CMSC 451 - Algorithm Design
Lecture 8 - Dynamic Programming: Weighted Interval Sched.

Dynamic Programming -

- A fundamental algorithm design principle
- Involves recursively breaking a problem into subproblems
- Optimal substructure - Property of some optimization problems. To obtain a globally optimal result, all subproblems should be solved optimally

Isn't this obvious? No- Sometimes it is better to be suboptimal on one subproblem so you can do better on another subproblem

- Developed in 1950's by Richard Bellman
  - Bellman-Ford algorithm
  - Coined the term "curse of dimensionality"

Overlapping subproblems -

- Prevents methods like divide-and-conquer
- Can be solved top-down or bottom-up

Weighted Interval Scheduling -

Given a set of n requests to be scheduled on an exclusive resource.

Each request has:

- start-finish time interval - $[s_i, f_i]$
- weight or value - v_i

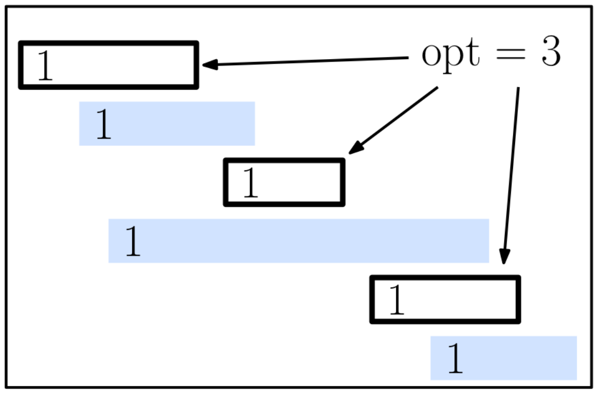
Objective - Schedule a set of non-overlapping requests to maximize sum of weights.

