CMSC 451: Dynamic Programming

Slides By: Carl Kingsford

Department of Computer Science
University of Maryland, College Park

Based on Sections 6.1&6.2 of *Algorithm Design* by Kleinberg & Tardos.

Dynamic Programming

Dynamic Programming

- Our 3rd major algorithm design technique

- Similar to divide & conquer
  - Build up the answer from smaller subproblems
  - More general than “simple” divide & conquer
  - Also more powerfulcy

- Generally applies to algorithms where the brute force algorithm would be exponential.

Weighted Interval Scheduling

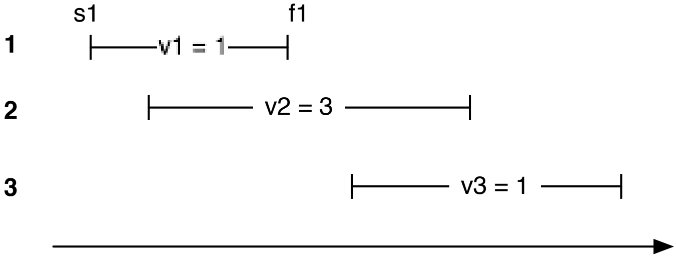
Recall the interval scheduling problem we've seen several times: choose as many non-overlapping intervals as possible.

What if each interval had a value?

Problem (Weighted Interval Scheduling)

Given a set of n intervals (s_i, f_i), each with a value v_i, choose a subset S of non-overlapping intervals with $\sum_{i \in S} v_i$ maximized.

Example



Note that our simple greedy algorithm for the unweighted case doesn't work.

This is becasue some interval can be made very important with a high weight.

Greedy Algorithm For Unweighted Case

Greedy Algorithm For Unweighted Case:

1. Sort by increasing finishing time
2. Repeat until no intervals left:
    1. Choose next interval
    2. Remove all intervals it overlaps with

Just look for the value of the OPT

Suppose for now we're not interested in the actual set of intervals.

Only interested in the *value* of a solution
(aka it's cost, score, objective value).

This is typical of DP algorithms:

- You want to find a solution that optimizes some value.
- You first focus on just computing what that optimal value would be. E.g. what's the highest value of a set of compatible intervals?
- You then post-process your answer (and some tables you've created along the way) to get the actual solution.

Another View

Another way to look at Weighted Interval Scheduling:

Assume that the intervals are sorted by finishing time and represent each interval by its value.

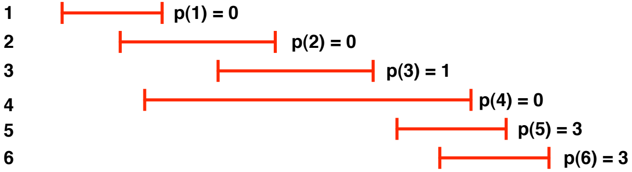
Goal is to choose a subset of the values of maximum sum, so that none of the chosen ($\surd$) intervals overlap:

v_1	v_2	v_3	v_4	$\cdots$	v_{n-1}	v_n
X	$\surd$	X	$\surd$		$\surd$	X

Notation

Definition

$p(j)$ = the largest $i < j$ such that interval i doesn't overlap with j.

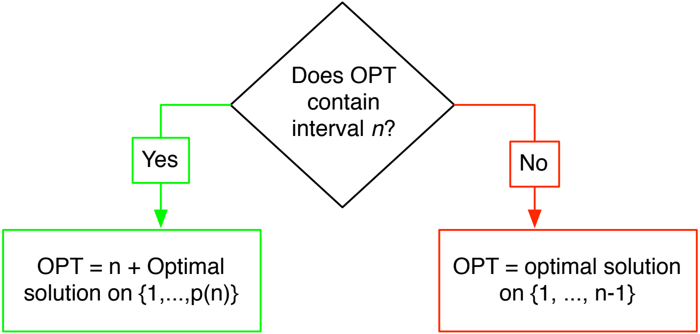


$p(j)$ is the interval farthest to the right that is compatible with j.

