5 steps of Dynamic Programming

1. **Optimal substructure**
2. **Memorization** by define arrays for storing previously computed values
3. **Rewrite the recurrence relation** in terms of arrays defined previously
4. **Bottom-up approach**: write down an iterative solution
5. **Compute a path to an actual solution**

Weighted interval scheduling problem

Given a set of jobs $\{1, 2, \cdots, n\}$ with start time s_i, finish time f_i, and weight w_i for each job at index i. The goal is to schedule jobs in such a way that you obtain maximum possible value/weight.

Solution. □

Sort the jobs by finish time and define $P(j)$ be maximum job i such that $i < j$ and i does not overlap with j (i.e. first job before j that does not overlap with j)

1. **optimal substructure** Let O_n denote an optimal solution and let $OPT(n)$ be such value.
   (a) **Case 1**: $n \in O_n$, then $OPT(n) = w_n + OPT(P(n))$
   (b) **Case 2**: $n \notin O_n$, then $OPT(n-1)$
2. **Define array for caching** Define $M[j]$ be optimal value obtained with jobs $\{1, \cdots, j\}$
3. **Rewrite recurrence relation**:

$$M[j] = \begin{cases} w_j + M[P(j)] & j \in O_j \\ M[j-1] & j \notin O_j \end{cases}$$

4. **Convert from recursive to bottom-up approach**

```
Function Recursive-Compute-OPT (j)
    if j = 0 then
        return 0
    if M[j] is defined then
        return M[j]
    else
        M[j] =
        Max{Recursive-Compute-OPT(j − 1), w_j + Recursive-Compute-OPT(P(j))}
        return M[J]
```

$M[n]$ is the final optimal value. Complexity is $O(n)$ since each job in index is processed just once and the computed result is stored in memo.

Function `Iterative-Compute-OPT`
    $M \leftarrow [0 \cdots n]$
    $M[0] = 0$
    for $j = 1$ **to** n **do**
        $M[j] = Max\{M[j-1], w_j + M[P(j)]\}$
    return $M[n]$

Complexity $\Theta(n)$

5. **Find an actual solution with the optimal value**

Function `Compute-Path` (j, M)
    if $j = 0$ **then**
        return ""
    if $w_j + M[P(j)] > M[j-1]$ **then**
        Output `Compute-Path` $(P[j], m) + j$
    Output `Compute-Path` $(j-1, n)$

weighted interval scheduling where sort by start time and similar recurren relation? does it work? what is special about sorting by finish time

Problem 2: Rod cutting problem

Given a rod of length n, we have $P(i)$ which holds the price of rod with length i. The goal is to cut the rod into pieces such that the prices of the pieces is maximized

1. **Optimal substructure** If you cut the rod at location i then

$$OPT(n) = \underset{1 \leq i \leq n}{Max}\{P(i) + OPT(n-i)\}$$

2. **Array definition**: $M[j]$ holds optimal value on a rod of length j

3. **Recurrence relation**:

$$M[j] = \underset{1 \leq i \leq j}{Max}\{P(i) + M(j-i)\}$$

4. **Bottom-up approach**:

```
Function Bottom-Up-Cut-Rod (P, n)
    M ← [0 ··· n]
    M[0] = 0
    for j = 1 to n do
        for i = 1 to j do
            M[j] = Max{M[j], P[i] + M[j − i]}
```

Complexity $\Theta(n^2)$

5. **Find a way of cutting rod optimally**

```
Function Cut-Rod (P, n)
    M, S ← [0 ··· n]
    M[0] = 0
    for j = 1 to n do
        q ← −∞
        for i = 1 to j do
            if q < P[i] + M[j − i] then
                q = P[i] + M[j − i]
                S[j] = i
        M[j] = q
    return (S, M)
Function Print-Cut-Rod (p, n)
    (S, M) = Cut-Rod(p, n)
    while n > 0 do
        print s[n]
        n = n − s[n]
```

$S[i]$ holds index of first cut in optimal solution for rod of length i

Proposition. *Correctness of the algorithm*

Proof. Prove by strong induction

1. **basis**: $n = 0$, $M[0] = 0$
2. **inductive step**: Assume $M[j]$ is the optimal value for $0 \leq j < n$.i.e. $M[j] = O[j]$ for all $0 \leq j < n$. Now prove $M[n]$ is optimal with dynamic programming. Let the first cut for the optimal solution be at i where $1 \leq i \leq n$, then $O[n] = P[i] + O[n - i]$. By inductive hypothesis then $O[n] = P[i] + M[n - i] \leq M[n]$ (since $M[n] = \underset{1 \leq i \leq j}{Max}\{P(i) + M(j - i)\}$). Since O optimal hence $O[n] = M[n]$.