Example: People make bids to use picnic table at local park

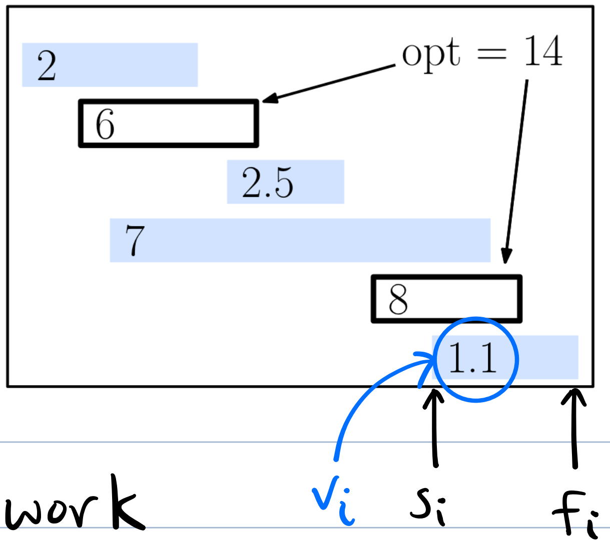
$[s_i, f_i]$ - when they want to use it

v_i - amount they'll pay for use

optimal unweighted



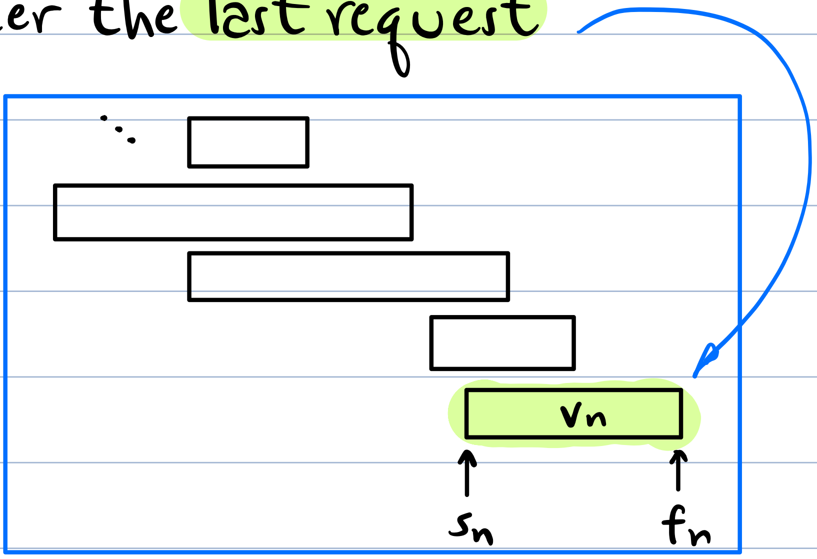
optimal weighted



Note: Greedy does not work for weighted version.

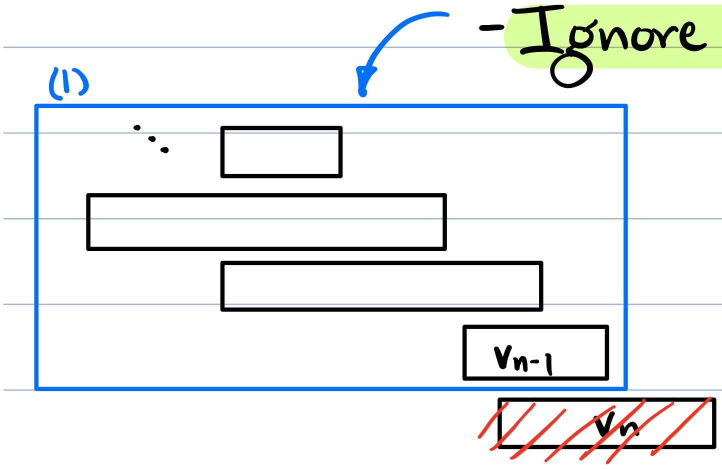
Recursive Formulation:

- Assume requests sorted by finish times
- Consider the last request

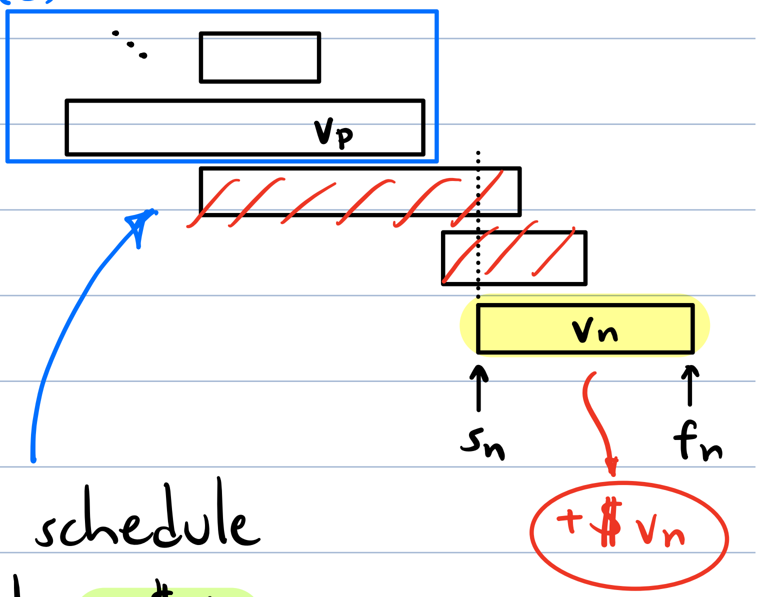


- Two possibilities:
  - (1) $[s_n, f_n]$ not in opt schedule
    - Ignore it + recurse on requests 1..n-1

(1)