What does an OPT solution look like?

Let OPT be an optimal solution.

Let *n* be the last interval.



Generalize

Definition

OPT(j) = the optimal solution considering only intervals $1, \ldots, j$

$$OPT(j) = \max \begin{cases} v_j + OPT(p(j)) & j \text{ in OPT solution} \\ OPT(j-1) & j \text{ not in solution} \\ 0 & j = 0 \end{cases}$$

This kind of recurrence relation is very typical of dynamic programming.

Slow Implementation

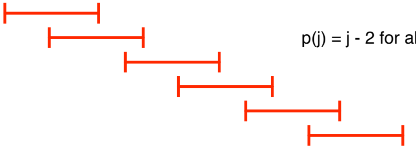
Implementing the recurrence directly:

```
WeightedIntSched(j):
    If j = 0:
        Return 0
    Else:
         Return  max(
             v[j] + WeightedIntSched(p[j]),
             WeightedIntSched(j-1)
         )
```

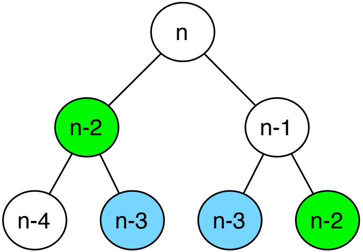
Unfortunately, this is exponential time!

Why is this exponential time?

Consider this set of intervals:



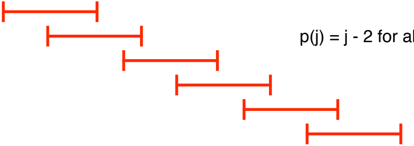
p(j) = j - 2 for all j ≥ 3



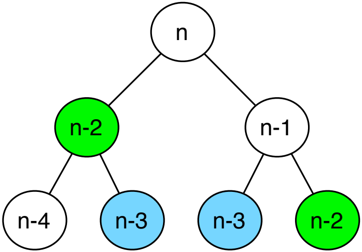
- What's the shortest path from the root to a leaf?
- Total # nodes is $\geq 2^{n/2}$
- Each node does constant work $\implies \Omega(2^n)$

Why is this exponential time?

Consider this set of intervals:



p(j) = j - 2 for all j ≥ 3



- What's the shortest path from the root to a leaf? $n/2$
- Total # nodes is $\geq 2^{n/2}$
- Each node does constant work $\implies \Omega(2^n)$

Memoize

Problem: Repeatedly solving the same subproblem.

Solution: Save the answer for each subproblem as you compute it.

When you compute $OPT(j)$, save the value in a global array M.

Memoize Code

```
MemoizedIntSched(j):
    If j = 0: Return 0
    Else If M[j] is not empty:
        Return M[j]
    Else
       M[j] = max(
                v[j] + MemoizedIntSched(p[j]),
                MemoizedIntSched(j-1)
              )
      Return M[j]
```

- Fill in 1 array entry for every two calls to `MemoizedIntSched`.
  $\implies O(n)$

