Lecture 11: Dynamic Programming

Version of Feb 26, 2019

Outline

1. Introduction to Dynamic Programming

2. The Rod-Cutting Problem

3. Weighted Interval Scheduling

4. Review & Summary

Dynamic Programming (DP) bears similarities to Divide and Conquer (D&C)

- Both partition a problem into smaller subproblems
  => build solution of larger problems from solutions of smaller problems

- In D&C, work top-down.
  Solve **exact smaller problems** that need to be solved to solve larger problem
- In DP, (usually) work bottom-up.
- Solve **all smaller size problems** => build larger problem solutions from them.
  - many large subproblems reuse solution to **same** smaller problem.

- DP often used for optimization problems
- Problems have many ***feasible solutions***; we want the ***bes***t solution.

Main idea of DP

1. Analyze the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution (usually bottom-up)

First Example: Stairs Climbing

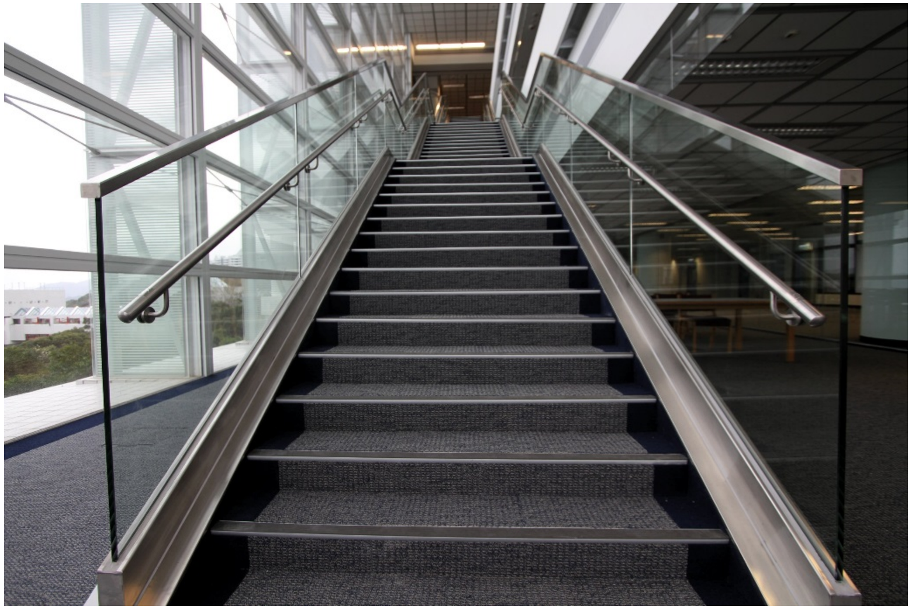
Problem: You can climb 1 or 2 stairs with one step.
How many *different* ways can you climb n stairs?

Solution: Let $\mathrm{F}(n)$ be the number of different ways to climb n stairs.

$$F(1) = 1, F(2) = 2, F(3) = 3, \ldots$$

$$F(n) = F(n-1) + F(n-2)$$

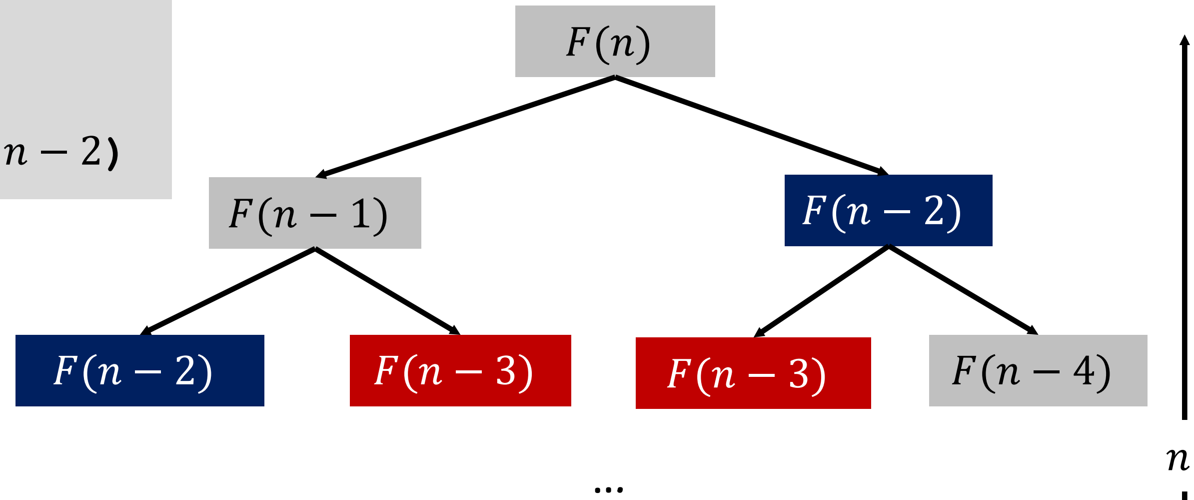
Q: How to compute $\mathrm{F}(n)$?