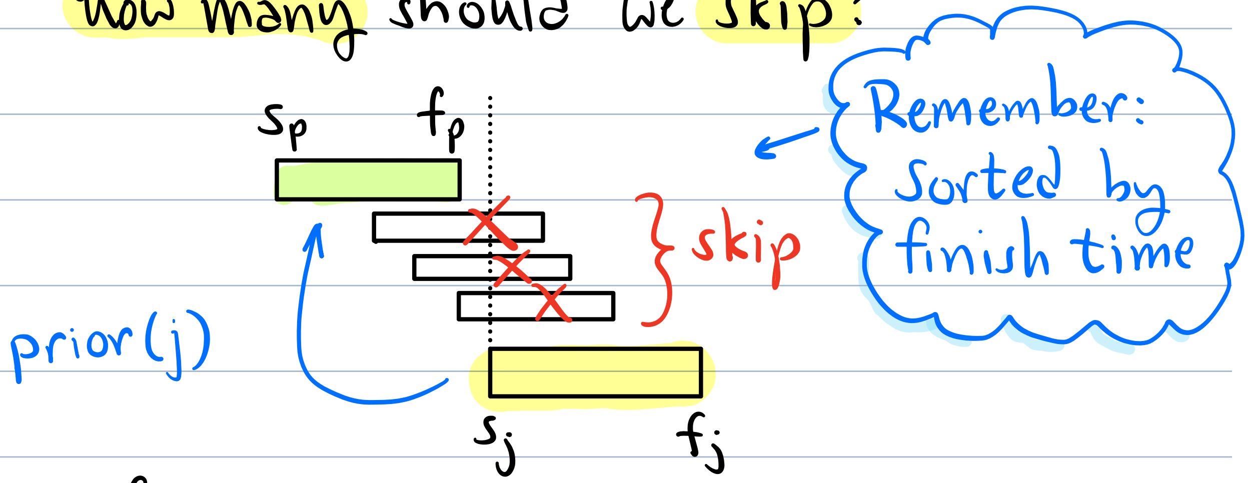


(2)



(2) $[s_n, f_n]$ is in opt schedule
- Add to schedule + \$ v_n
- Skip overlapping intervals $\{p+1, \ldots, n-1\}$
- Recurse on requests 1..p

When we add a request j to schedule, how many should we skip?

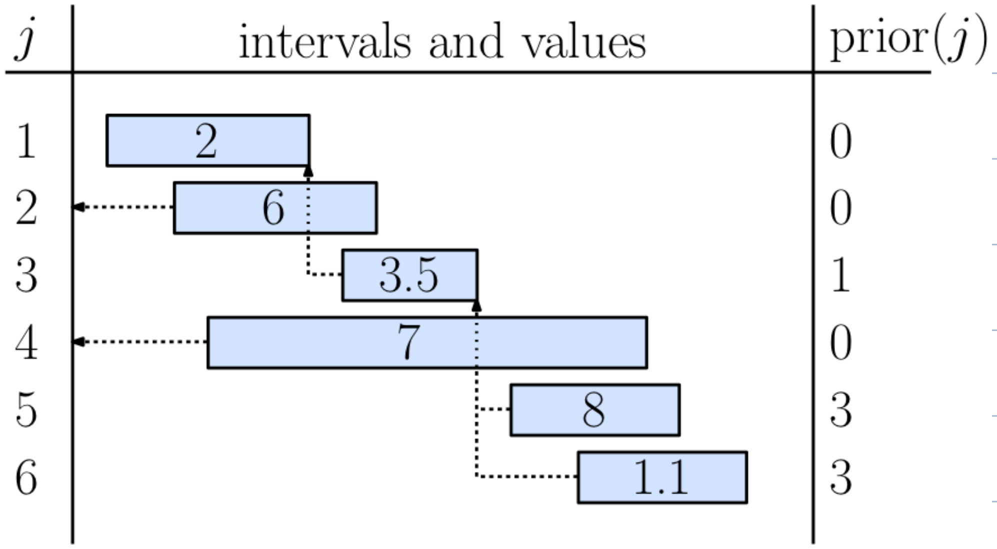


Define: for $1 \le j \le n$

$$\text{prior}(j) = \max_{p} \text{ s.t. } f_p < s_j$$

(or 0 if there is none)

Example:



j	intervals and values	prior(j)
1	2	0
2	6	0
3	3.5	1
4	7	0
5	8	3
6	1.1	3

Optimal Total Value:

for $0 \le j \le n$, $W(j) =$ max value possible for requests $1, 2, \ldots, j$

$$W(j) = \begin{cases} 0 \quad \text{(basis)} & \text{if } j = 0 \\ \max \begin{cases} W(j-1) & \text{(reject)} \\ v_j + W(\text{prior}(j)) & \text{(accept)} \end{cases} & \text{if } j > 0 \end{cases}$$

skips requests that overlap j

Recursive implementation: (+ why this is bad!)