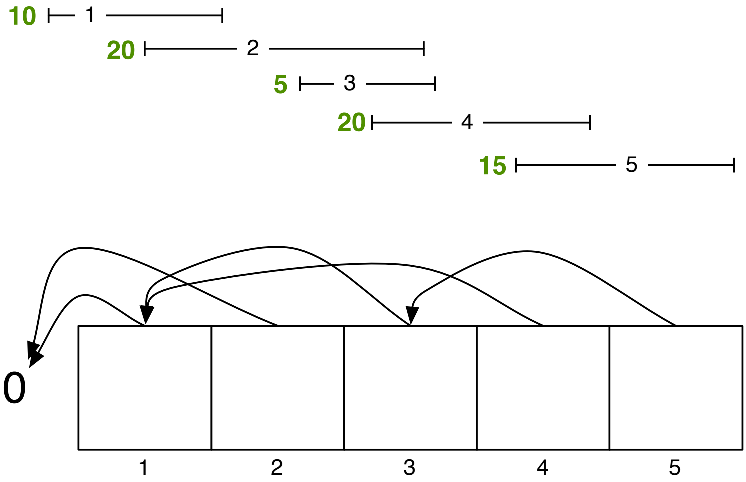
Easier Algorithm

When we compute $M[j]$, we only need values for $M[k]$ for $k < j$:

```
ForwardIntSched(j):
   M[0] = 0
   for j = 1, ..., n:
       M[j] = max(v[j] + M[p(j)], M[j-1])
```

Main Idea of Dynamic Programming: solve the subproblems in an order that makes sure when you need an answer, it's already been computed.

