Lecture 11: Dynamic Programming

First Example: Stairs Climbing

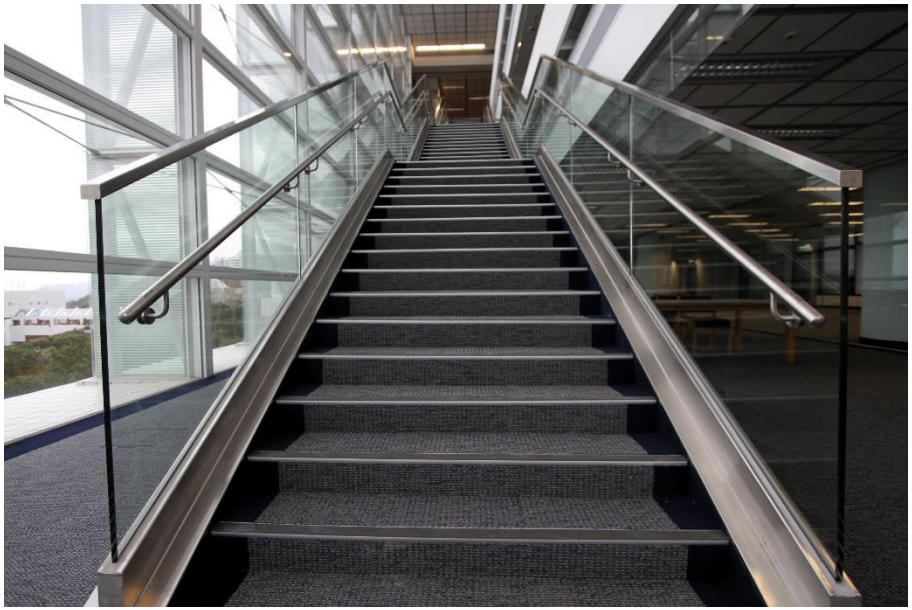
Problem: Suppose you can take 1 or 2 stairs with one step. How many different ways can you climb n stairs?

Solution: Let $f(n)$ be the number of different ways to climb n stairs.

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

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

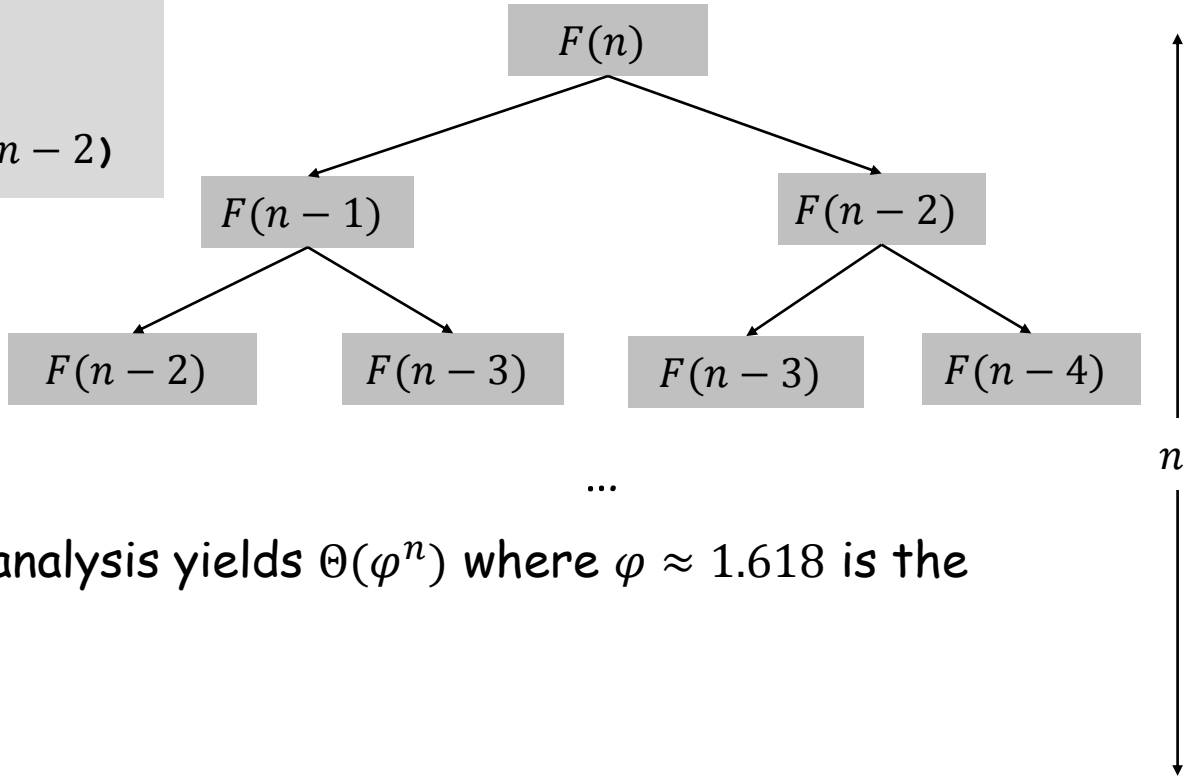
Q: How to compute $f(n)$?