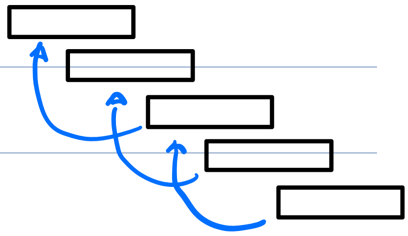
```
WIS(s[1..n], f[1..n], v[1..n])
    Sort requests by finish time
    Compute prior[j] (for 1 ≤ j ≤ n)
    return rec-WIS(n)                      // total value

rec-WIS(j)                                 // value of 1..j
    if (j = 0) return 0                    // basis
    else return max { W(j-1)
                    { v[j] + rec-WIS(prior[j])
```

Too Slow! Why?

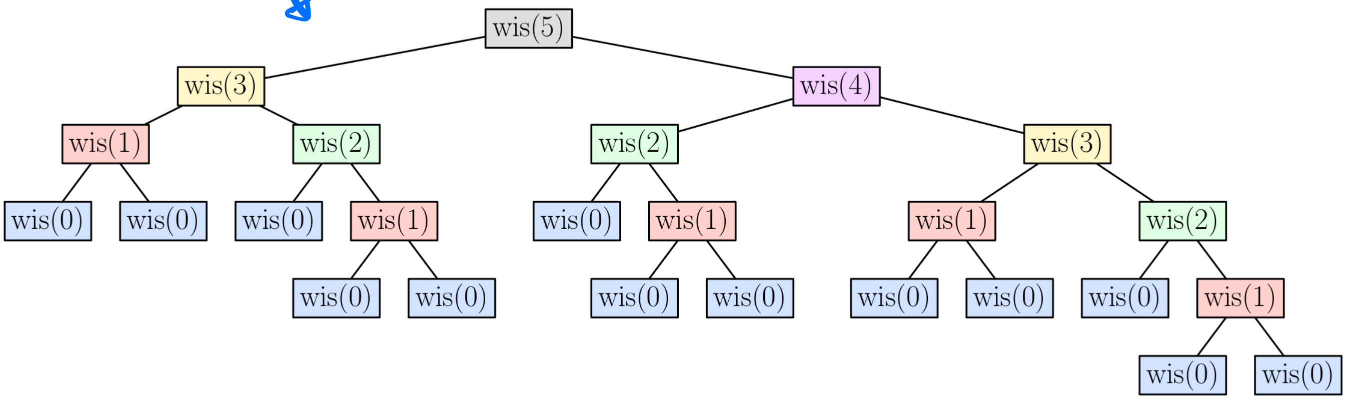
Let $T(j)$ = num. of recursive calls to recWIS(0)
arising from recWIS(j)

Suppose $\text{prior}(j) = j-2$, $\forall j$



rec-WIS(j) calls:

- rec-WIS(j-1)
- rec-Wis(prior(j)) = rec-WIS(j-2)



$$\left.\begin{aligned} &\Rightarrow T(j) = T(j-1) + T(j-2) \\ &+ \; T(0) = 1 \end{aligned}\right\} \text{Fibonacci}$$

Grows fast!

