DYNAMIC PROGRAMMING

Dynamic Programming is an optimization on the divide & conquer approach where we avoid repetition of solving sub-problems by memorizing solutions.

USE CASE | ROD CUTTING

Input: array $P[1 \ldots n]$ of rod prices, length of rod as n.

Output: Maximum profit by cutting and selling the rod piece by piece.

CASE 1 | Classic Divide & Conquer (Recursion)

① Divide the problem into choosing from multiple solutions of sub-problems corresponding to all lengths $l \leq n$

② Choose and return the maximum value from the solutions

```
MaxRodProfit (P, n) {
    If n=0
        return 0
    max_profit = -∞
    for i=1 to n
        max-profit = max (max-profit,
                          P[i] + MaxRodProfit(
                          P, n-i))
    end for
    return max-profit }
```

Too many repeated subproblems

<u>Runtime</u>: $T(1) = 1$

$$T(n) = n + (T(1) + T(2) + \dots + T(n-1)) \quad ①$$

$$T(n-1) = n-1 + (T(1) + T(2) + \dots + T(n-2)) \quad ②$$

From ① & ②

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

$$\Rightarrow T(n) = 2T(n-1) + 1 = \boxed{2^n - 1}$$

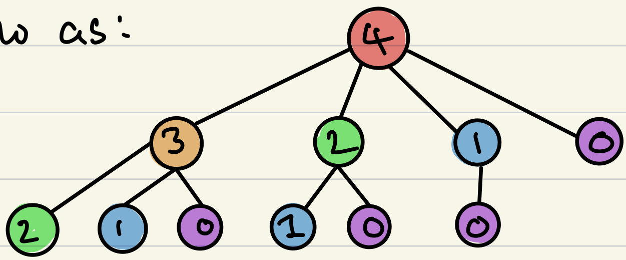
→ Pretty bad runtime!

How do we do better?

Ans: Remember your solutions, solve nothing twice.

▷ Overlapping Subproblems

Let $n = 4$. Our recurrence tree for input "n" would follow as:

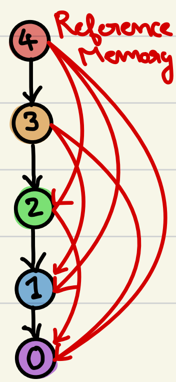


We are solving:
- 4 → 1 time
- 3 → 1 time
- 2 → 2 times
- 1 → 3 times
- 0 → 4 times

0, 1 and 2 show instances of overlapping sub-problems as we solve them more than once.

Case 2 | Memoization (Top-down)

Store the solution to every sub-problem.

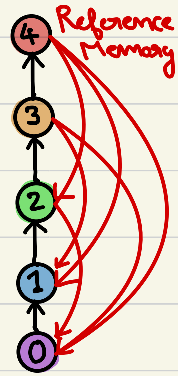



```
MemRodProfit(P,n) {
        M[1,...n] = [-∞] × n
        MRPUtil(P,n) }
```

```
MRPUtil(P,n) {
    if M[n] ≥ 0
        return M[n]        ]-> Refer from memory
    if n=0
        return 0
    max_profit = -∞
    for i=1 to n
        max_profit = max(max_profit, P[i] +
                                   MRPUtil(P,n-i))
    end for
    M[n] = max_profit      ]-> Save to memory
}
```

Case 3 | Tabulation (Bottom-up)

Iteratively calculate from base case to the top call while maintaining a table of solutions.




```
Iterative Rod Profit (P,n) {
    table = [0,...,n]
    table[0]=0
    for i=1 to n
        max_profit = -∞
```

```
for j = 1 to i
    max_profit = max(max_profit, P[j] + table[i-j])
end for
table[i] = max_profit
end for
return table[n]
}
```

Refer from table (memory)

Save to table (memory)

Runtime: $T(n) = n + (n-1) + \dots + 1$

$$\boxed{T(n) = \Theta(n^2)}$$ → For both
- ▷ memoization
- ▷ tabulation

▷ Both are effectively doing the same thing, with the tabulation avoiding overhead of maintaining a stack.

▷ Intuition | 47% – 52% Rule

Figure out 99% (47+52) of a DP problem as follows:

Use Case | Longest Common Subsequence (LCS)

Input: Two strings of length m & n respectively.

Output: Length of the longest common subsequence in the strings.

Case 1 | Brute Force

Check all possible combinations.

Runtime = $T(m,n) = n \cdot 2^m$

Case 2 | Divide And Conquer

We have the following subproblems for $X[1,\dots,m]$ & $Y[1,\dots,n]$

▷ If $X[m] \neq Y[n]$, i.e., either one of:

$X[m] \notin LCS$, $Y[n] \in LCS$

⇒ Shrink scope of "X"

$\Rightarrow LCS(X[1,\dots,m], Y[1,\dots,n]) = LCS(X[1,\dots,m-1], Y[1,\dots,n])$

$X[m] \in LCS$, $Y[n] \notin LCS$

⇒ Shrink scope of "Y"

$\Rightarrow LCS(X[1,\dots,m], Y[1,\dots,n]) = LCS(X[1,\dots,m], Y[1,\dots,n-1])$

Solve for both and choose the max. output

$$\max(LCS(X[1,\dots,m-1], Y[1,\dots,n]), LCS(X[1,\dots,m], Y[1,\dots,n-1]))$$

▷ If $X[m] = Y[n]$, our subsequence grows, hence, check for the next match

$$\boxed{\Rightarrow LCS(X[1,\dots,m], Y[1,\dots,n]) = 1 + LCS(X[1,\dots,m-1], Y[1,\dots,n-1])}$$

As you can expect, the above-mentioned approach has a lot of overlapping subproblems.
We can solve this better with DP.

47% Rule

If the number of subproblems is polynomial:

- ▷ Formally define it
- ▷ Give it a name