Solving the recurrence by recursion

$$f(1) = 1, f(2) = 2, f(3) = 3, \ldots$$
$$f(n) = f(n-1) + f(n-2)$$

```
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 complicated 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 recursion

$$f(1) = 1, f(2) = 2, f(3) = 3, \ldots$$
$$f(n) = f(n-1) + f(n-2)$$

```
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.

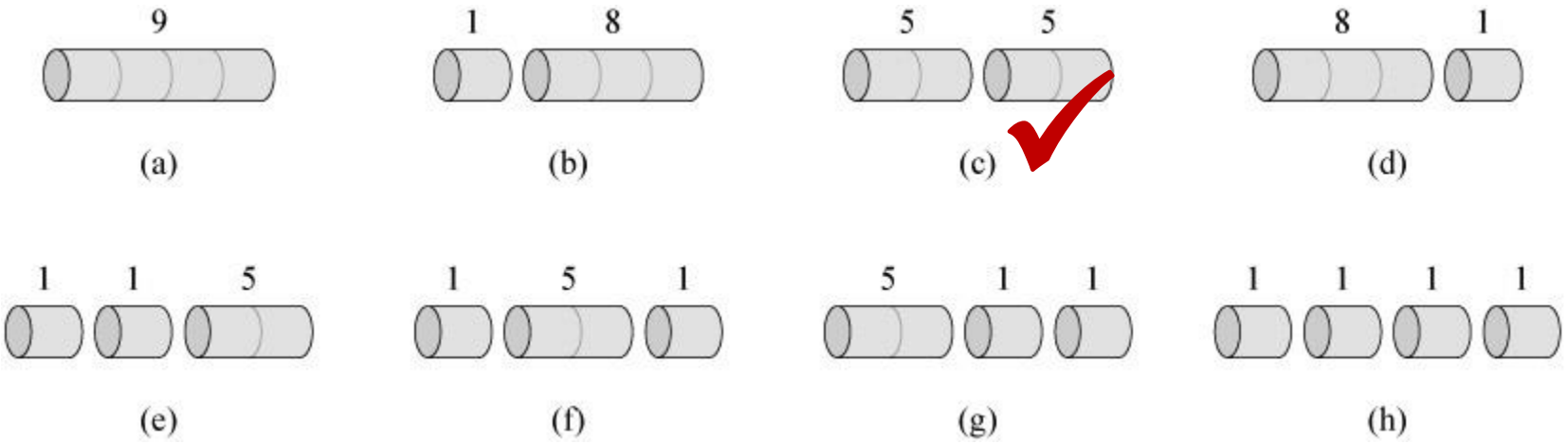
Dynamic programming:

- Used to solve recurrences
- Avoid solving a subproblem more than once by memorization
- Can be either top-down or bottom-up
  - Bottom-up is usually more efficient in practice
- "Programming" here means "planning", not coding!

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



Rod Cutting: The Algorithm

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

Recurrence: $r_n = \max\{p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \dots, p_{n-1} + r_1\}, r_0 = 0$

- 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
- …

```
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])
    r[j] ← q
return r[n]
```

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

Reconstructing the Solution

Idea: Remember the optimal decision for each subproblem.

