Introduction to Dynamic Programming

Exercise on Stairs Climbing

Problem: You can climb 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.

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

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

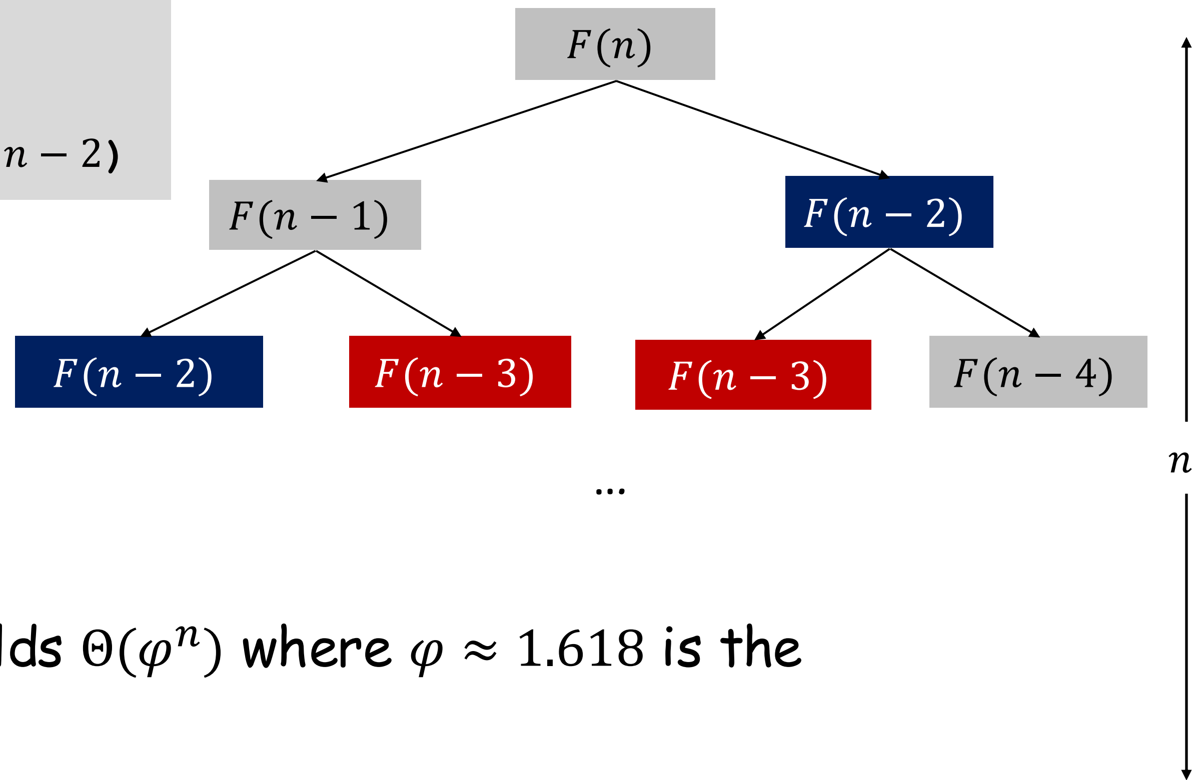
Because you can only reach n-th stair from the $(n-1)$-th or $(n-2)$-th stair.

Observation: $F(1), F(2), \ldots, F(n)$ are the Fibonacci numbers.

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 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 F of size n
F[1] ← 1
F[2] ← 2
for i = 3 to n
      F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Running time: $\Theta(n)$

Space: $\Theta(n)$?

Dynamic programming:

- Used to solve recurrences
- Avoid solving a subproblem more than once by storing solutions
- Usually done "bottom-up", filling in subproblem solutions in table in order from "smallest" to largest".
  - There is also "top-down" version (memoization).
- "Programming" here means "planning", not coding!

