Dynamic Programming:

- Often the case that no greedy algorithm works, despite what we learned in the section on greedy algorithms.

- Divide & conquer may often help, but many times the reduction won't be from brute-force (often exponential) to tractable (polynomial). Usually, this technique helps polynomial algorithms become faster

- Dynamic Programming - can we decompose the search space such that we can construct provably optimal solutions without ever considering the entire space of solutions explicitly?

Consider the problem of finding the shortest path in a DAG. One can argue that the following very simple Dynamic programming algo will find the shortest path from s to all other nodes.

SPDAG(G, s):

dist[u] = ∞ for all $u \in G - \{s\}$
dist[s] = 0
for each $v \in V - \{s\}$ in topological order:
    $dist[v] = \min_{(u,v) \in E} \{ dist[u] + l(u,v) \}$
return dist

Why does this give the shortest path in all cases?

- The key is the topological order!

The algorithm solves a set of "subproblems" $\{dist[u] \mid u \in V\}$

- we begin with the "smallest" subproblem dist[s] and build up solutions to progressively larger subproblems.

Generally, DP exploits two aspects of problem structure

(1) Optimal substructure: The solution to a "larger" problem can be constructed from the optimal solutions to smaller "subproblems"

(2) Overlapping subproblems: subproblems should not need to be solved over and over again independently.

Consider once more computing the fib sequence.

$$Fib_n = Fib_{n-2} + Fib_{n-1}, \qquad Fib_1 = Fib_2 = 1$$

we saw the recursive algo, but if I asked <u>you</u> to compute Fib_n, how would you do it?

```
fibBU(n):
    if n = 2 or n = 1: return 1
    else:
      fib = [0,1,1]
      for i=3 to n:
          fib.append(fib[i-2] + fib[i-1])
    return fib[n]
```

- What is the runtime of this algorithm?
  (assuming fib(n) fits in a machine word)

- $O(n)$ ... why?

- This is exponentially better than the naive recursive algorithm.
- The following will also work

```
fibMemo = {}

fibTD(n):
    if n=1 or n=2: return 1
    else if n is in fibMemo:
        return fibMemo[n]
    else:
        fibMemo[n] = fibTD(n-2) + fibTD(n-1)
        return fibMemo[n]
```

This solution "remembers" the solution to subproblems it has seen before to avoid recomputing them. This idea is known as "memoization". DPs can usually be written in either a top down (memoized) or bottom-up manner. They usually have the same asymptotic efficiency, though bottom-up is often faster in practice.

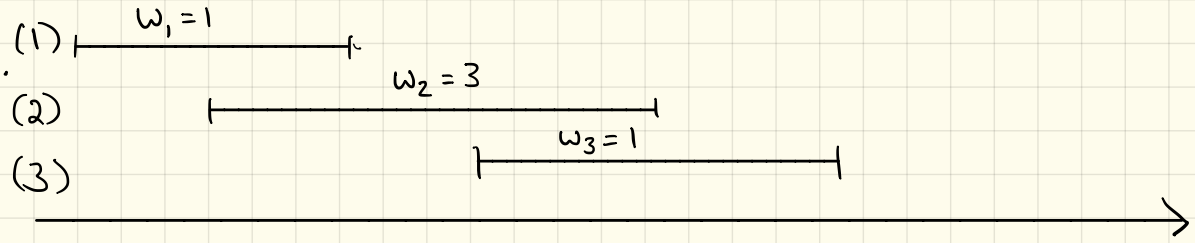
Weighted Interval Scheduling:

Given: A collection of n requests labeled $1, 2, \ldots, n$ each specifying a start time s_i, finish time f_i, and a weight w_i.

Find: The subset $S \subseteq \{1, 2, \ldots, n\}$ that is compatible and of maximum value/weight, where we define

$$w(S) = \sum_{i \in S} w_i$$

E.g.