Solving the recurrence by recursion

$$F(1) = 1, \qquad F(2) = 2$$
$$F(n) = F(n-1) + F(n-2)$$

```
F(n):
if n = 1 return 1
if n = 2 return 2
return F(n − 1)+F(n − 2)
```



Running time?

Between $2^{n/2}$ and 2^n.

A more deeper analysis yields $\Theta(\varphi^n)$ where $\varphi \approx 1.618$ is the golden ratio.

Q: Why so slow?

A: Solving the same subproblem many many times.

Solving the recurrence by dynamic programming

$$F(1) = 1, \qquad F(2) = 2$$
$$F(n) = F(n-1) + F(n-2)$$

```
F(n):
allocate an array A of size n
A[1] ← 1
A[2] ← 2
for i = 3 to n
      A[i] ← A[i-1] + A[i-2]
return A[n]
```

Running time: $\Theta(n)$

Space: $\Theta(n)$ but can be improved to $\Theta(1)$ by freeing array entries that are no longer needed.

Dynamic programming:

- Used to solve recurrences
- Avoid solving a subproblem more than once by remembering solution to old problems
- Usually done "bottom-up", filling in subproblem solutions in table in order from "smallest" to largest".
  - There is also "top-down" version (memoization) that we will not be discussing. Essentially equivalent
- "Programming" here means "planning", not coding!

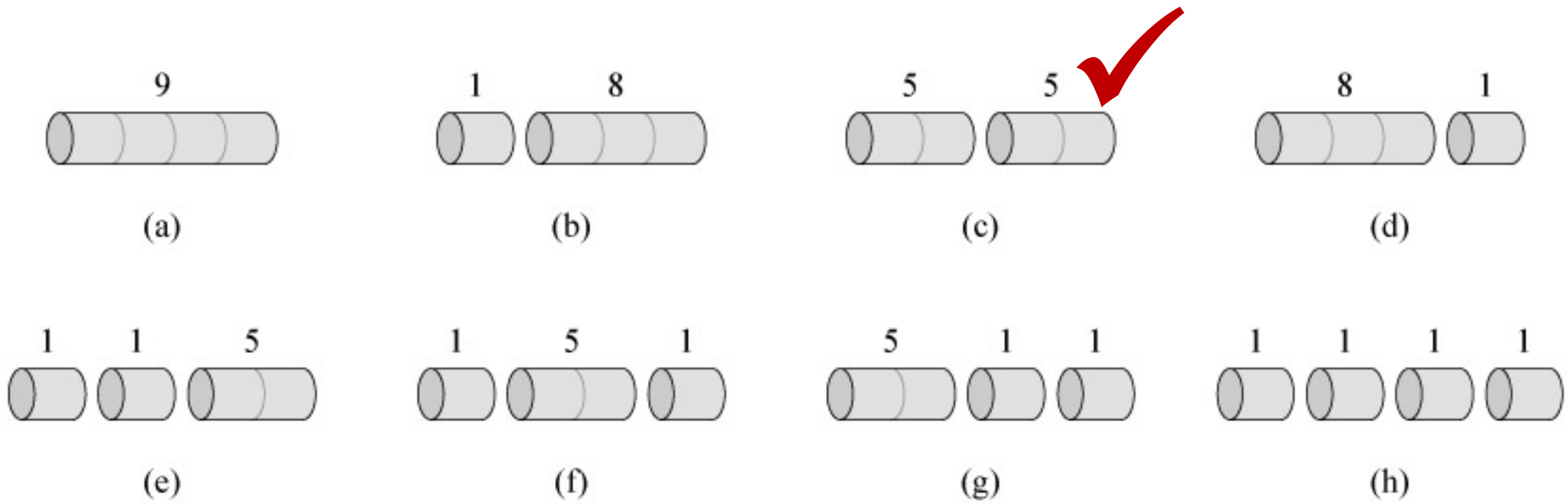
Outline

1. Introduction to Dynamic Programming
2. The Rod-Cutting Problem
3. Weighted Interval Scheduling
4. Review & Summary

The Rod Cutting Problem

Problem: Given a rod of length n and prices p_i for $i = 1, \ldots, n$, where p_i is the price of a rod of length i. Find a way to cut the rod to maximize total revenue.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



