11/1/18

DYNAMIC PROGRAMMING

- [Chapter 6 in the Textbook]
- Closest to a systematic recipe for an algorithm.
- Bellman in 1950s a pioneer for this class of algorithm

- What does "Dynamic Programming" even mean?
  - ↳ Name doesn't tell you anything unlike "Greedy" or "Divide & Conquer" algorithm.

- In Divide & Conquer → could split problem into independent portions.

  ↓

  In Dynamic programming, not independent, may have overlap. Although still looking at a smaller subproblem to solve a larger problem.
  → avoid extra work to do!

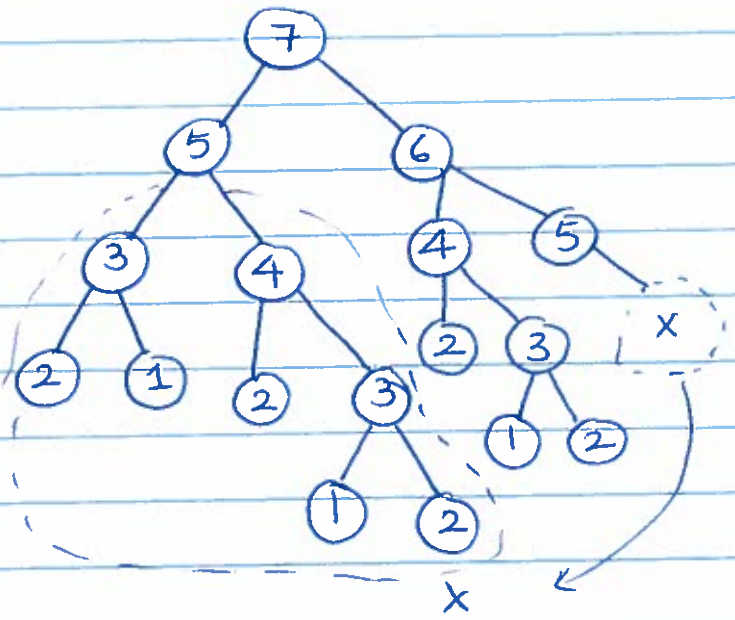
Example:

→ Fibonacci Sequence: → Example of efficiently handling overlapping recursive calls

$$fib(n) = fib(n-2) + fib(n-1)$$
$$fib(1) = 1$$
$$fib(2) = 1$$

Recursive Implementation:

n = 7



Exponential time to run! A lot of repetitions.

$\ominus(1.6^n)$ time to run. So slooowww....

→ We have some tricks up our sleeves!

(1) Memoisation:

Avoid repeated calculations by storing the values computed and reuse them when required later.

```
fib(n):                                   ↑ shared b/w all recursive calls
O(n){ initialize array M of size n to nil
      M[1]=1, M[2]=1
      Mfib(j):
          if M[j] = nil
              M[j] = Mfib[j-2) + Mfib[j-1]
          end if
          return M[j]
```

↑ Memoised version of fibonacci

- Don't recursively called fib(n) otherwise M will get re-initialised and it defeats the purpose.
- We call Mfib( ) recursively.

- Runtime of Mfib(n):

f(n) → Other than recursive calls, it does O(1) work.
fill one entry of M = make 2 recursive calls
⇓
make at most 2n recursive calls.

Overall: [# of recursive calls] x [work per recursive c
$= 2n \cdot O(1)$
$= O(n)$

Went from exponential to linear time! Just had to remember some values.

Why do it recursively & not just unroll it?
↓
(2) Iterative Solution : unwind recursion

fib(n):
initialize array M of size n
M[1] = 1
M[2] = 1

```
for i = 3 to n:
    M[i] = M[i-2] + M[i-1]
return M[n]
```

Runtime: $\Theta(n)$

- Could be times when you would prefer memoisation over iterative solution (for some applications where not all intermediate values need to be computed).

⟹ Dynamic Programming:

Method for solving optimization problems which exhibit:

- overlapping subproblems
- optimal substructure

→ Revisiting Interval Scheduling with Weighted Interval Scheduling -

- find max weight subset of compatible jobs.

- Can't sort by finish time like greedy!

→ Usual Steps of dynamic programming:

- Optimal substructure → Most difficult part!
- find a recursion expression for the value of th optimal solution. ↳ OPT
- memoisation or iteratively compute the table of OPT → analog of M seen in fibonacci
- use OPT table to find actual solution

Optional depending on the problem.

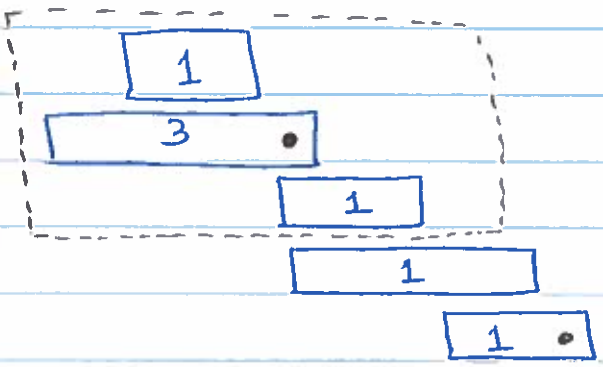
- Let's use this to solve the weighted interval scheduli problem!

Optimal Substructure:

Claim: Let J be the set of jobs.

Suppose $j_1, \ldots, j_i$ is an optimal solution fo

then $j_1, \ldots, j_{i-1}$ is an optimal solution for $J' = J$ removing j_i and all jobs that ove with j_i



Look a smaller subcompo

Proof: Suppose not. Then there is another solution $o_1, \ldots, o_k$ for J' with greater total weight.

Since none of the intervals in J' overlap with j_i, we can replace the first $(i-1)$ intervals of the optimal solution for J $(j_1, \ldots, j_i)$ with $(o_1, \ldots, o_k)$ and increase the total weight. This contradicts that $(j_1, \ldots, j_i)$ is not optimal.