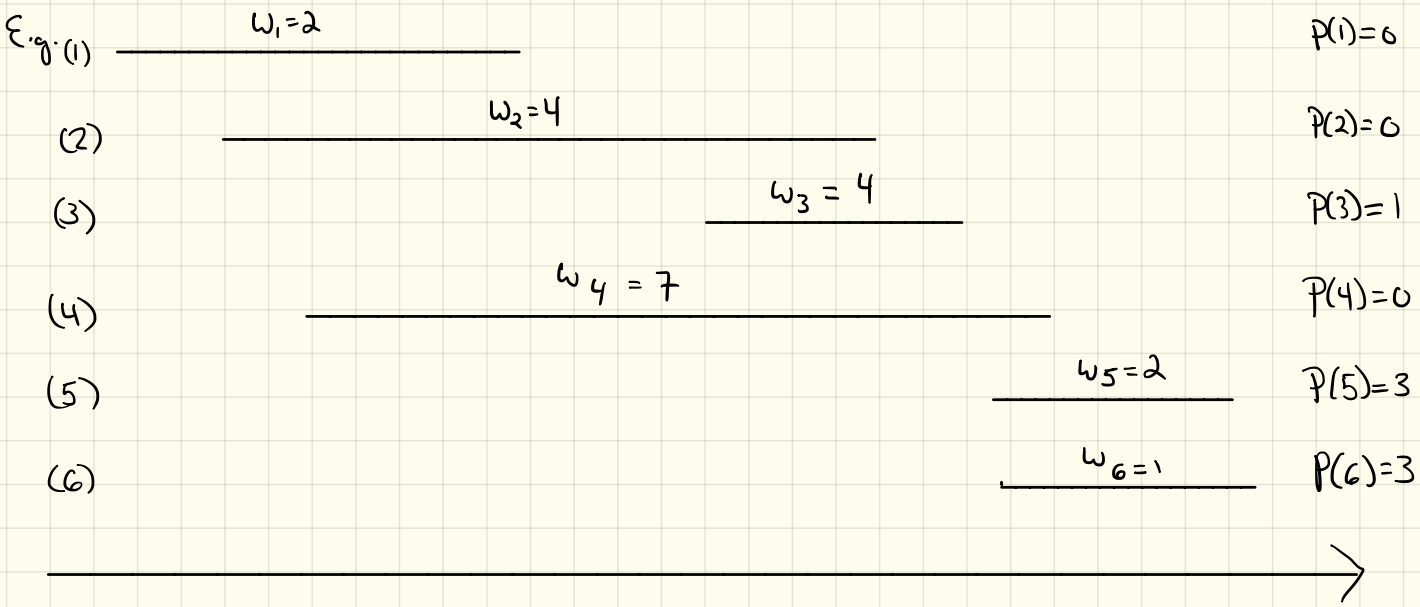


Here, we prefer to choose $S = \{2\}$ since selecting just interval 2 gives a greater weight than selecting $\{1, 3\}$.

How do we search for an optimal solution in this case?

⇒ Assume intervals are sorted by finishing times

⇒ Define function $p(j)$ for interval j to be the largest $i < j$ such that i and j are compatible.



Observe the following about the structure of an optimal solution O

→ Either $n \in O$ or $n \notin O$

→ If $n \in O$ then no interval strictly between $p(n)$ and n can be in O because $p(n)+1, p(n)+2, \ldots, n-1$ must all be incompatible with n.

→ If $n \in O$ then, in addition to n, O must contain the optimal solution to the subproblem $\{1, 2, \ldots, p(n)\}$, why?
- if not, it would not be optimal!

→ In $n \notin O$ then O is the same as the optimal solution of the subproblem $\{1, 2, \ldots, n-1\}$ for the same reason as above.

For any subproblem $\{1, \ldots, j\}$ let O_j be an optimal sol. and let $OPT(j)$ be the weight of O_j. We know $OPT(\emptyset) = 0$

We seek O_n and $OPT(n)$. Using our reasoning above, for some $\{1, \ldots, j\}$ either

$$j \in O_j \Rightarrow OPT(j) = w_j + OPT(p(j)) \quad \text{or}$$

$$j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$$

So, there are only 2 choices! Another way to write this is

$$OPT(j) = \max\left[OPT(j-1) \;,\; w_j + OPT(p(j)) \right]$$

i.e. choose whichever is better.

