

Program Structure and Algorithms

Sid Nath

Lecture 12

Agenda

- Administrative
 - HW5, PA2 will be out after today's class
- Lecture
- Quiz

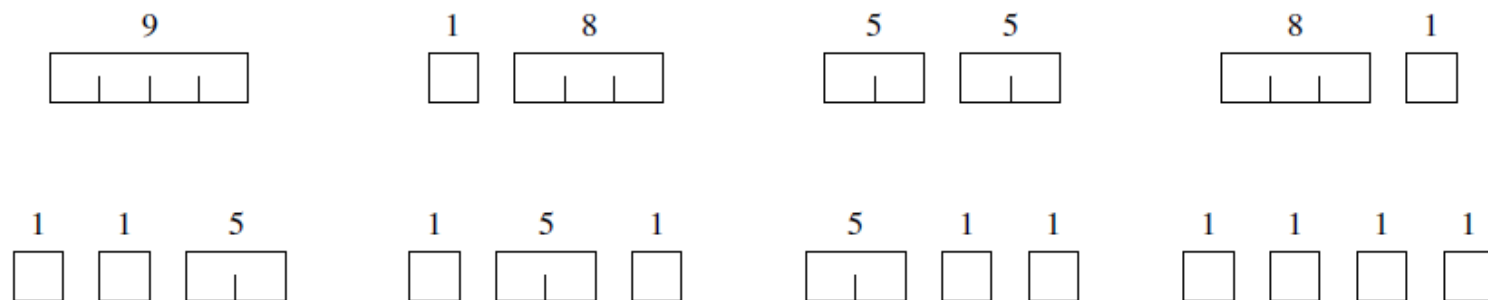
P5: Rod Cutting

- How to cut steel rods into pieces in order to maximize the revenue you can get? Each cut is free. Rod lengths are always an integer number of inches.
- Input: A length n (inches) and table of prices $p_i, i = 1, 2, \dots, n$
- Output: Maximum revenue obtainable from rods whose lengths sum to n , computed as the sum of the prices for the individual rods
- Trivial solution: If p_n is very large, optimal solution needs no cuts, i.e., leave the rod as n inches long

Example

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

- After first $n - 1$ inches, we can either cut or not ..
 2^{n-1} ways
- 8 ways to cut a rod of length 4
 - best two 2 inches that yields a revenue of $5+5=10$



Formulation

Let r_i be the max revenue for a rod of length i

By inspection

For $n = 7$, one opt sol makes a cut at 3in \rightarrow 2 subproblems of lengths 3 and 4 \rightarrow solve both optimally.

The opt sol of length 4 $\rightarrow 2 + 2 \rightarrow$ used in opt sol of length 7

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2 + 2
5	13	2 + 3
6	17	6 (no cuts)
7	18	1 + 6 or 2 + 2 + 3
8	22	2 + 6

Optimal revenue r_n by taking max of

- p_n : revenue from not making a cut
- $r_1 + r_{n-1}$: max revenue from a rod 1in and rod $(n - 1)$ in
- $r_2 + r_{n-2}$: max revenue from a rod 2in and rod $(n - 2)$ in, ...
- $r_{n-1} + r_1$

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1\}$$

Formulation

Intuition: Every opt sol has a leftmost cut, i.e., there's some cut that gives a first piece of length i cut off the left end, and a remaining piece of length $n - i$ on the right

Need to divide only the remainder, not the first piece

Sol with no cuts has first piece $i = n$ with revenue p_n and remaining size of length 0 has revenue $r_0 = 0$ // base case

$$r_n = \max\{p_i + r_{n-i}, \forall i = 1 \dots n\}$$

Subproblem: Max revenue achievable by cutting a rod of length i off the left end

If you think of the rod demarcated at intervals of unit length, then entire rod is a sequence $x[1:n]$

Then, subproblem will be max revenue achievable by cutting $x[1:i]$

Naïve Recursive Top-Down Solution

What's the problem?

CUT-ROD calls itself repeatedly on solved subproblems!!!

Let's study for $n = 4$

Many repeated subproblems

Solves for subproblem for size 2 twice, for size 1 four times, for size 0 eight times!!!