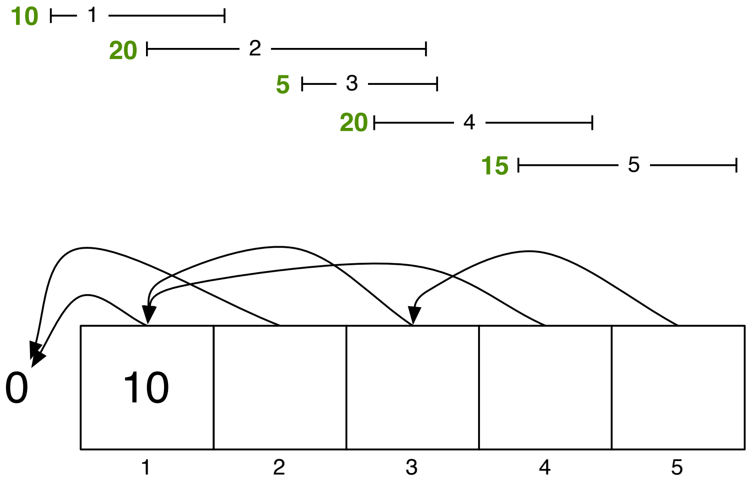
Example



$v_j + M[p(j)]$

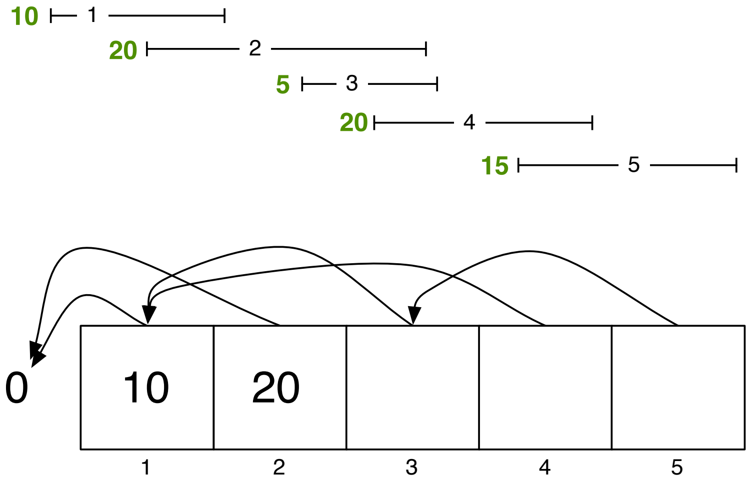
$M[j-1]$

Example



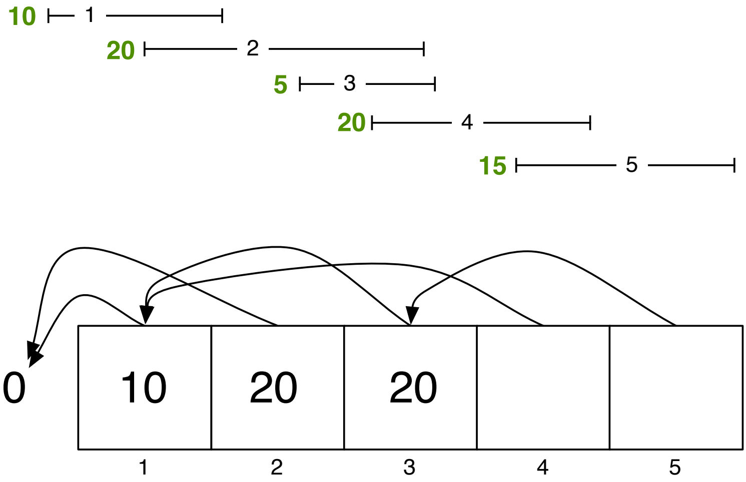
v_j + M[p(j)]	10
M[j-1]	0

Example



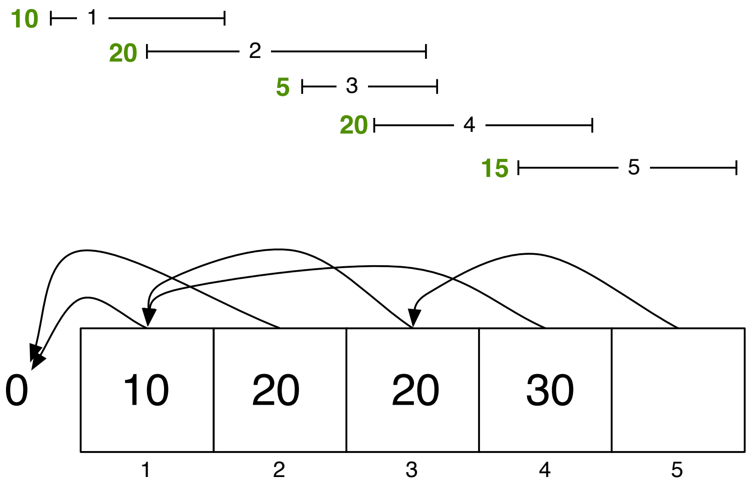
	1	2
v_j + M[p(j)]	10	20
M[j-1]	0	10

Example



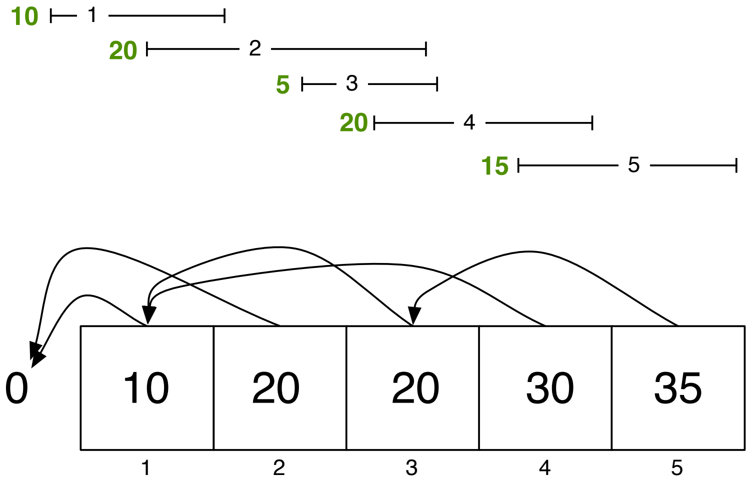
v_j + M[p(j)]	10	20	15
M[j-1]	0	10	20

Example



v_j + M[p(j)]	10	20	15	30
M[j-1]	0	10	20	20

Example



	1	2	3	4	5
$v_j + M[p(j)]$	10	20	15	30	35
M[j-1]	0	10	20	20	30

General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems.