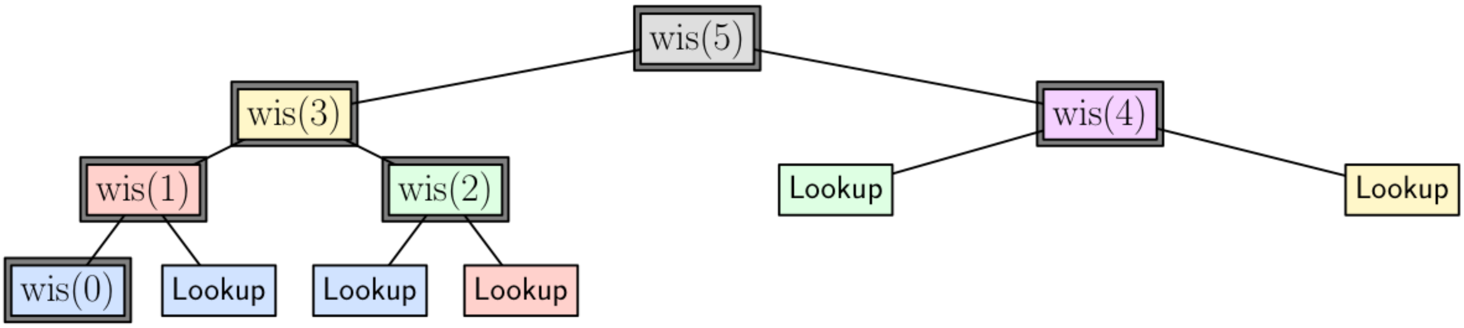
j:	0	1	2	3	4	5...	30	50
$T(j)$:	1	2	3	5	8	13...	2 Meg!	30 Gig!

How do we improve?

Memoization (a.k.a. caching)

Idea: After computing $W(j)$, save it in an array, say $W[j]$.

- Next time we need its value, look it up.
- Results in many fewer recursive calls, $O(n)$



Updated implementation:

- Sort by finish times
- Compute prior[ ] array

} (as before)

- Init: $W[j] = -1$ (means "undefined")
- Array: accept[1..n]

  accept[j] = True - accept request j

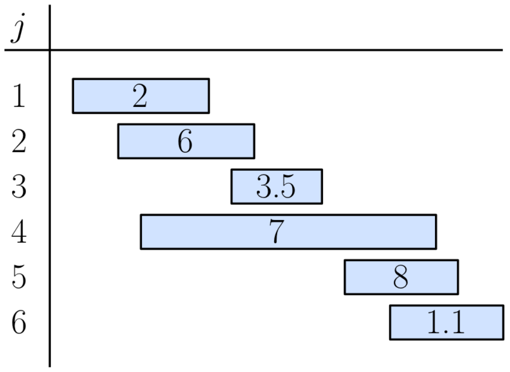
  False - reject " "

We'll use this to construct final schedule (later)

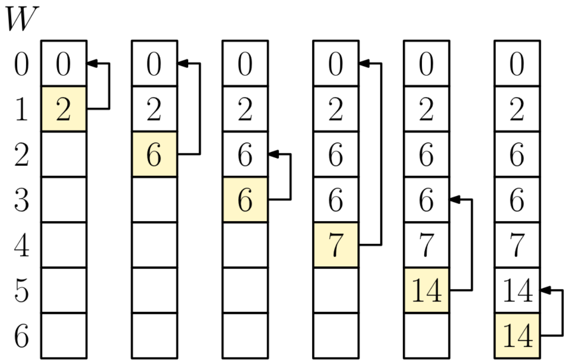
- return memo-WIS(n)

```
memo-WIS(j)                                  // memoized WIS
    if (W[j] = -1)                           // W[j] undefined?
        if (j = 0) W[j] ← 0                  // basis
        else
            rejVal ← memo-WIS(j-1)           // rej/acc values
            accVal ← v[j] + memo-WIS(prior[j])
            if (rejVal > accVal)             // better to reject
                W[j] ← rejVal
                accept[j] ← false
            else                             // better to accept
                W[j] ← accVal
                accept[j] ← true
    return W[j]                              // return value
```

Example: (W-values are created bottom-up)



prior	
1	0
2	0
3	1
4	0
5	3
6	3



accept	
1	T
2	T
3	F
4	T
5	T
6	F

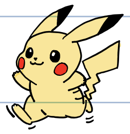
Running Time - $O(n)$ [n+1 rec. calls, each $O(1)$]