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j), & \text{if } n \geq 1 \end{cases} = O(2^n)$$

CUT-ROD(p, n)

if $n == 0$

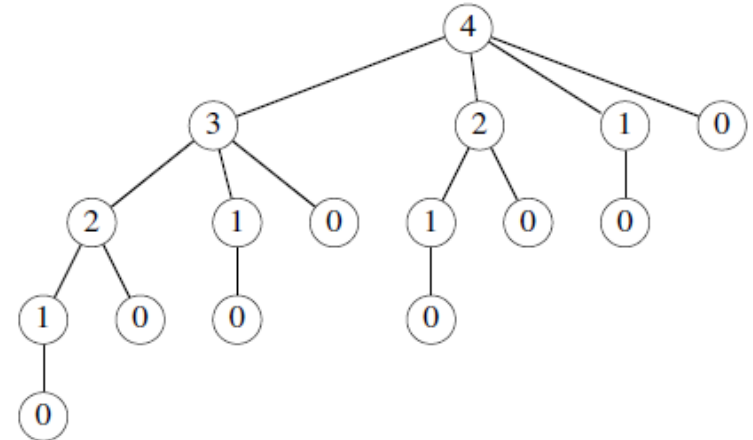
return 0

$q = -\infty$

for $i = 1$ to n

$q = \max \{q, p[i] + \text{CUT-ROD}(p, n - i)\}$

return q



Subproblem Graphs

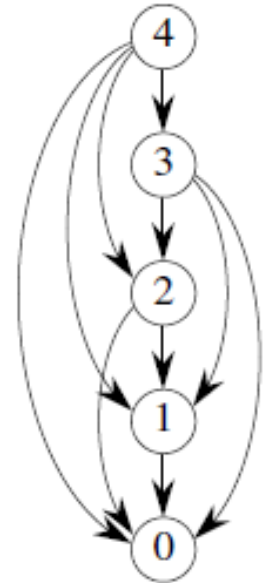
How to understand the dependency of subproblems? Directed graph

One vertex for each distinct subproblem

Directed edge (x, y) if computing an opt sol to subproblem x directly requires knowing an opt sol to subproblem y

Running time: Sum of times needed to solve each subproblem

Time to compute a subproblem is typically linear in the out-degree of its vertex



DP Solution

Arrange to solve each subproblem just once by saving solution in a table

Pick solution from table when revisiting solved subproblems

Turns exponential-time solution to a polynomial-time solution

Two basic approaches:

Top-down with memoization

Bottom-up

Top-Down with Memoization

Memorize what has been computed previously

Store solution to subproblem of length i in an array entry $r[i]$

MEMOIZED-CUT-ROD(p, n)

```
    let  $r[0:n]$  be a new array      // will remember solution values in  $r$ 
    for  $i = 0$  to  $n$ 
         $r[i] = -\infty$ 
    return MEMOIZED-CUT-ROD-AUX( $p, n, r$ )
```

MEMOIZED-CUT-ROD-AUX(p, n, r)

```
    if  $r[n] \geq 0$                 // already have a solution for length  $n$ ?
        return  $r[n]$ 
    if  $n == 0$ 
         $q = 0$ 
    else  $q = -\infty$ 
        for  $i = 1$  to  $n$     //  $i$  is the position of the first cut
             $q = \max \{q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r)\}$ 
     $r[n] = q$                 // remember the solution value for length  $n$ 
    return  $q$ 
```

Bottom-Up

Sort the subproblems by size and solve the smaller ones first

BOTTOM-UP-CUT-ROD(p, n)

```
let  $r[0:n]$  be a new array      // will remember solution values in  $r$ 
 $r[0] = 0$ 
for  $j = 1$  to  $n$                 // for increasing rod length  $j$ 
     $q = -\infty$ 
    for  $i = 1$  to  $j$             //  $i$  is the position of the first cut
         $q = \max \{q, p[i] + r[j - i]\}$ 
     $r[j] = q$                   // remember the solution value for length  $j$ 
return  $r[n]$ 
```

How to Reconstruct a Solution?

Extend bottom-up approach to output opt values AND opt choices

Save opt choices in a separate table and print it

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

let $r[0:n]$ and $s[1:n]$ be new arrays

$r[0] = 0$

for $j = 1$ **to** n // for increasing rod length j

$q = -\infty$

for $i = 1$ **to** j // i is the position of the first cut

if $q < p[i] + r[j - i]$

$q = p[i] + r[j - i]$

$s[j] = i$ // best cut location so far for length j

$r[j] = q$ // remember the solution value for length j

return r and s

PRINT-CUT-ROD-SOLUTION(p, n)

$(r, s) = \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)$

while $n > 0$

 print $s[n]$ // cut location for length n

$n = n - s[n]$ // length of the remainder of the rod

Example of Output

Example: PRINT-CUT-ROD-SOLUTION(p , 8)

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

Example of Output

Example: PRINT-CUT-ROD-SOLUTION(p , 8)

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$		1	2	3	2	2	6	1	2

Runtime and Space Complexity

Top-down: To solve a subproblem of size n , the for loop iterates n times \rightarrow total #iters form an arithmetic series

Bottom-up: #Iters of inner for loop forms an arithmetic series

Running time: $\theta(n^2)$

Space complexity: $\theta(n)$

Chain of Thought

- What are my subproblems?
- What are the decisions to solve each subproblem?
- Recursive formulation
 - Base case
- How many subproblems? What is the running time per subproblem?
- What is the overall running time?