2. There are only a polynomial # of subproblems.

3. There is a "natural" ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.

General DP Principles

1. Optimal value of the original problem can be computed easily from some subproblems. OPT(j) = max of two subproblems

2. There are only a polynomial # of subproblems. $\{1, \ldots, j\}$ for $j = 1, \ldots, n$.

3. There is a "natural" ordering of the subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems. $\{1, 2, 3\}$ is smaller than $\{1, 2, 3, 4\}$

Getting the actual solution

We now have an algorithm to find the *value* of OPT. How do we get the actual choices of intervals?

Interval j is in the optimal solution for the subproblem on intervals $\{1, \ldots, j\}$ only if

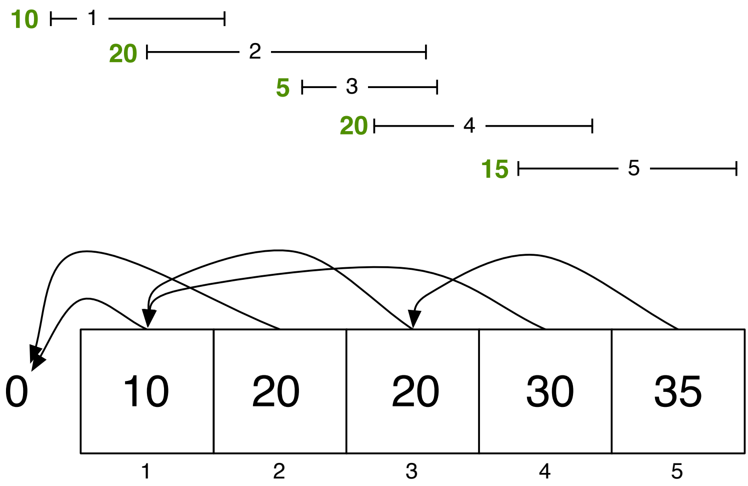
$$v_j + OPT(p(j)) \geq OPT(j-1)$$

So, interval n is in the optimal solution only if

$$v[n] + M[p[n]] \geq M[n-1]$$

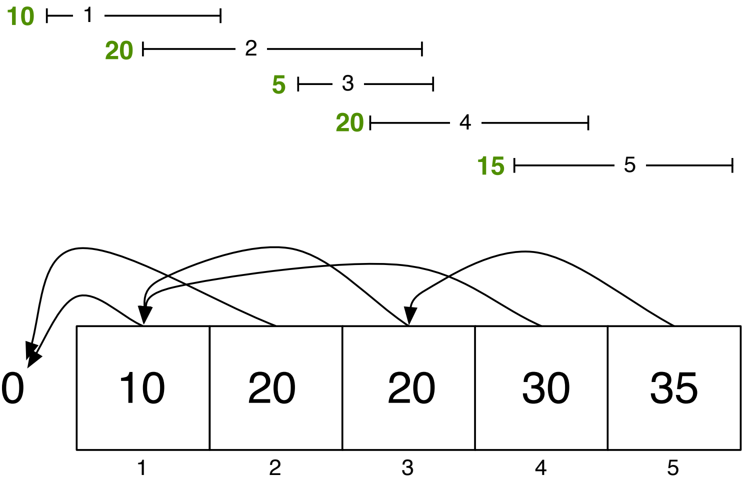
After deciding if n is in the solution, we can look at the relevant subproblem: either $\{1, \ldots, p(n)\}$ or $\{1, \ldots, n-1\}$.