□

Subset Sum & Knapsack Problem

1. **subset sum** Given jobs $J = \{1, \cdots, n\}$ with non-negative weights $w_1, \cdots, w_n$ The goal is to find $S \subseteq J$ that maximizes $\sum_{i \in S} w_i$ such that $\sum_{i \in S} w_i \leq W$

2. **knapsack problem** Given jobs $J = \{1, \cdots, n\}$ with non-negative weights $w_1, \cdots, w_n$ and value $v_1, \cdots, v_n$ The goal is to find $S \subseteq J$ that maximizes $\sum_{i \in S} v_i$ such that $\sum_{i \in S} w_i \leq W$.

3. Hence subset sum problem is a special case of knapsack problem where $v_i = w_i$

1. **optimal substructure** Let O_n be optimal solution and $OPT(n)$ be the optimal value.

   $OPT(n) = w_n + OPT(n-1)$ wrong because the weight constraint not satisfied

   To take care of the constraint,

   (a) If $n \notin O_n$, then
   $$OPT(n, W) = OPT(n-1, W)$$

   (b) If $n \in O_n$, then
   $$OPT(n, W) = w_n + OPT(n-1, W - w_n)$$

   Hence
   $$OPT(j, W) = Max\{OPT(j-1, W), w_n + OPT(j-1, W - w_j)\}$$

2. **Define array** $M[1 \cdots n][1 \cdots W]$. Hence $M[j][W]$ is the optimal value on $\{1, \cdots, j\}$ jobs with weights $w \in \{1 \leq w \leq W\}$,

3. **Redefine recurrence relation**
   $$M[j][W] = Max\{M[j-1][W], w_n + M[j_i][W - w_j]\}$$

   For knapsack
   $$M[j][W] = Max\{M[j-1][W], v_j + M[j-1][W - w_j]\}$$

   where we use v_j instead of w_j

4. **Bottom-Up Approach**

```
Function Subset-Sum (n, W)
    M ← [0 ··· n][0 ··· W]
    M[0, w] = 0 for w = 0 ··· W
    for j = 1 to n do
        for w = 1 to W do
            if w < w_j then
                M[j][w] = M[j − 1][w]
            else
                M[j][w] = Max{M[j − 1][w], w_n + M[j_i][w − w_j]}
```

Complexity $\Theta(nW)$ Polynomial expression involving an actual input value is called pseudo-polynomial. If W is really large cant really control it... Knapsack is NP hard, so use approximation algorithms instead.

5. **Actual solution** Run through array $M[j][W]$ and figure out if j was is included or not. With time complexity of $\Theta(n)$

Longest Commmon Subsequence Problem

Given two sequences

$$X = \langle x_1, x_2, \cdots, x_m \rangle$$

$$Y = \langle y_1, y_2, \cdots, y_n \rangle$$

The goal is to find a subsequence that is common to both X and Y and that has the maximum possible length

Example.

$$X = \langle 5, 10, 13, 12, 11, 7 \rangle$$

$$Y = \langle 6, 10, 13, 7, 11, 8 \rangle$$

$$\langle 10, 13 \rangle \quad \text{is a commmon subsequence of } X \text{ and } Y$$

$$\langle 10, 13, 11 \rangle \quad \text{is the longest commmon subsequence of } X \text{ and } Y$$

1. **optimal substructure**

   (a) $x_m = y_n$, in other words, the last 2 item in X and Y same, hence

   $$OPT(X, Y) = OPT(X_{1\cdots m-1}, Y_{1\cdots n-1}) + 1$$

   (b) $x_m \neq y_n$

   $$OPT(X_{1\cdots m}, Y_{1\cdots n}) = Max\{OPT(X_{1\cdots m-1}, Y_{1\cdots n}), OPT(X_{1\cdots m}, Y_{1\cdots n-1})\}$$

2. **Array definition** $M[0 \cdots m, 0 \cdots n]$

$$M[i,j] := \text{ legnth of a LCS of } X_{1 \cdots i} \text{ and } Y_{1 \cdots j}$$

3. **recurrence relation**

$$M[i,j] = \begin{cases} M[i-1, j-1] + 1 & \text{if } x_i = x_j \\ Max\{M[i-1,j], M[i,j-1]\} & \text{if } x_i \neq x_j \end{cases}$$

4. **Bottom-Up Approach**

```
Function Longest-Common-Subsequence (X, Y)
    M ← M[0···m, 0···n]
    Initialize M[i,0] and M[0,j] to be zero
    for i = 1 to m do
        for j = 1 to n do
            if X[i] = Y[i] then
                M[i,j] = M[i−1, j−1] + 1
            else
                M[i,j] = Max{M[i−1, j], M[i, j−1]}
    return M[m,n]
```

5. **Actual solution**

```
Function Longest-Common-Subsequence-Path (M, X, Y, i, j)
    if i = 0 or j = 0 then
        return
    else if X[i] = Y[j] then
        return Longest-Common-Subsequence-Path (M, X, Y, i−1, j−1) + X[i]
    else
        if M[i, j−1] > M[i−1, j] then
            return Longest-Common-Subsequence-Path (M, X, Y, i, j−1)
        else
            return Longest-Common-Subsequence-Path (M, X, Y, i−1, j)
```

Complexity $\Theta(m+n)$. each recursion i and j are decremented by 1 each time, so total of $m+n$ function call