P6: Knapsack w/ Repetition

Given a “knapsack” that can hold a maximum weight of W pounds, a robber has n items to pick from. Each item has weight $\{w_1, w_2, \dots, w_n\}$ and dollar value $\{v_1, v_2, \dots, v_n\}$.

What’s the most valuable combination of items the robber can fit into his bag / knapsack?

Generalizes to a wide variety of resource-constrained selection problems!

Example

Item	Weight (lbs)	Value (\$)
1	7	12
2	3	2
3	20	41

W = 58 lbs

Option 1: $2 \times \text{Item3} + 1 \times \text{Item2} + 2 \times \text{Item1}$

Weight: $40 + 3 + 14 = 57 \text{ lbs} \leq 58 \text{ lbs}$

Value: $82 + 2 + 24 = \$108$

Option 2: $2 \times \text{Item3} + 6 \times \text{Item2} + 0 \times \text{Item1}$

Weight: $40 + 18 + 0 = 58 \text{ lbs} \leq 58 \text{ lbs}$

Value: $82 + 12 + 0 = \$94$

A Brute-Force Approach

- Try all possible combinations of items
- Compute weight and profits; eliminate combinations with total weight $> W$
- Find max value

Runtime: $\sum_{k=0}^n C(n, k) = \Omega(2^n)$

DP: Structure of an optimal solution

Idea: Look at smaller knapsack capacities $w \leq W$

Let $K(w)$ be the max value achievable with a knapsack capacity $w \leq W$.

We want $K(W)$.

If the optimal solution includes item i ,

- removing this item leaves an optimal solution to $K(w - w_i)$
- Add value of item i , which is v_i

Need to consider all possible values of i

DP: A Recursive Solution

$$K(w) = \max\{K(w - w_i) + v_i\} \forall i: w_i \leq w$$

Base case: $K(0) = 0$

DP: Pseudocode (Tabular, bottom-up)

$K(0) = 0$

for $w = 1$ to W :

$K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$

return $K(W)$

$O(W)$ subproblems

Each entry can take up to $O(n)$ time to compute

Runtime: $O(nW)$

If $W \sim 2^8$ or 2^{20} , i.e., 8bits to 20bits, complexity increases exponentially with #bits

P7: Knapsack w/o Repetition

If repetitions are not allowed to our knapsack problem. How can we solve it using DP?

$K(j, w)$ = max value achievable using knapsack of capacity w and items $1, \dots, j$

We need $K(n, W)$

$$K(j, w) = \max\{K(j - 1, w - w_j) + v_j, K(j - 1, w)\}$$

P5: Example

Item	1	2	3	4	5
W (lbs)	1	2	1	3	5
V (\$)	10	40	20	20	60

$W = 10 \text{ lbs}$

$j = 1, w = 1, w_j=1, v_j=10$

$K[1, 1] = \max(K[0,0]+10, K[0,1]) = 10$

$K[1,w] = \max(K[0, w-w_1]+10, K[0,w]); K[0, w] = 0; K[0, w-w_1] = 0 \forall 2 \leq w \leq 10$

$j = 2, w = 1, w_j=2, v_i=40; w_j > w, \text{ so } K[2,1] = K[1,1]$

$j = 2, w = 2, w_j=2, v_i=40; w_j \leq w, \text{ so } K[2,2] = \max(K[1,0]+40, K[1,2]) = \max(40,10) = 40$

$j = 2, w = 3, w_j=2, v_i=40; w_j \leq w, \text{ so } K[2,3] = \max(K[1,1]+40, K[1,3]) = \max(10+40,10) = 50$

$K[2,w] = \max(K[1, w-w_2]+40, K[1,w]) = 50$

weight capacity →			0	1	2	3	4	5	6	7	8	9	10
weights	values	0	0	0	0	0	0	0	0	0	0	0	0
1	10	1	0	10	10	10	10	10	10	10	10	10	10
2	40	2	0	10	40	50	50	50	50	50	50	50	50
1	20	3	0	20	40	60	70	70	70	70	70	70	70
3	20	4	0	20	40	60	70	70	80	90	90	90	90
5	60	5	0	20	40	60	70	70	80	100	120	130	130

$K[n,W] = K[5,10] = 130$
 $= \text{max knapsack value}$

P8: Weighted Interval Scheduling

Given a set of n jobs, each job $j \in n$ starts at s_j and finishes at f_j and has a weight or value v_j . Two jobs are compatible if they don't overlap.