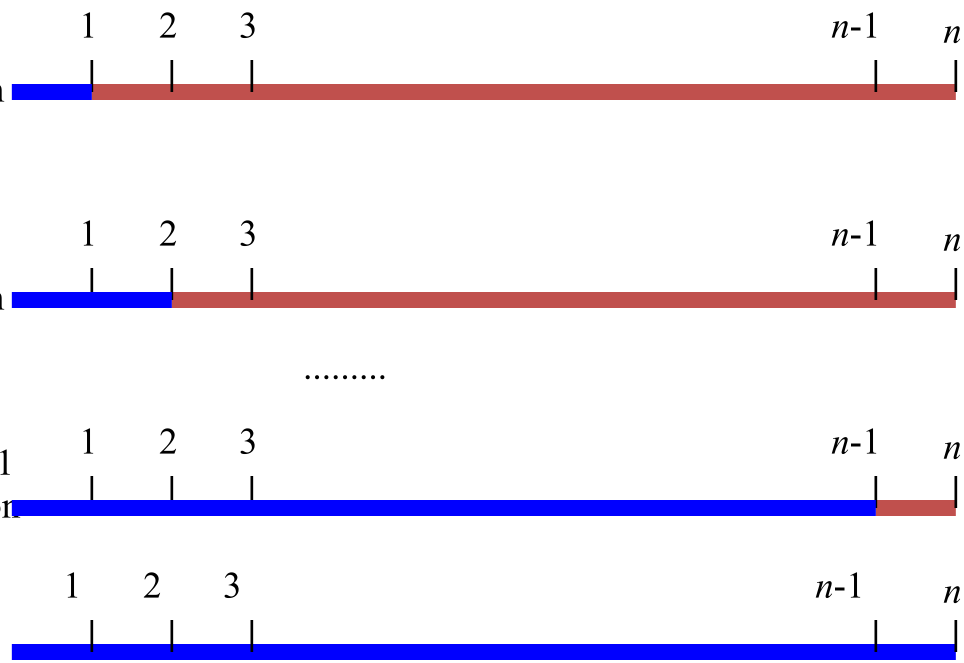
Want to calculate the maximum revenue r_n that can be achieved by cutting a rod of size n. Will do this by finding a way to calculate r_n from $r_1, r_2, \ldots, r_{n-1}$

There are 2^{n-1} ways of cutting rod of size n . Too many to check all of them separately.

Visualization of Optimal Substructure

Initial rod

Choices



- Cut a piece of length 1
- Find the optimal division for length n-1

Revenue p_1+r_{n-1}

- Cut a piece of length 2
- Find the optimal division for length n-2

Revenue p_2+r_{n-2}

....

- Cut a piece of length n-1
- Find the optimal division for length 1

Revenue $p_{n-1}+r_1$
$=p_{n-1}+p_1$

Sell in one piece

Revenue p_n

The best choice is the maximum of p_1+r_{n-1}, p_2+r_{n-2} , ..., $p_{n-1}+r_1$, p_n

Rod Cutting: Another View

Define: Let r_n be the maximum revenue obtainable from cutting a rod of length n.

Consider an optimal (revenue maximizing) cutting.

- Suppose the "first" cut created a piece of length j (with revenue p_j)
- that leaves a piece of length n-j.
  - The max revenue from that piece is r_{n-j}
- Total max revenue from cutting with first piece length j is $p_j + r_{n-j}$
- Try out every possible first cutting and calculate max revenue for each
  - Largest of those is max possible revenue

Recurrence: $r_n = \max\{p_n, \; p_1 + r_{n-1}, \; p_2 + r_{n-2}, \ldots, p_{n-1} + r_1 \}, \; r_1 = p_1$

- p_n if we do not cut at all
- $p_1 + r_{n-1}$ if the first piece has length 1
- $p_2 + r_{n-2}$ if the first piece has length 2
- …

Rod Cutting: The Algorithm

Define: Let r_n be the maximum revenue obtainable from cutting a rod of length n.