Example



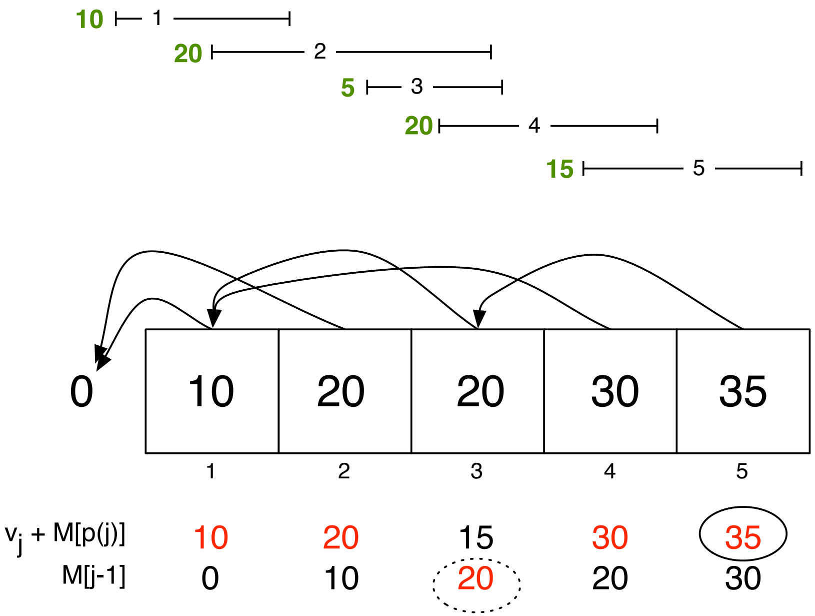
$v_j + M[p(j)]$	10	20	15	30	35
$M[j-1]$	0	10	20	20	30

Example

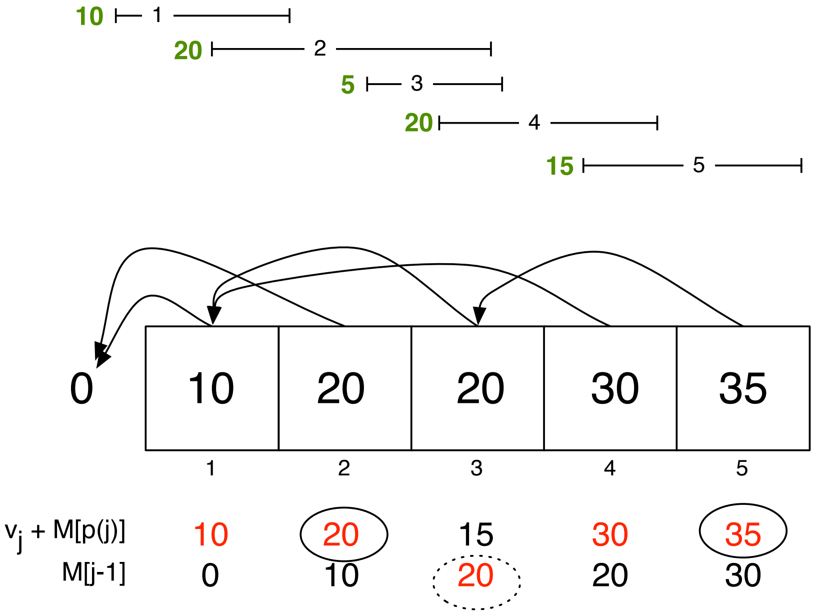


	1	2	3	4	5
v_j + M[p(j)]	10	20	15	30	35
M[j-1]	0	10	20	20	30

Example


10
1
20
2
5
3
20
4
15
5
0
10
20
20
30
35
1
2
3
4
5
v_j + M[p(j)]
10
20
15
30
35
M[j-1]
0
10
20
20
30

Example



10
1
20
2
5
3
20
4
15
5
0
10
20
20
30
35
1
2
3
4
5
v_j + M[p(j)]
10
20
15
30
35
M[j-1]
0
10
20
20
30

Code

```
BacktrackForSolution(M, j):
  If j > 0:
      If v[j] + M[p[j]] ≥ M[j-1]:    // find the winner
         Output j                    // j is in the soln
         BacktrackForSolution(M, p[j])
      Else:
         BacktrackForSolution(M, j-1)
      EndIf
  EndIf
```

Running Time

Time to sort by finishing time: $O(n \log n)$

Time to compute $p(n)$: $O(n^2)$

Time to fill in the `M` array: $O(n)$

Time to backtrack to find solution: $O(n)$