Goal: Find maximum weight subset of mutually compatible jobs

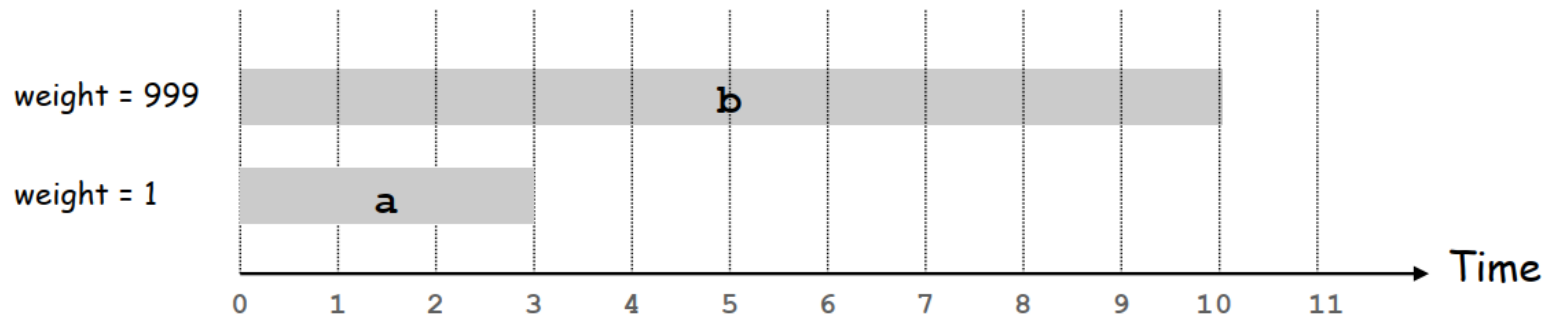
Weighted Interval Scheduling

Given a set of n jobs, each job $j \in n$ starts at s_j and finishes at f_j and has a weight or value v_j . Two jobs are compatible if they don't overlap.

Goal: Find maximum weight subset of mutually compatible jobs

When does greedy work? $v_j = 1, \forall j = 1, 2, \dots, n$

-- When all weights are 1 (or, the same)



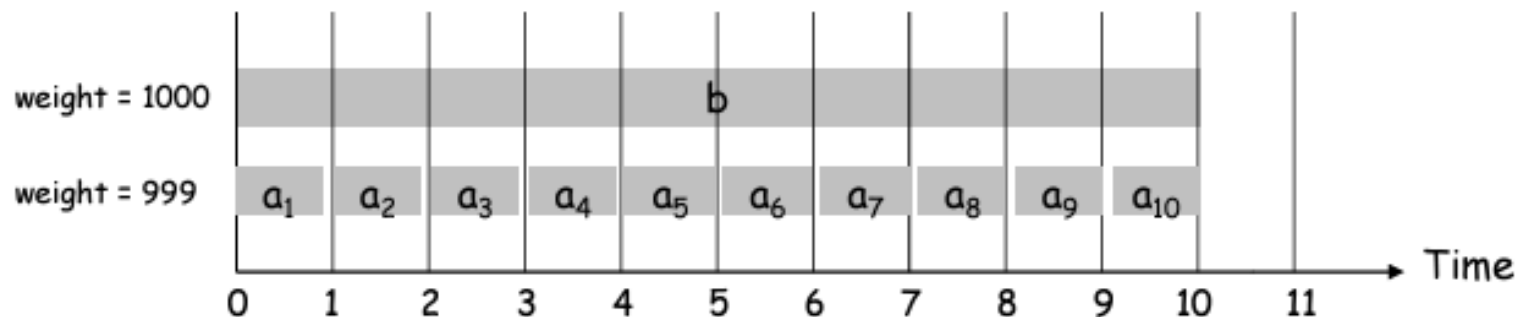
Weighted Interval Scheduling

Given a set of n jobs, each job $j \in n$ starts at s_j and finishes at f_j and has a weight or value v_j . Two jobs are compatible if they don't overlap.

