Dynamic Programming:

- Often the case that <u>no</u> 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 <u>decompose</u> the search space such that we can construct provably optimal solutions without ever considering the entire space of solutions <u>explicitly</u>?

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

SPDAG(G, s):

$\text{dist}[u] = \infty$ for all $u \in G - \{s\}$
$\text{dist}[s] = 0$
for each $v \in V - \{s\}$ in <u>topological order</u>:
$$\text{dist}[v] = \min_{(u,v)\in E} \{\text{dist}[u] + \ell(u,v)\}$$
return dist

Why does this give the shortest path in all cases?

- The key is the <u>topological order</u>!

The algorithm solves a set of "subproblems" $\{\text{dist}[u] \mid u \in V\}$
- we begin with the "smallest" subproblem $\text{dist}[s]$ 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.

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

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.

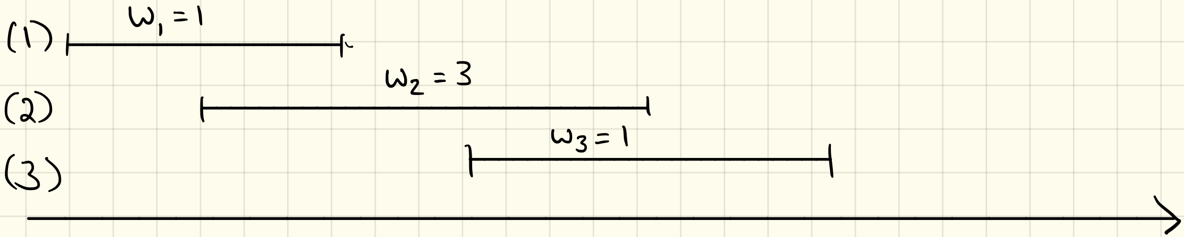
Weighted Interval Scheduling:

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 <u>compatible</u> and of <u>maximum value/weight</u>, where we define

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

E.g.

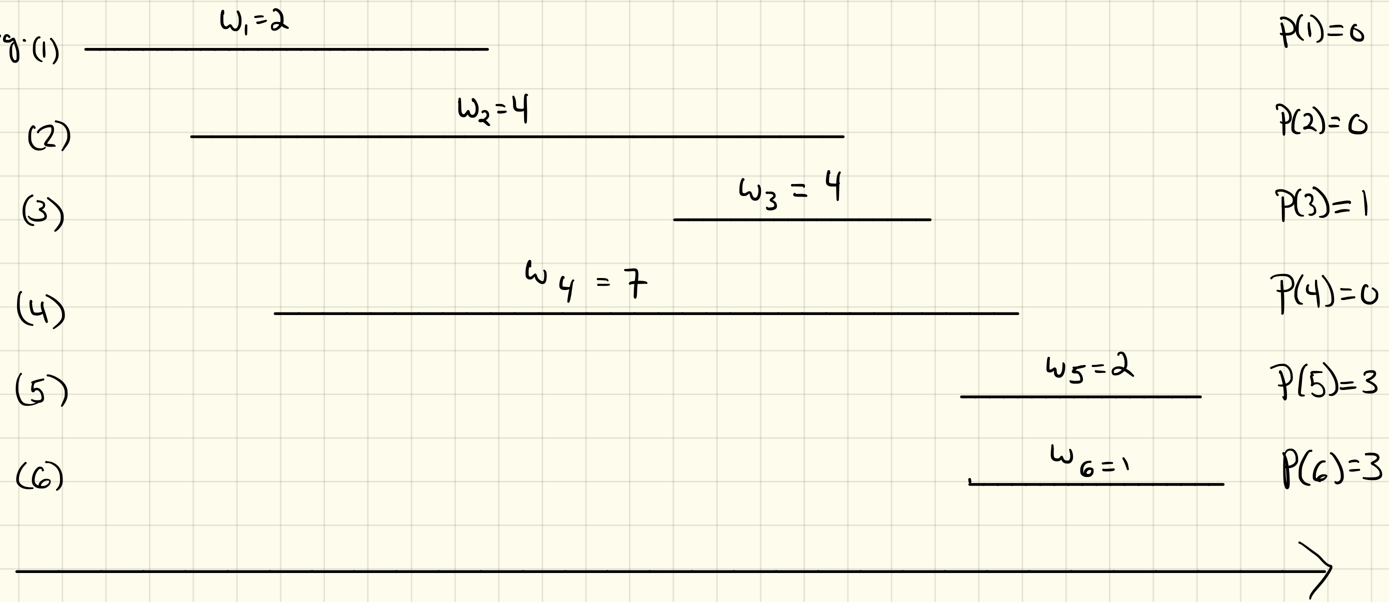


Here, we prefer to choose $S = \{2\}$ since selecting just interval 2 gives a greater weight than selecting $\{1, 3\}$.