Recurrence: $r_n = \max\{p_n,\ p_1 + r_{n-1},\ p_2 + r_{n-2}, \dots, p_{n-1} + r_1\}$, $r_1 = p_1$

```
let r[0..n] be a new array
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        q ← max(q, p[i] + r[j − i])  max revenue if first
                                     piece has length ∈ [1, j]
    r[j] ← q
return r[n]
```

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

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1									

Rod Cutting: The Algorithm

Define: Let r_n be the maximum revenue obtainable from cutting a rod of length n.

Recurrence: $r_n = \max\{p_n,\ p_1 + r_{n-1},\ p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\}$, $r_1 = p_1$

```
let r[0..n] be a new array
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        q ← max(q, p[i] + r[j − i])  max revenue if first
                                     piece has length ∈ [1, j]
    r[j] ← q
return r[n]
```

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

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5								

$r[2] = max(p_1 + r_1,\ p_2 + r_0) = max(5 + 0, 1 + 1) = 5$

Rod Cutting: The Algorithm

Define: Let r_n be the maximum revenue obtainable from cutting a rod of length n.

Recurrence: $r_n = \max\{p_n,\ p_1 + r_{n-1},\ p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\},\ r_1 = p_1$

```
let r[0..n] be a new array
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        q ← max(q, p[i] + r[j − i])  max revenue if first
                                     piece has length ∈ [1, j]
    r[j] ← q
return r[n]
```

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

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8							

$r[3] = max(p_1 + r_2, p_2 + r_1, p_3 + r_0) = max(1 + 5, 5 + 1, 8 + 0) = 8$

Rod Cutting: The Algorithm

Define: Let r_n be the maximum revenue obtainable from cutting a rod of length n.

Recurrence: $r_n = \max\{p_n,\ p_1 + r_{n-1},\ p_2 + r_{n-2}, \dots, p_{n-1} + r_1\}$, $r_1 = p_1$

```
let r[0..n] be a new array
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        q ← max(q, p[i] + r[j − i]) max revenue if first
                                    piece has length ∈ [1, j]
    r[j] ← q
return r[n]
```

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

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10						

$r[4] = max(p_1 + r_3, p_2 + r_2, p_3 + r_1, p_4 + r_0)$= $max(1 + 8, 5 + 5, 8 + 1, 9 + 0)$= 10

Rod Cutting: The Algorithm

Define: Let r_n be the maximum revenue obtainable from cutting a rod of length n.

Recurrence: $r_n = \max\{p_n,\ p_1 + r_{n-1},\ p_2 + r_{n-2}, \ldots, p_{n-1} + r_1\}$, $r_1 = p_1$

```
let r[0..n] be a new array
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        q ← max(q, p[i] + r[j − i])  max revenue if first
                                     piece has length ∈ [1, j]
    r[j] ← q
return r[n]
```

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

This only finds max-**revenue**.

How can we construct SOLUTION that yields max-revenue

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then    similar to previous alg
            q ← p[i] + r[j − i]        but keeps track in s[j]
            s[j] ← i                   of where max occurred
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j − s[j]
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0										
$s[i]$	0										

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
     q ← −∞
     for i ← 1 to j
          if q < p[i] + r[j − i] then    similar to previous alg
               q ← p[i] + r[j − i]       but keeps track in s[j]
               s[j] ← i                  of where max occurred
     r[j] ← q
j = n
while j > 0 do
     print s[j]
     j ← j − s[j]
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	**1**	5	8	9	10	17	17	20	24	30
$r[i]$	**0**	1									
$s[i]$	0	1									

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then     similar to previous alg
            q ← p[i] + r[j − i]         but keeps track in s[j]
            s[j] ← i                    of where max occurred
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j − s[j]
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	**1**	**5**	8	9	10	17	17	20	24	30
$r[i]$	**0**	**1**	5								
$s[i]$	0	1	**2**								

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then     similar to previous alg
            q ← p[i] + r[j − i]         but keeps track in s[j]
            s[j] ← i                    of where max occurred
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j − s[j]
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8							
$s[i]$	0	1	2	3							

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then     similar to previous alg
            q ← p[i] + r[j − i]         but keeps track in s[j]
            s[j] ← i                    of where max occurred
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j − s[j]
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	**1**	**5**	**8**	**9**	10	17	17	20	24	30
$r[i]$	**0**	**1**	**5**	**8**	10						
$s[i]$	0	1	2	3	**2**						

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then        similar to previous alg
            q ← p[i] + r[j − i]            but keeps track in s[j]
            s[j] ← i                       of where max occurred
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j − s[j]
```

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem in $s[j]$