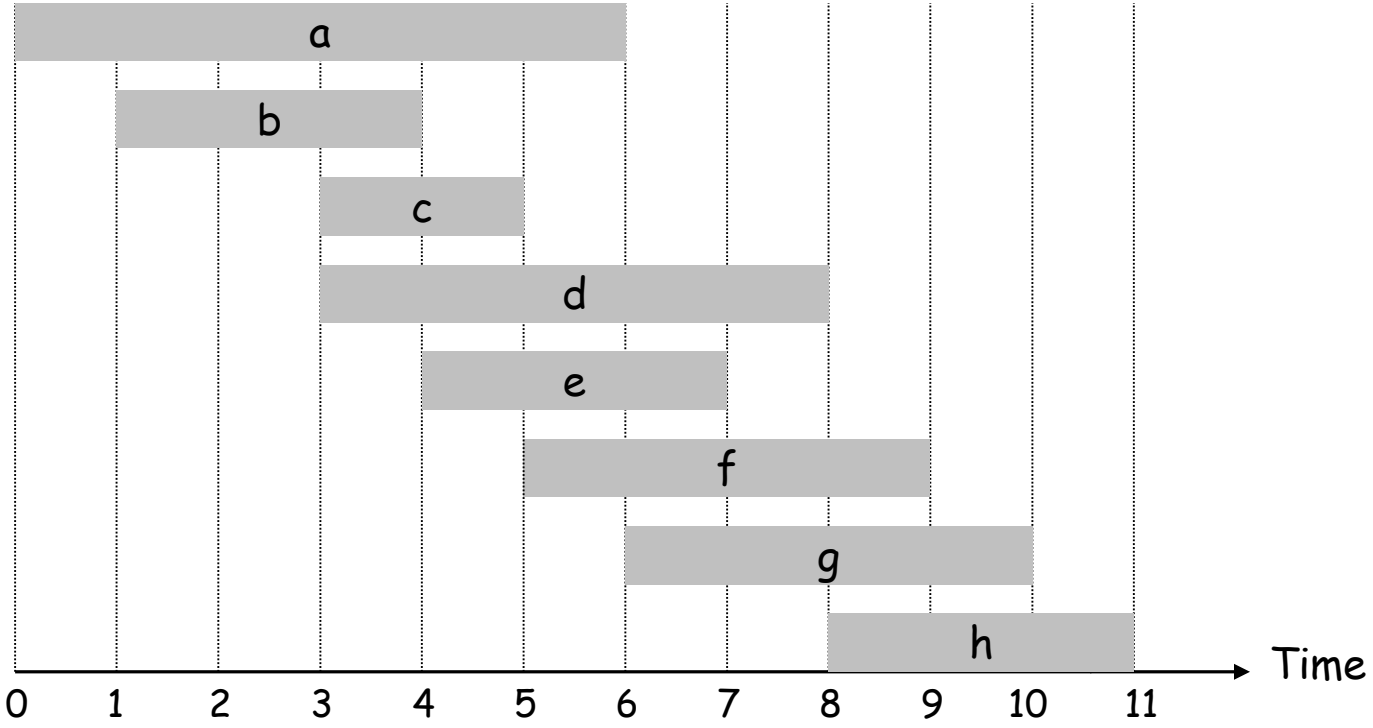
```
let r[0..n] and s[0..n] be new arrays
r[0] ← 0
for j ← 1 to n
    q ← −∞
    for i ← 1 to j
        if q < p[i] + r[j − i] then
            q ← p[i] + r[j − i]
            s[j] ← i
    r[j] ← q
j = n
while j > 0 do
    print s[j]
    j ← j − s[j]
```

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

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.