Bottom-Up Construction: (Optional)

- In practice it is often more efficient to unravel the recursion, and build W bottom-up
- As before:
  - Sorted by finish times
  - prior[...] computed

```
bottom-up-WIS( )                        // bottom-up implementation
    W[0] ← 0                            // basis
    for (i ← 1 to n)
        rejVal ← W[j-1]                 // rej/acc values
        accVal ← v[j] + W[prior[j]]
        if (rejVal > accVal)            // better to reject
            W[j] ← rejVal
            accept[j] ← false
        else                            // better to accept
            W[j] ← accVal
            accept[j] ← true
    return W[n]                         // final value
```

Running Time - $O(n)$ (obvious)



Computing the Final Schedule:

- So far we only compute the final value, W[n]
- Use the accept[] array to guide us

Start at end (j ← n) + work back (until j == 0)

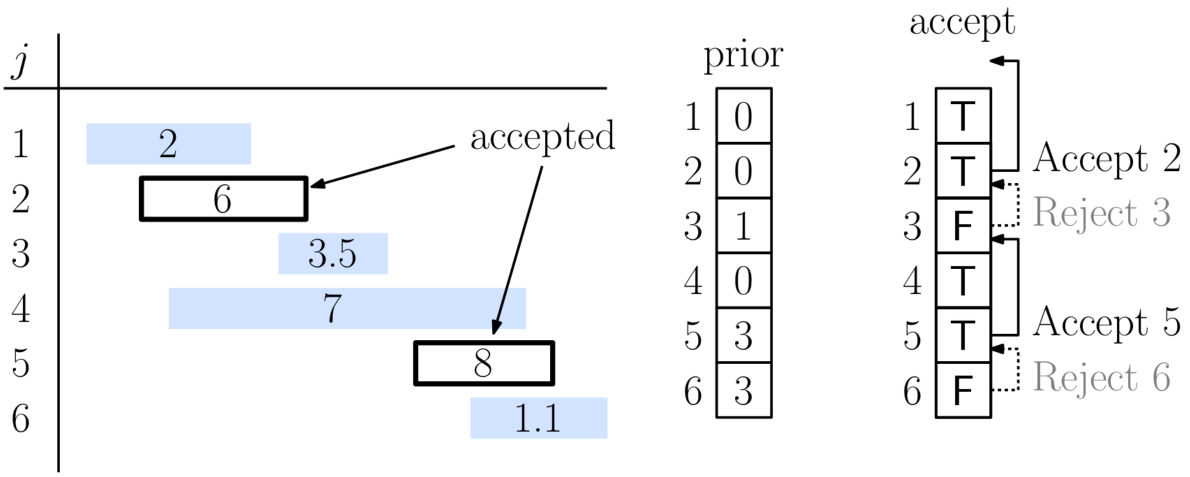
if accept[j] = True:
- add j to schedule
- continue with j ← prior[j]

skip over overlapping requests

else:
- don't add to schedule
- continue with j ← j-1

```
get-schedule()                  // get final schedule
    j ← n                       // start at end
    sched ← ∅                   // init empty sched.
    while (j > 0)
        if (accept[j])          // accepted j
            prepend [j] to sched
            j ← prior[j]
        else                    // rejected j
            j ← j-1
    return sched                // final schedule
```

Example:



j	prior	accept
1	0	T
2	0	T
3	1	F
4	0	T
5	3	T
6	3	F

Trace: $j \leftarrow 6$

accept[6] = F $j \leftarrow 6-1=5$

accept[5] = T   add 5, $j \leftarrow$ prior[5] = 3

accept[3] = F $j \leftarrow 3-1=2$

accept[2] = T   add 2, $j \leftarrow$ prior[2] = 0

$j=0 \rightarrow$ terminate

Note: Even though
accept[4] = T
accept[1] = T
we never visit these

Summary –

- Intro. to dynamic programming
  - Recursive structure (subproblems)
  - Principle of optimality
- Weighted Interval Scheduling
  - Recursive (slow!), memoized, bottom-up