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.

E.g.



Observe the following about the structure of an optimal solution $\mathcal{O}$

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

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

- In $n \notin \mathcal{O}$ then $\mathcal{O}$ 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 $\mathcal{O}_j$ be an optimal sol. and let $OPT(j)$ be the weight of $\mathcal{O}_j$. We know $OPT(\emptyset) = 0$

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

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

$$j \notin \mathcal{O}_j \Rightarrow OPT(j) = OPT(j-1)$$

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

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

i.e. choose whichever is better.

This gives that $j \in \mathcal{O}_j \iff w_j + OPT(p(j)) \geq OPT(j-1)$

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

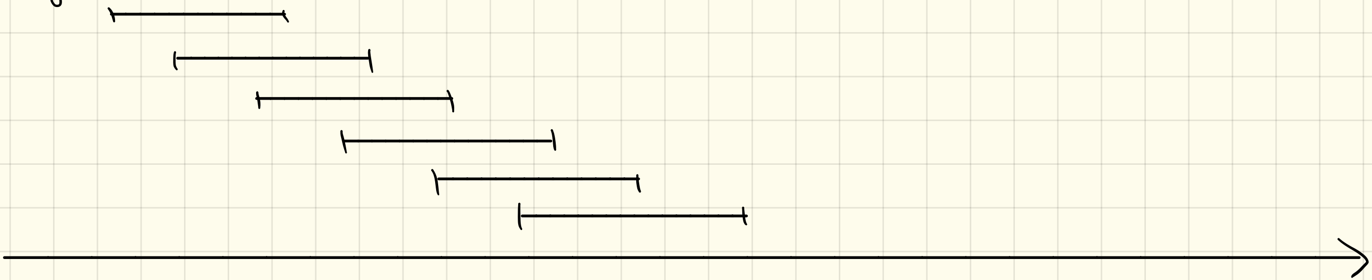
```
RecOpt(j):
    if j=0 : return 0

    else : return max(wj+RecOpt(p(j)), RecOpt(j-1))
```

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

- $\rightarrow$ Same issue as with fib
- $\rightarrow$ 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 $j \in M$:
        return $M[j]$
    else:
        $M[j] = \max(w_j + \text{MemOpt}(p(j)), \text{MemOpt}(j-1))$
        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 $O(n)$