```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then
            q ← p[i] + r[j − i]
            s[j] ← i
    r[j] ← q
j = n
while j > 0 do
    print s[j]        pull off first piece
    j ← j − s[j]       & construct opt soln
                      of remainder
```

Reconstructing solution for n =9

j=9 s[j] = 3

j=9-3 =6 s[j] = 6

Solution is to cut 9 into {3, 6}

i	0	1	2	3	4	5	6	7	8	9	10
$p[i]$	0	1	5	8	9	10	17	17	20	24	30
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

A Quick Review

- Our Goal was to solve a problem of size n
  - Maximize Revenue from cutting a rod of size n
- Defined smaller subproblems
  - Maximize Revenue from cutting a rod of size i: $i \le n$
- Noted that structure of optimal solution can be expressed in terms of optimal solution of subproblems
  - Maximal revenue solution does an initial cut into one piece of size i, and cuts the remaining n-i size rod optimally
- Implicitly used the optimal substructure property
  - e.g., the subproblem of size n-i must be cut optimally.
    If it wasn't we could replace the solution to the subproblem with an optimal one, contradicting optimality of original solution
- Used this to develop a recurrence describing cost of optimal solution in terms of previously calculated optimal solutions to subproblems
  - $r_n = \max\{p_n,\ p_1 + r_{n-1},\ p_2 + r_{n-2}, \dots, p_{n-1} + r_1\},\ \ r_1 = p_1$
- Recurrence translated into algorithm

Outline

1. Introduction to Dynamic Programming

2. The Rod-Cutting Problem

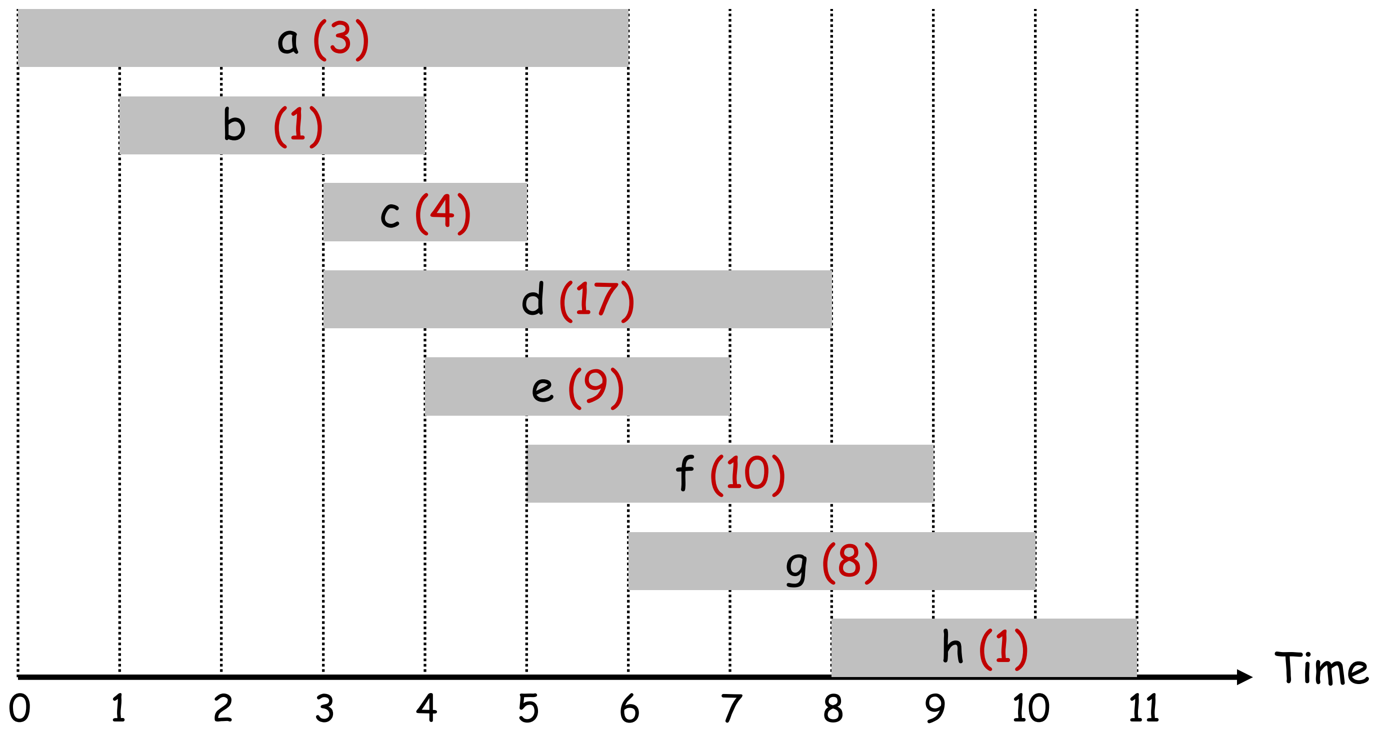
3. Weighted Interval Scheduling

4. Review & Summary

Weighted Interval Scheduling

Weighted interval scheduling problem.

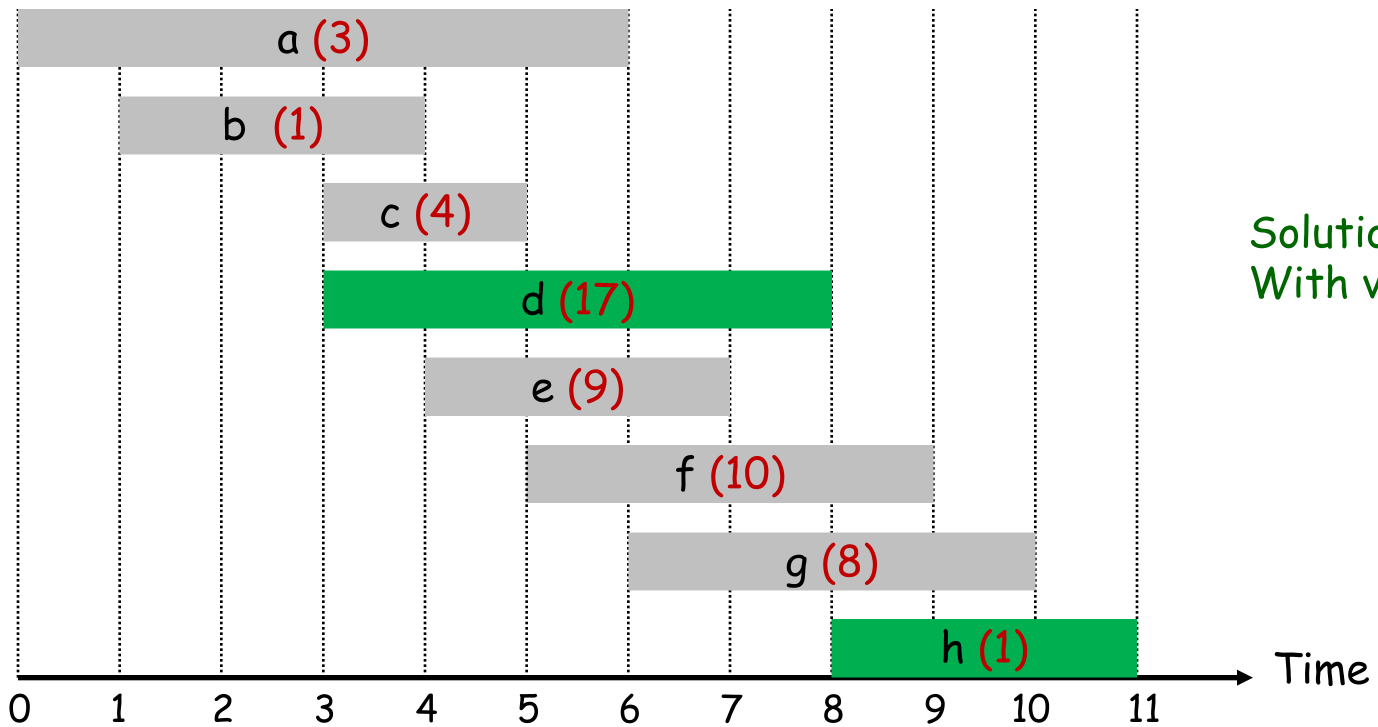
- Job j starts at s_j, finishes at f_j, and has weight (or value) v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum-weight subset of mutually compatible jobs.



Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight (or value) v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum-weight subset of mutually compatible jobs.



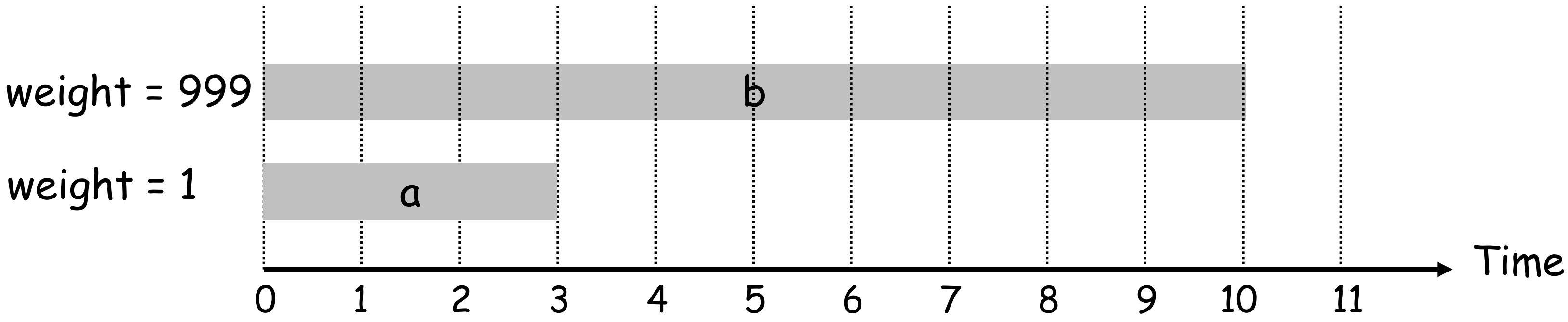
Solution is {d,h}