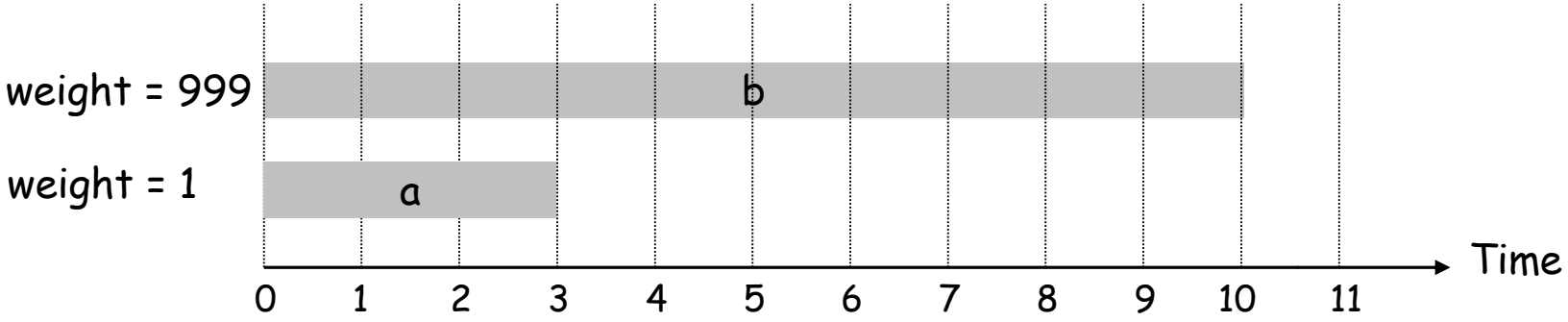


Unweighted Interval Scheduling Review

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

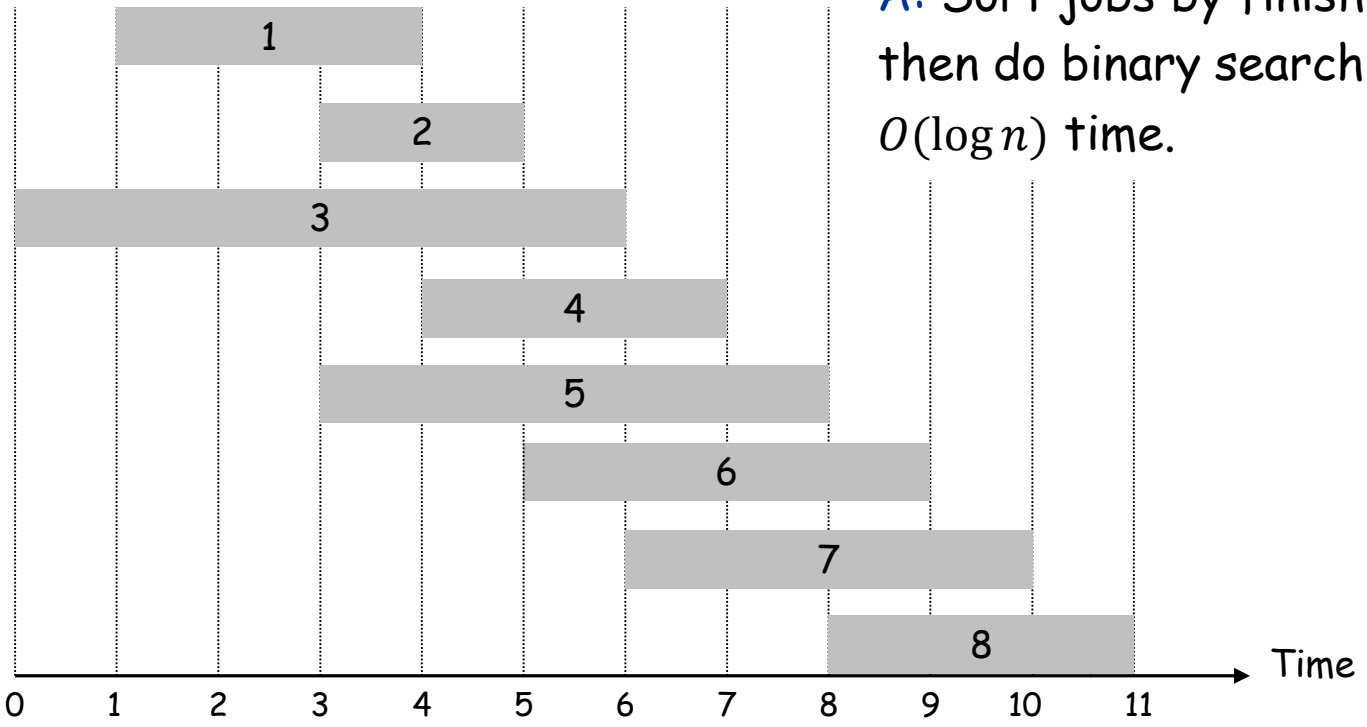
Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$.

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



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 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
        V[j] ← v_j + V[p(j)]
        keep[j] ← 1
    else
        V[j] ← V[j − 1]
        keep[j] ← 0
j ← n
while j > 0 do
    if keep[j] = 1 then
        print j
        j ← p(j)
    else
        j ← j − 1
```

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

Dynamic Programming: Summary

Structure: Analyze structure of an optimal solution, and thereby choose a definition of subproblems.

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

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.

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 they are the most important steps. The last two steps are usually straightforward.