$\boxed{\text{Solution 2}}$ 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 <u>clearly</u> $O(n)$ ... 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 $\mathcal{O}_n$ rather than just OPT(n)?

```
ItOptSln(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 U {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.

Consider a related (but different) problem.

Problem: Subset Sum

Given: A collection of n items, each with a positive integer weight w_i, and an integer bound W.

Find: A subset S of items that maximizes

$$\sum_{i \in S} w_i$$

subject to

$$\left(\sum_{i \in S} w_i \right) \leq W$$

E.g. You have W CPU cycles to use and want to run a set of jobs (each taking w_i cycles) that leaves the fewest idle cycles. This is somewhat similar to job scheduling.

NOTE: The assumption that w_i and W are all integers is important; we will see why later.

Notation:

- Let S^* be an optimal selection of items
- Let $OPT(n, w)$ be the value of S^*

What are the subproblems?

- The single set we used for WIS doesn't work here, why?
- Including n items doesn't necessarily preclude any other item, but just reduces the usable weight budget.

- So, we need to consider both a smaller set of items and a smaller remaining budget to define subproblems in this case.

Consider the following recurrence

$$OPT(j, W) = \max \begin{cases} OPT(j-1, W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W-w_j) & \text{if } j \in S^* \end{cases}$$

$OPT(0, W) = 0$ ← no items

$OPT(j, 0) = 0$ ← no space/budjet

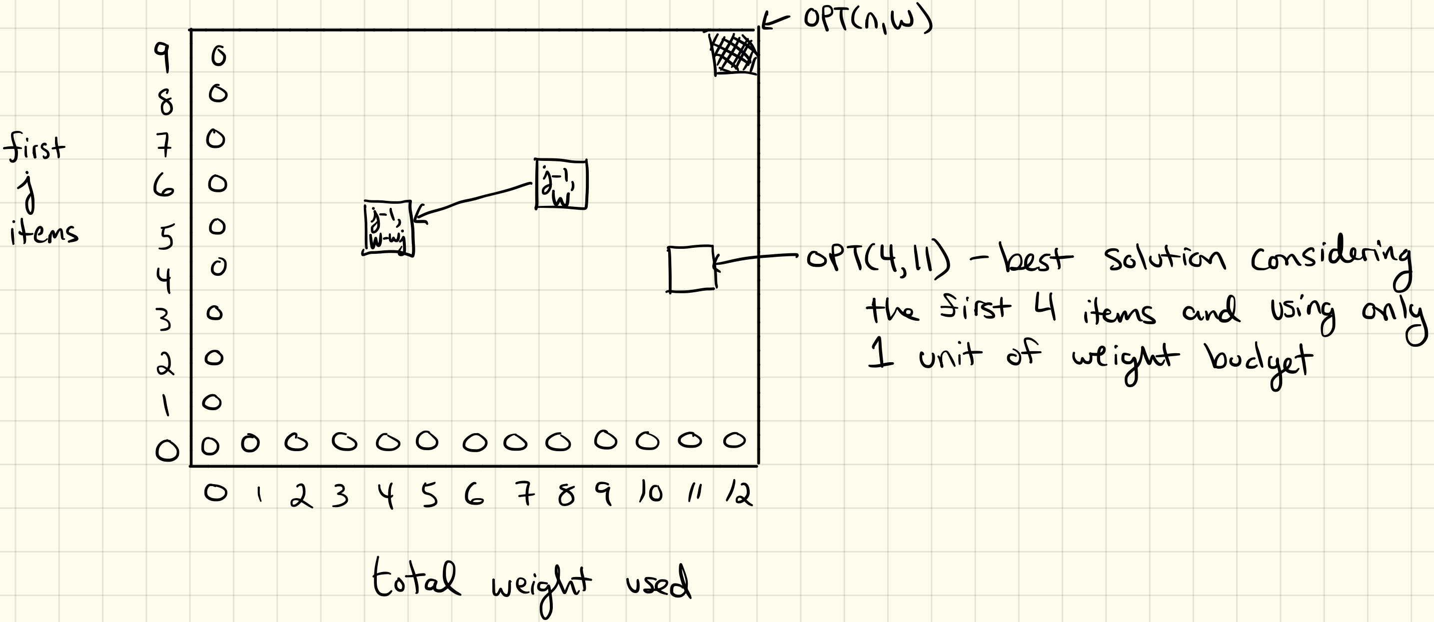
Special case if $w_j > W$ then $OPT(j, W) = OPT(j-1, W)$

Equivalently, we can write.

$$OPT(j,W) = \begin{cases} 0 & \text{if } j=0 \text{ or } W=0 \\ OPT(j-1,W) & \text{if } w_j > W \\ \max \begin{cases} OPT(j-1,W) & \text{if } j \notin S^* \\ w_j + OPT(j-1, W-w_j) & \text{if } j \in S \end{cases} \end{cases}$$

don't know which is better so we must compute both.

If we use the recurrence, we fill in a table that looks like



Each box is filled in by looking at smaller subproblems.

If we fill in the table from bottom left to top right, then every time we need the value of a subproblem, it has already been computed

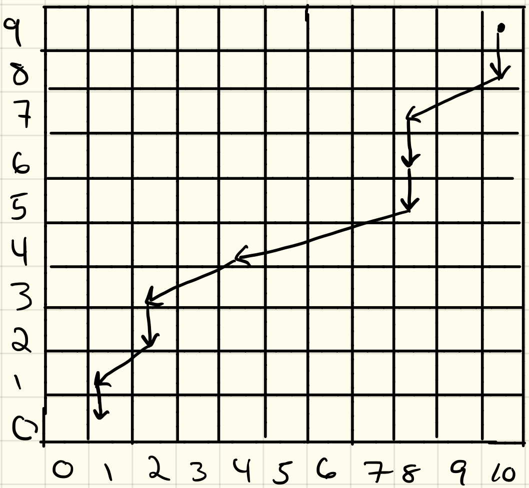
SubsetSum (n, w):

$M[0, r] = 0$ for $r = 0, \dots, W$
$M[j, 0] = 0$ for $j = 0, \dots, n$