With value 18

Unweighted Interval Scheduling Review

Recall. Greedy algorithm works if all weights are 1.

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail miserably if arbitrary weights are allowed.



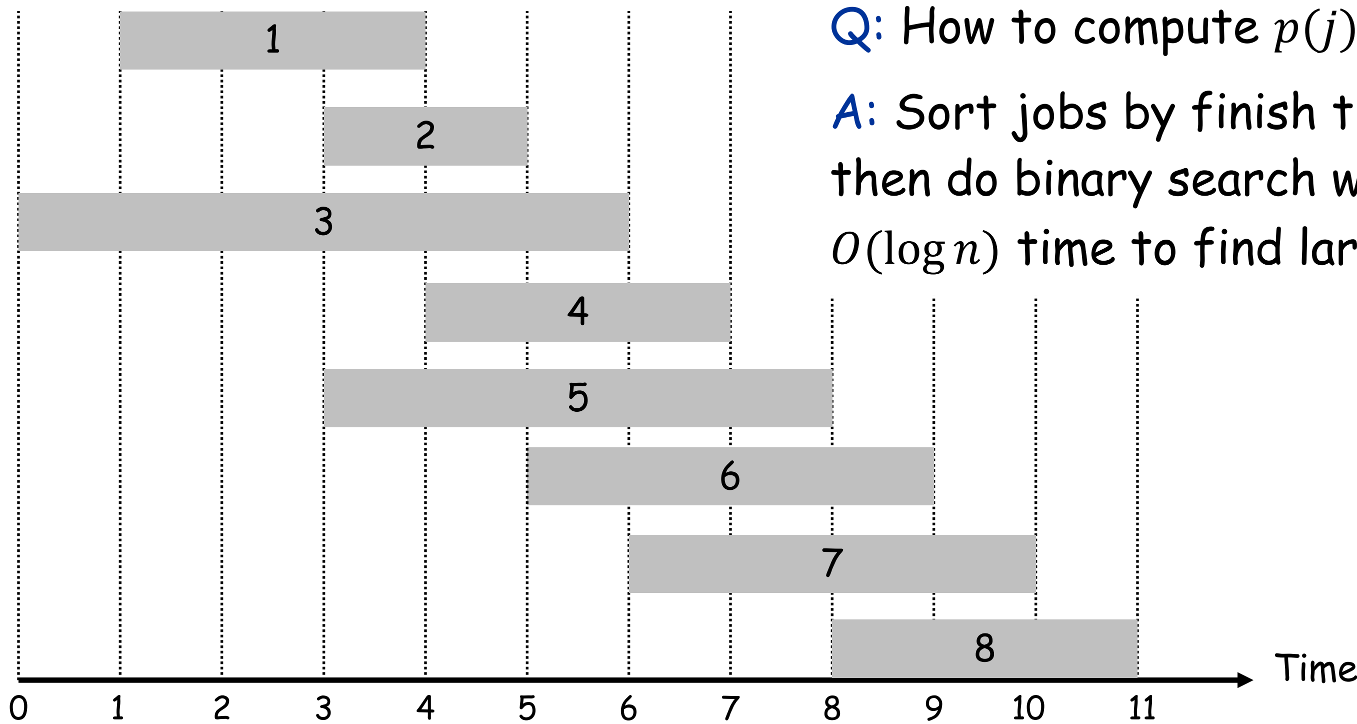
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with job j.

Note: all jobs i' *with* $p(j) < i' < j$ *are not compatible with j*

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



Q: How to compute $p(j)$?

A: Sort jobs by finish time, and then do binary search with s_j in $O(\log n)$ time to find largest comp f_j.

The Recurrence

Def. $V[j]$ = value of optimal solution to the problem on jobs $1, 2, \ldots, j$.

Recurrence:

- **Case 1: OPT selects job j.**
  - can't use incompatible jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$
  - must include an optimal solution to problem on jobs $1, 2, \ldots, p(j)$

- **Case 2: OPT does not select job j.**
  - must include optimal solution to problem on jobs $1, 2, \ldots, j-1$

$$V[j] = \max\{\, v_j + V[p(j)], V[j-1] \,\}$$
$$V[0] = 0$$