This gives that $j \in O_j \iff w_j + OPT(p(j)) \geq OPT(j-1)$

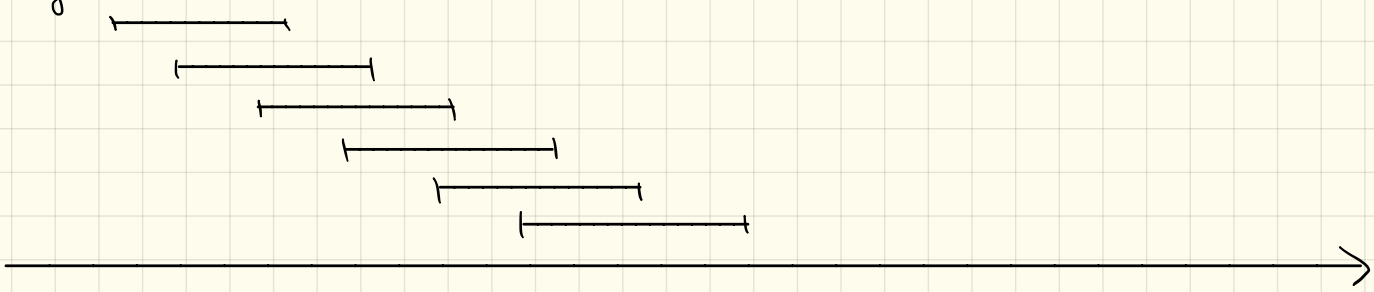
$\Rightarrow$ These simple observations lead us toward a DP solution for WIS.
Consider the recursive algo

```
RecOpt(j):
    if j=0 : return 0
    else : return max(w_j + RecOpt(p(j)), RecOpt(j-1))
```

By induction, this algo is correct, so what is the problem with it?

- → Same issue as with fib
- → Solution to some subproblems is computed repeatedly; could be exponential in the worst case.

E.g.



Two solutions to reduce the runtime:
(1) Memoize RecOpt

$M = [\emptyset, \emptyset, \ldots, \emptyset]$

```
MemOpt(j):
    if j=0 : return 0
    else if j ∈ M:
        return M[j]
    else:
        M[j] = max(wj + MemOpt(p(j)), MemOpt(j-1))
        return M[j]
```

What is the runtime of MemOpt (if $p(\cdot)$ is constant)?
$O(n)$... why?

Proof: Excluding recursive calls, time spent in MemOpt() is $O(1)$. But, since there are only $O(n)$ subproblems, we assign an entry to M at most $O(n)$ times since each pair of recursive calls fills in one value of M. Thus, the total running time of MemOpt is $O(n)$

Solution 2 Rather than rely on memoization, is there an ordering that allows us to avoid recursion?

Consider:

```
ItOpt(j):
    M[0] = 0
    for j = 1, 2, ... n:

        M[j] = max(wj + M[p(j)], M[j-1])
    return M[n]
```

The runtime of ItOpt is clearly $O(n)$... constant work for each of the n steps. So, total time for this problem is dominated by sorting the intervals by finish time.

How would we also return O_n rather than just OPT(n)?

```
If OptSln(n):
    M[0] = 0, S[0] = (∅, -1)
    for j = 1, 2, ... n:
        if wj + M[p(j)] >= M[j-1]:
            M[j] = wj + M[p(j)]
            S[j] = (j, p(j))
        else:
            M[j] = M[j-1]
            S[j] = (∅, j-1)

    Sol = {}
    j = n
    While j ≠ -1:
        if S[j][0] ≠ ∅:
            Sol = Sol ∪ {S[j][0]}
        j = S[j][1]
    return M[n], Sol
```

This part is called "backtracking" or "backtracing". It tells us which decisions we made to arrive at OPT. We see this often in DP.