```
for j = 1, ..., n :
    for r = 0, ..., W :
        If w[j] > r :
            M[j, r] = M[j-1, r]
        Else
            M[j, r] = max(M[j-1, r], w[j] + M[j-1, r - w[j]]

return M[n, W]
```

To obtain the actual set, we also need to maintain our backpointers, when we fill in a cell, we poin to the cell whose value we used.



What items does this path of back-pointers include?

$\{8, 5, 4, 2\}$

Why?

For each ↓ arrow, we don't use any of the weight budget, which means we can't include the element since <u>all</u> w_i are positive integers.

What is the runtime of Subset Sum?

- Each cell takes $O(1)$ time to decide
- Trace back takes $O(n)$ time
- There are $O(nW)$ cells
- Overall $O(nW + n) = O(nW)$

Why are we including W here, isn't it just a constant factor?

This algorithm is what is known as pseudo-polynomial. The runtime depends not just on n, but on the <u>size</u> of the input weights.

We will learn more about this when we talk about complexity, but it is important to draw a distinction between algorithms whose complexity is polynomial in the number of inputs and those whose complexity is polynomial in the <u>numeric value</u> of the input.

This is particularly important because a polynomial number of bits (e.g. n) can represent a number of exponential value in the number of bits (e.g. 2^n).

A related problem : Knapsack

Given: A collection of n items $\{1, \dots, n\}$ each with a weight w_i, value v_i, and a global weight budget W.

Find: A subset S of items that maximizes:

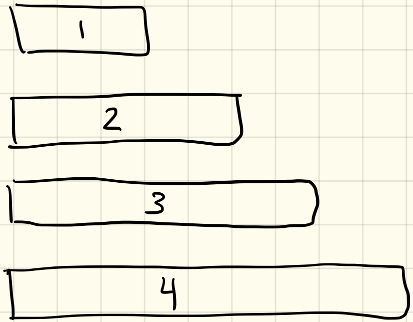
$$\sum_{i \in S} v_i \quad \text{subject to} \quad \sum_{i \in S} w_i \leq W$$

Note: The difference from subset sum here is that you want to maximize the value rather than the weight. For example, a laptop may be worth more than a TV, but be much lighter.

How about a greedy approach?

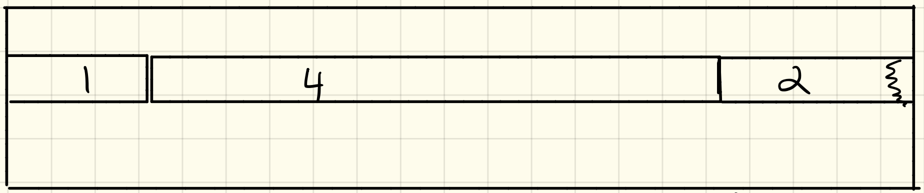
- ⇒ Larger v_i is better
- ⇒ Smaller w_i is better
- ⇒ Sort items by $p_i = v_i / w_i$ (value per unit weight)

E.g.



\$30	1	$p_1 = 30$
\$40	2	$p_2 = 20$
\$45	3	$p_3 = 15$
\$50	4	$p_4 = 25$

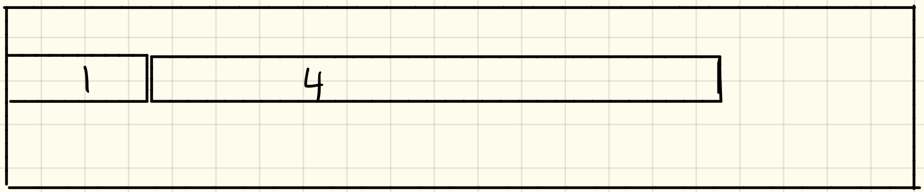
Say knapsack size (W) is 6



\$30 + \$100 + (1/2)(\$20) = \$150

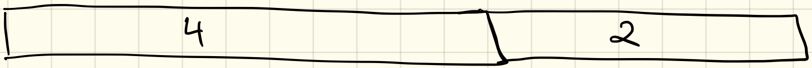
This Greedy approach would work if we could take fractional items, but we can't.

Consider a variant of the greedy alg. that discards elements that don't fit.



\$30 + \$100 = \$130

A better choice is



\$100 + \$40 = \$140

Since we must include an element as "all or nothing", we call this variant 0-1 Knapsack.

Recall the subset sum recurrence

$$OPT(j, w) = \max \begin{cases} OPT(j-1, w) & \text{if } j \notin S^* \\ w_j + OPT(j-1, w-w_j) & \text{if } j \in S^* \end{cases}$$

and consider the following modification for 0-1 knapsack

$$OPT(j, w) = \max \begin{cases} OPT(j-1, w) & \text{if } j \notin S^* \\ v_j + OPT(j-1, w-w_j) & \text{if } j \in S^* \end{cases}$$

Since we have no value "budget" we only have to maximize our value subject to our weight budget. This has the same basic form as subset-sum. A trivial modification of that algorithm solves this problem.