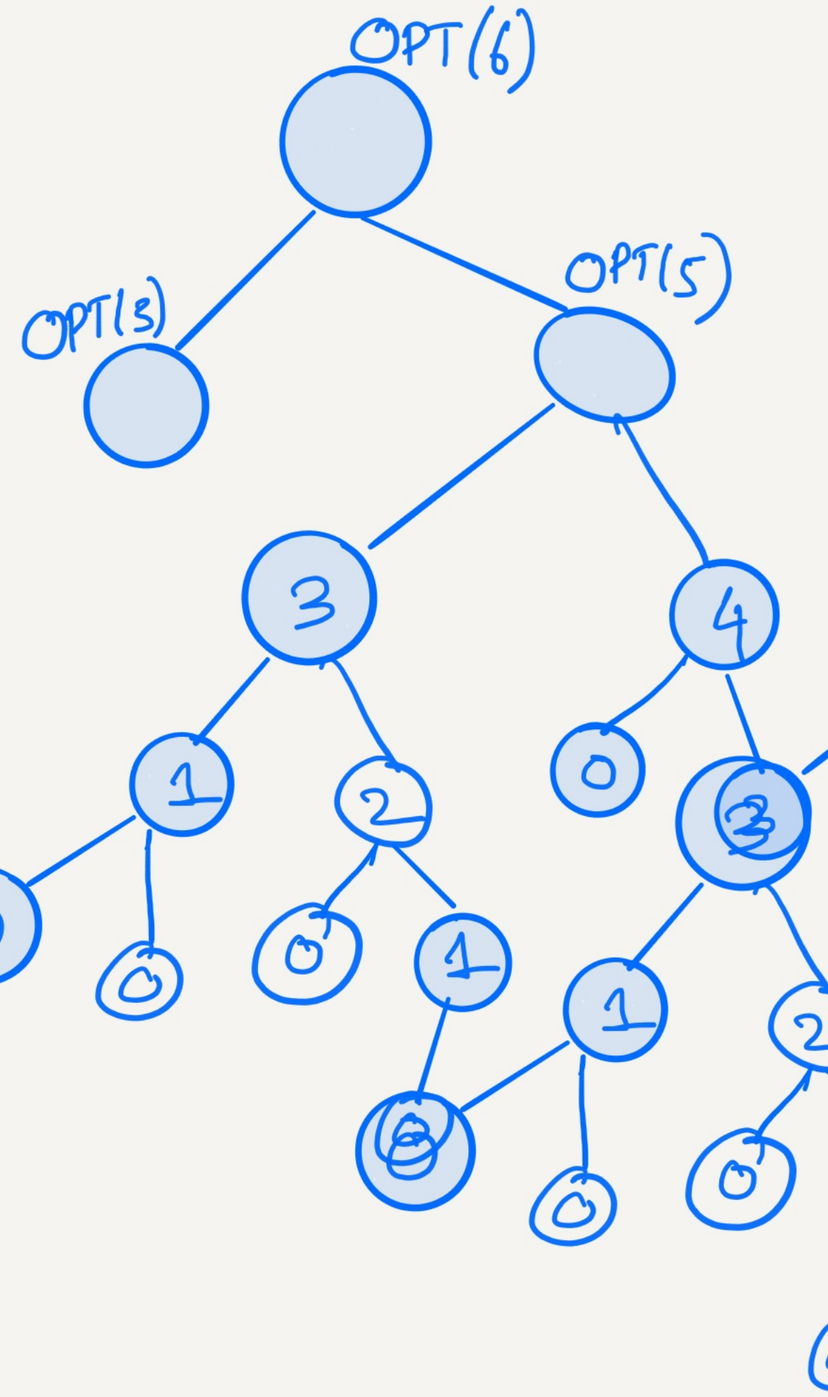
Compute-Opt Computes OPT(j) Correctly

Proof: Induction on j. Use the recurrence relation we showed.

Run-time of the algorithm is Exponential

Ex:





Issue: Spectacularly Redundant

"Memoization" / Memoize / Memoized version ...

① Create a global array M of size n, M[i] = null/empty.

M-Compute-OPt(j)

If j=0
RETURN 0

ELSEIF M[j] IS NOT EMPTY
RETURN M[j]

} O(1) time

ELSE
Set $M[j] \leftarrow \max\left(v_j + \text{M-Compute-Opt}(p(j)),\ \text{M-Compute-Opt}(j-1)\right)$

} O(1) + time spent in recursive calls.

Return M[j].

→ M-Compute-Opt Works Correctly, ie; returns OPT(j)

→ M-Compute-Opt takes O(n) time!

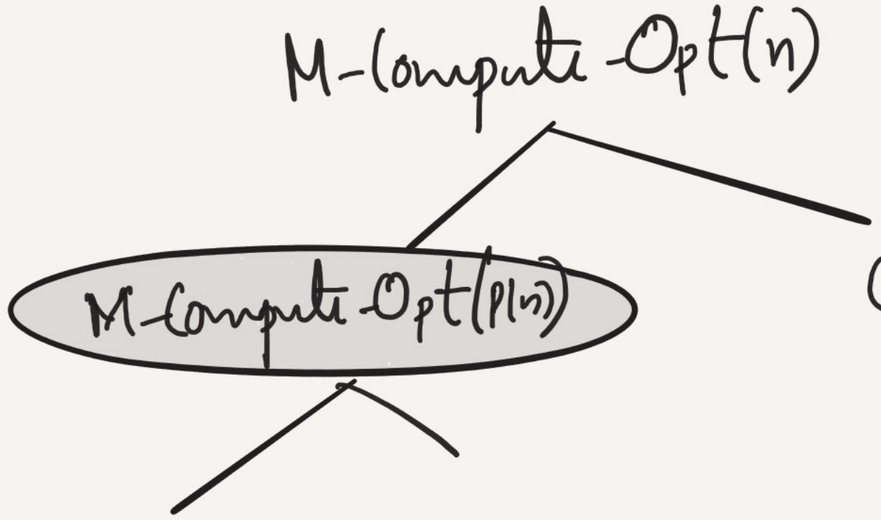
Proof: Thus, the total time spent is

O(# recursive calls to M-Compute-Opt).

Look at # empty cells in M.

Observation: Every call that invokes another recursive call reduces the # empty cells in M by 1.

Initially all of M is empty.



→ If I branch, then one cell is marked non-empty.

⇒ # calls to M-Compute-Opt is at most $\underline{\underline{n}}$.

$\Rightarrow$ Run-time of M-Compute Opt(n)

$= O($ # calls to M-Compute-Opt$)$

$= O(n)$.

An alternate way to view the memoized algorithm.

→ We used the array M and the recursion

→ The recursion used calls to "smaller indices".

Iterative-Compute-Opt:

→ Initialise an array M of size n

→ Set $M[0] = 0$

→ For $j = 1, \ldots, n$

$$M[j] = \max\left(v_j + M[p(j)],\ M[j-1]\right)$$

→ $O(1)$ time per iteration.

Observation 1: Just as before, $M[j]$ computes $OPT(j)$ correctly.

Observation 2: Run-time is $O(n)$.