# Dynamic Programming

Examples:

Schedule covid tests? Formalize:

def: set of intervals $\{([s_i,f_i])_{i=1}^{n} : s_i<f_i\}$

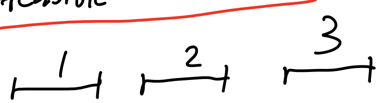
def compatible if $i\neq j$, $\left\{\begin{array}{l} f_i\le s_j \\ f_j\le s_i\end{array}\right.$

def: feasible if: $\forall i,j\in S$, $i\neq j$, $[s_i,f_i]$ compatible with $[s_j,f_j]$

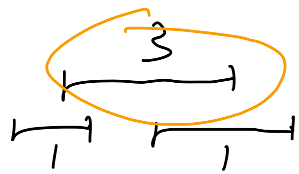
def: weighted interval scheduling: given $([s_i,f_i])_{i=1}^{n}$ and weights $W=\{v_1,\dots,v_n\}$

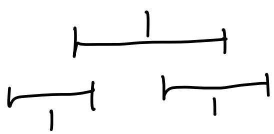
Goal: $\max\limits_{\substack{S\subseteq[n] \\ S\text{ feasible}}} \sum_{i\in S} v_i$

convention, $S=\emptyset \Rightarrow \sum v_i = 0$

e.g. 

→ take all, earn 6

e.g. 


e.g. 

→ take bellow, each 2.

Assumption: $f_1, \dots, f_n$ are sorted. (or sort in $O(n\log n)$)

prop: weighted interval scheduling doable in $O(n \cdot 2^n)$ time.

Pf: Brute force all $\underbrace{P([n])}_{O(2^n)}$, $\underbrace{\text{check if feasible, and}}_{O(n)}$ $\underbrace{\text{output maximum}}_{O(2^n)}$

Better!

def: $0 \le k \le n$ → introduce this new notation and this subproblem

$$OPT_k = \max_{\substack{S \subseteq [k] \\ S \text{ feasible}}} \sum_{i \in S} v_i$$

Given $1 \le i \le n$,

def: $prev(i) = \max \{ j : j \le i, [s_j, f_j] \text{ compatible with } [s_i, f_i] \}$

Observe, $prev(1), \dots, prev(n)$ can be each computed in $O(n\log n)$

prop:

$S \subseteq [n]$ feasible iff either

(a) $S = T \subseteq [n-1]$ feasible

(b) $S = [n] \cup T$, $T \subseteq prev(n)$, $T$ feasible.

Pf:

$\Leftarrow$ : (a) trivial

(b) $T$ feasible, and $T \subseteq prev(n) \Rightarrow f_{i1}, \ldots, f_{ik} \leq s_n \Rightarrow [s_n, f_n]$ compatible with $T$.

Cor

$$OPT_n = \max \begin{cases} OPT_{n-1} \\ OPT_{prev(n)} + v_n \end{cases}$$

**Algo:**

$$\text{SOLVE}(k) = \begin{cases} 0 & \text{if } k=0 \\ \text{output} & \max\begin{cases} \text{SOLVE}(k-1) \\ \text{SOLVE}(\text{prev}(k)) + v_k \end{cases} \end{cases}$$

Recursive Definition!

**Prop:** SOLVE compute WIS in $O(2^n)$

Complexity:

$$T(k) = \max_{j \le k} \{\text{runtime of SOLVE}(j)\}$$

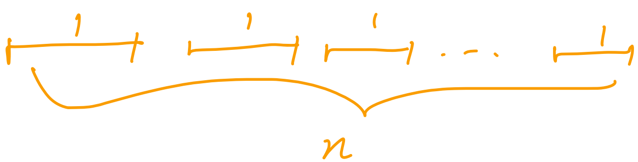
$\rightarrow \le k-1$

$$\Rightarrow \quad T(k) \le T(k-1) + T(\text{prev}(k)) + O(1)$$

$$\le 2T(k-1) + O(1)$$

$$\le O(2^k)$$

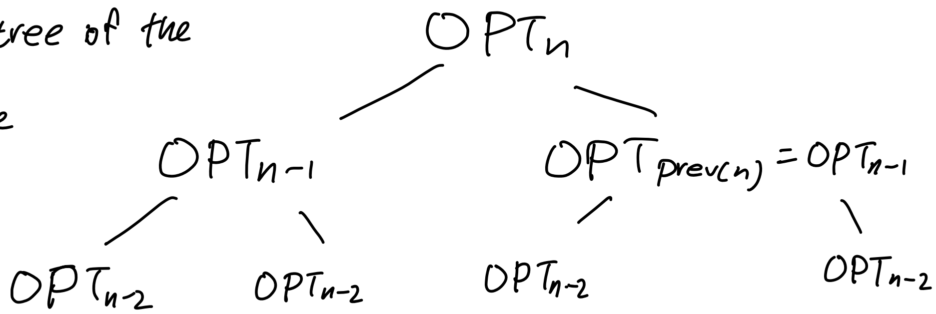
In simple case:




SOLVE($n$) uses $\Omega(2^n)$ time.

