Dynamic Programming (contd).

→ Scheduling of intervals.

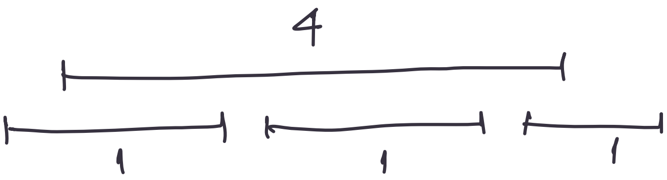
Given a set of requests, we want to get a largest set of compatible requests.

$s(i)$ $f(i)$

$$\left\{\begin{matrix} R_1, & R_2, & \dots, & R_n \\ w_1 & w_2 & \dots & w_n \end{matrix}\right.$$

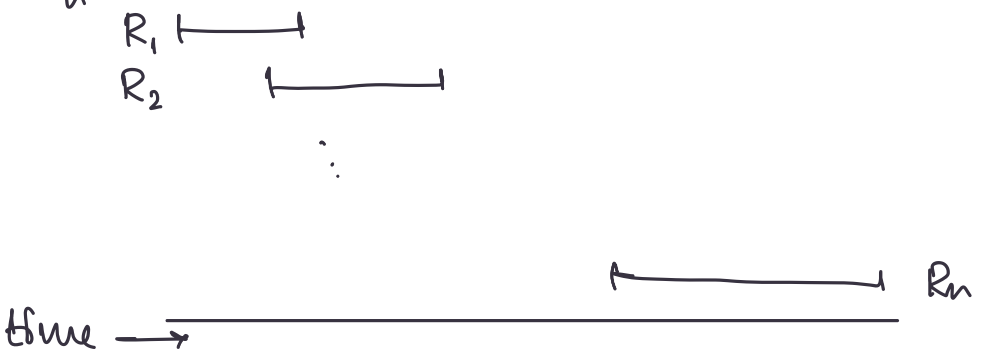
Say $\left.\begin{matrix} R_{i_1}, \dots, R_{i_k} \leftarrow \text{compatible} \\ w_{i_1} \quad\quad w_{i_k} \end{matrix}\right\}$ $\max \sum w_{i_k}$

Question: Find a subset of compatible req. that has maximum wt.



All greedy strategies that we discussed fail.

$\left.\begin{matrix} R_1, & R_2, & \dots, & R_n \\ w_1 & w_2 & & w_n \end{matrix}\right\}$ W.L.O.G let $f(1) \leq f(2) \leq \dots \leq f(n)$



Case-1: R_n belongs to the optimal set O.

Case-2: does not

Case-1: $R_n \in O$.

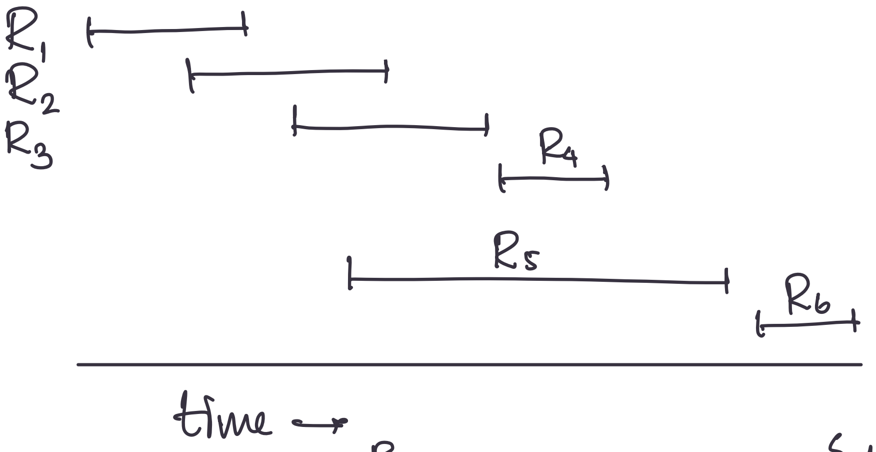
⇒ All those requests that are $\overbrace{\text{not compatible}}^{\text{overlap of intervals.}}$ with R_n do not belong to O.

Let $p(j)$ be the index of the interval s.t it is the largest request that is compatible with R_j.

$$p(j) = \max_{i<j} \{ i \mid R_i \text{ is compatible with } R_j \}.$$

$$\text{Optimal}\left(\{R_1, \dots, R_{p(n)}\}\right) + w_n.$$

In otherwords, $\underbrace{R_{p(n)+1}, \dots, R_{n-1}}_{\text{cannot be part of } O.}$ overlap with R_n.



$P(1) = 0$

$p(2) = 0$

$P(3) = 1$

$P(4) = 3$

$P(5) = 1$

$P(6) = 5$

$$\max \underbrace{\{\overset{R_1}{1}, \overset{R_5}{5}, \overset{R_4}{4}, 2, 3\}}_{R_1, R_2, \dots, R_5 \text{ are compat w/ } R_6} = 5$$

$$\underbrace{\{1, 2, 3\}}_{R_1, R_2, R_3 \text{ are compat w/ } R_4}$$

$$\underbrace{\boxed{R_1, R_2, \ldots, R_{p(n)}}}_{\uparrow}, \underbrace{R_{p(n)+1}, \ldots,}_{\times} \underset{\substack{\uparrow \\ \in O}}{R_n}$$

$$\text{Optimal S|n}(\{R_1, \ldots, R_n\}) \quad \text{Optimal solution}(\{R_1, \ldots, R_{p(n)}\}) + w_n$$

conditioned on $R_n \in O$.

Case-2: If R_n is not in O, then optimal solution for $\{R_1, \ldots, R_n\}$ is given by optimal solution of $\{R_1, \ldots, R_{n-1}\}$.

$$\overset{\text{Opt}(n)}{\text{Optimal Solution}(\{R_1, \ldots, R_n\})}$$

$$= \max \left\{ \overset{w_n + \text{opt}(p(n))}{w_n + \text{Optimal Solution}(\{R_1, \ldots, R_{p(n)}\})}, \underset{\text{opt}(n-1)}{\text{optimal Solution}(\{R_1, \ldots, R_{n-1}\})} \right\}.$$

case when $R_n \in O$

$R_n \notin O$.

$$\text{Opt}(j) = \text{Optimal Solution}(\{R_1, \ldots, R_j\})$$

$$\text{Opt}(j) = \max\{ w_j + \text{opt}(p(j)), \text{opt}(j-1) \}.$$

$\{\text{opt}(1), \ldots, \text{opt}(n)\}$.

Preprocessing:

- Sort the requests in non-decreasing order of finish times
- Compute p() for each request.

Runtime:

- Run Compute-Opt(n).

```
Compute-Opt(j):    ← Add memoization. Init a global arrays M.
                                      and Back Pointers
    (return max{ $w_j$ + Compute-Opt(p(j)), Compute-Opt(j-1)}.
    i=0
    <Base case>
    while i<n+1:
        $M[i] = \max\{\overbrace{w_i + M[p(i)]}^{A_i}, \overset{B_i}{M[i-1]}\}$    // Handle base case.
        if $A_i < B_i$:
            BackPointer[i] = i-1
        else:
            BackPointer[i] = p(i)   ← // Also add i to an optimal solution list.

    return M[j]
```

→ Fractional Knapsack.

→ Bin packing.

→ 0-1 Knapsack.

$$\left.\begin{array}{ll} \text{Items} & I_1, \dots, I_n \\ & w_1, \dots, w_n \end{array}\right\}$$