Goal: Find maximum weight subset of mutually compatible jobs

When does greedy work? $v_j = 1, \forall j = 1, 2, \dots, n$

- When all weights are 1 (or, the same)
- Sort by weight?



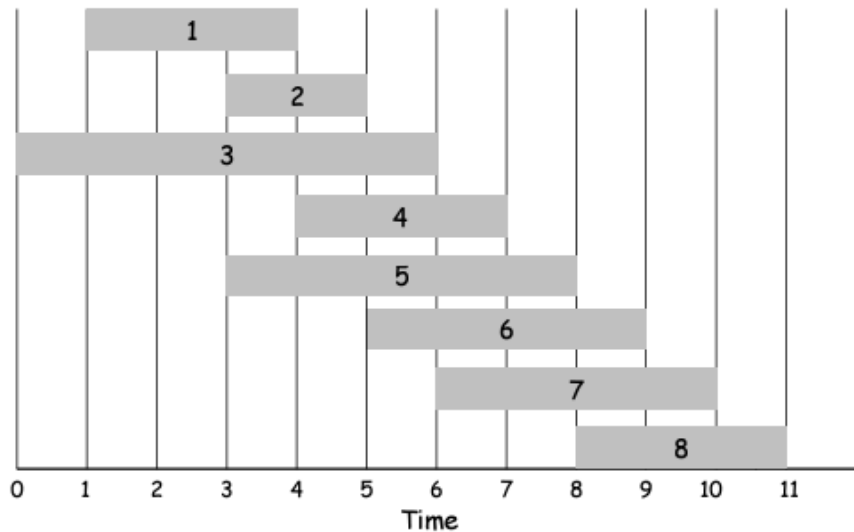
Preprocessing Step (Something new!)

Idea:

Sort by finish times: $f_1 \leq f_2 \leq \dots \leq f_n$.

Maintain a table $p(j) = i$, where i is the largest index $i < j$ s.t. job i and j are m.c.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



j	p(j)
0	-
1	0
2	0
3	0
4	1
5	0
6	2
7	3
8	5

DP Approach

Subproblems: $WIS(j)$ = max weight subset of jobs start at 1 and ending at j

Decisions:

- Include j : add profit v_j , include optimal solution to previous set of m.c. jobs, i.e., 1, ..., $p(j)$
- Do not include j : include optimal solution to previous jobs, 1, ..., $j-1$

$$WIS(j) = \begin{cases} 0, & \text{if } j = 0 \\ \max\{v_j + WIS(p(j)), WIS(j - 1)\}, & \text{otherwise} \end{cases}$$

Running time:

$O(n \log n)$ to sort. $O(n)$ to compute $WIS(j) \forall j = 1, \dots, n$

P9: Share Trading

We're given the price of a stock over n consecutive days $i = 1, 2, \dots, n$. For each day i , we're given a price $p(i)$ per share for the stock on that day. (Assume that the price is fixed per day.)

How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it so that we maximize the profit per share, $p(j) - p(i)$? If there is no way to make money during the n days, we should conclude this instead.

Share Trading

Subproblem: Let $OPT(j)$ ($j = 1, \dots, n$) = max possible return if investor sells share on day j .. We want $OPT(n)$

Decisions: On day j , investor either holding it on day $j - 1$ or weren't.

- If not, $OPT(j) = 0$.
- If yes, then $OPT(j) = OPT(j - 1) + (p(j) - p(j - 1))$.

Recursion:

$$OPT_j = \max\{ OPT(j) + (p(j) - p(j - 1)), 0 \}$$

Base case: $OPT(1) = 0$

Running time: $O(n)$ subproblems, $O(1)$ per subproblem, so $O(n)$

Problems So Far ...

Prob #	Definition	Opt Type	Template	Running Time (only DP part)
1	Shortest paths in DAGs	Min	#1	$O(V + E) \times O(1)$
2	Bellman Ford	Min	#1	$O(V) \times O(E)$
3	Floyd Warshall	Min	#2	$O(n^2) \times O(n)$
4	Transitive Closure of Graph	Min	#2	$O(n^2) \times O(n)$
5	Rod Cutting	Max	#1	$O(n) \times O(n)$
6	Knapsack w/ repetition	Max	#1	$O(W) \times O(n)$
7	Knapsack w/o repetition	Max	#1	$O(W) \times O(n)$
8	Weighted Interval Scheduling	Max	#1	$O(n) \times O(1)$
9	Share trading	Max	#1	$O(n) \times O(1)$

Lecture 12 summary

- Rod cutting
- Knapsack w/ repetition
- Knapsack w/ repetition
- Weighted Interval Scheduling
- Share trading