# BETTER!

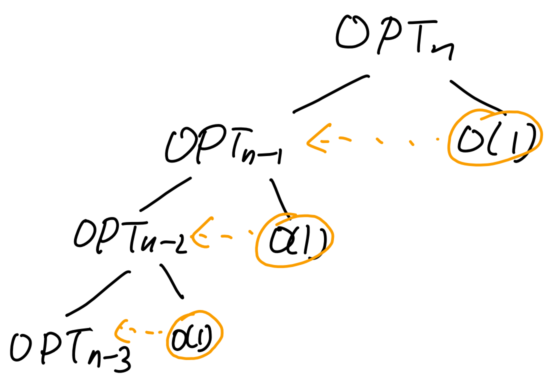
Look at rec tree of the above simple case




problem ⇒ resolving the same problem many times!!!

**Dynamic Programming:** Solve each subproblem only once by storing solutions. ] memoization.

Recursion Tree (refined):




## Algo (DP):

global array $M$ on $[1, \dots, n]$

$$\text{SOLVE-DP}(k) = \begin{cases} 0 & \text{if } k = 0 \\ \max \begin{cases} \text{SOLVE-DP}(k-1) \\ \text{SOLVE-DP}(prev(k)) + v_k \end{cases} & \text{if } M[k] \text{ empty} \\ M[k] & \text{o.w.} \end{cases}$$

Prop: SOLVE-DP(n) solves $O(n)$ to compute WIS

Pf: correctness same as non-DP version.

Complexity: runtime is # of recursive calls. ( $O(n)$ )

claim: # rec calls $\le 2n$

Pf: Subclaim: always true: (2 # empty cells in M + # rec calls) = 2n.

Pf: By induction

---

How about finding the solution to the schedule?

Cor: $\exists$ optimal solution contains $[s_n, f_n]$ iff

$$OPT_{prev} + v_n \geq OPT_{n-1}$$

Pf:

$$OPT_n = \max \begin{cases} OPT_{n-1} & \rightarrow \text{type (a)} \\ OPT_{prev(n)} + v_n & \rightarrow \text{type (b)} \end{cases}$$

Algo :

Globle arr M , N

SOL-DP(k) :

- if k=0

  return (∅, 0)

- if M[k] empty

  - if SOL-DP(prev(k))[1] + Vk ≥ SOL-DP(k-1)[1]

    M[k] =

    N[k] = [k] ∪ SOL-DP(prev(k)) + V[k]

  - else

?

- retur (N[k],

prop: find opt solution in $O(n^2)$

BETTER:

Algo:

$M \leftarrow$ init with SOLVE-DP

SOL-DP-FAST $(k)$ =

- if $k=0$, return nothing $(\emptyset)$
- if $M(\text{prev}(k)) + v_k \geq M[k-1]$
  - output $k$ → or append to $N$.
  - SOL-DP-FAST $(\text{prev}(k))$
- else
  - SOL-DP-FAST $(k-1)$

prop ~~~ in $O(n)$

Complexity: $O(n) + T(n)$

$$T(k) \leq \max\{T(\text{prev}(k)), T(k-1)\} + O(1) \leq O(k)$$

$\Rightarrow$ $O(n)$ in total.

# NON - RECURSIVE → easier to analyze.

Observe :

$OPT_n$
↓
$OPT_{n-1}$
↓
⋮
↓
$OPT_{prev(n)}$

OR

↑ solve bottom up

Also :

SOLVE-ITER(n)

- $M[0] = 0$
- for $k = 0 \sim n$
  - $M[k] = \max \begin{cases} M[k-1] \\ M[prev(k)] + V_k \end{cases}$
- output $M[n]$

prop: SOLVE-ITER computs $OPT_n$ in $O(n)$

complexity: loop $n$ time, $\Rightarrow O(n)$