```
sort all jobs by finish time
V[0] ← 0
for j ← 1 to n
    V[j] ← max{v_j + V[p(j)], V[j − 1]}
return V[n]
```

Running time: $\Theta(n \log n)$, Space: $\Theta(n)$

The complete algorithm

```
sort all jobs by finish time
V[0] ← 0
for j ← 1 to n
    if v_j + V[p(j)] > V[j − 1] then    Job j in opt soln
        V[j] ← v_j + V[p(j)]            for jobs [1..j]
        keep[j] ← 1
    else
        V[j] ← V[j − 1]                 Job j NOT in opt
        keep[j] ← 0                     soln for jobs [1..j
j ← n

while j > 0 do
    if keep[j] = 1 then                 If  j in final soln
        print j                         then remainder (to
        j ← p(j)                        left) of soln is
    else                                opt soln for [1..p[j]]
        j ← j − 1
```

Alg on previous page only found optimal **value** of solution.

To find actual solution, we need to keep track of which jobs are kept in solution

Running time: $\Theta(n \log n)$

Outline

1. Introduction to Dynamic Programming

2. The Rod-Cutting Problem

3. Weighted Interval Scheduling

4. Review & Summary

A Quick Review

- Our Goal was to solve a problem of size n
  - Find maximum weighted compatible interval set among n
- Defined smaller subproblems
  - Find max interval set among FIRST j intervals: j$\leq n$
  - Needed to define meaning of FIRST j intervals
- Noted that structure of optimal solution can be expressed in terms of optimal solution of subproblems
  - If jth interval used, then rest of solution is optimal solution on first p(j) intervals; if not used, solution is optimal solution on first j-1 intervals
- Implicitly used the optimal substructure property
  - If did not use OPTIMAL solution for subproblem, could replace solution used for that subproblem with optimal one, increasing weight of interval set chosen, contradicting optimality of original solution
- Used this to develop a recurrence describing cost of optimal solution in terns of previously calculated optimal solutions to subproblems
  - $V[j] = \max\{v_j + V[p(j)], V[j-1]\}$, $V[0] = 0$
- Recurrence translated into algorithm

The Recurrences

1. Fibonacci Numbers

$$F(n) = F(n-1) + F(n-2), \qquad F(1) = 1,\ F(2) = 2$$

2. The Rod Cutting Problem

r_n is maximum revenue from cutting rod of length n

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}, \qquad r_0 = 0$$

3. Weighted Interval Scheduling

$V[j]$ is maximum-weight subset of mutually compatible jobs on jobs $1, 2, \ldots, j$.

$$V[j] = \max\{\, v_j + V[p(j)], V[j-1] \,\}, \qquad V[0] = 0$$

Dynamic Programming: Summary

1. Structure: Analyze structure of an optimal solution, and thereby define subproblems that need to be solved

2. Recurrence: Establish the relationship between the optimal value of the problem and those of some subproblems (optimal substructure).

3. Bottom-up computation: Compute the optimal values of the smallest subproblems first, save them in the table. Then compute optimal values of larger subproblems, and so on, until the optimal value of the original problem is computed.

4. Construction of optimal solution: Record the optimal decisions made for each subproblem. At the end, assemble the optimal solution by tracing the back computation in the previous step.

Remark: The first two steps are interdependent and the most important ones.

The last two steps are usually straightforward.