Dynamic Programming (DP)

- Similar to Greedy, DP is used for optimization problems, but it can find optimal solutions when Greedy fails
- Similar to Divide and Conquer (D&C), DP partitions a problem into subproblems
- D&C works recursively top-down
  Solve some smaller problems that are combined for the larger problem
- DP is also based on a recursive problem definition, using problems of smaller size

Main idea of DP

Recursively define the value of an optimal solution using smaller problem sizes (also called optimal substructure - this is the difficult part)

- Once you have the recurrence, it is easy to write recursive pseudocode (top down approach). To avoid re-computing the same problems many times, you can use memoization. Store results, and if there is a subsequent call to a result that has been already computed, use the stored result instead of executing again.
- We use the non-recursive, bottom up approach: solve the smallest problems first, store the solutions, and use them to solve larger problems.

Fibonacci Numbers
Recursive algorithm with Memoization
Non-recursive bottom up algorithm

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

Memoized – Top Down

Allocate array F[1..n]=[1,2,0,0,…] to store Fibonacci numbers already computed. Perform recursive call only for non-computed values.

TD-F(n):

if F[n] $\neq 0$ **return F[n]**

F[n] $\leftarrow$ **TD-F**$(n-1)$**+TD-F**$(n-2)$

return F[n]

Non- recursive – Bottom Up

No need for array because only last two numbers are useful

BU-F(n):

$F_p \leftarrow 1$; $F \leftarrow 2$

for $i = 3$ **to** n

$\quad temp \leftarrow Fp$; $F_p \leftarrow F$

$\quad F \leftarrow F_p + temp$

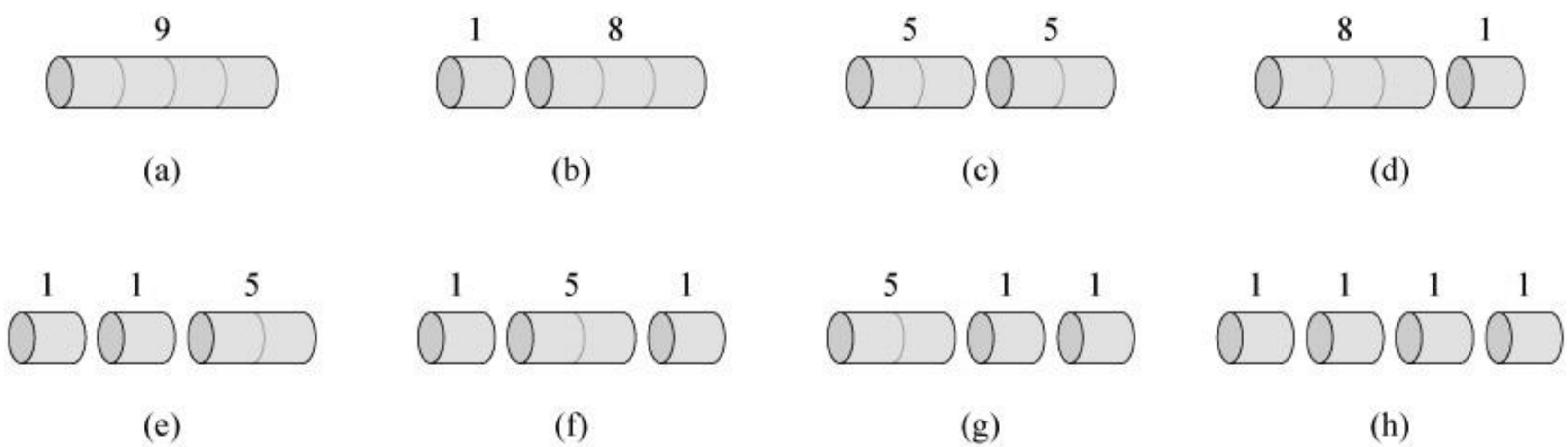
return F

- Both avoid solving a subproblem more than once by storing solutions.
- Bottom up is faster as it avoids recursive calls.
- It also permits space optimization ($\Theta(1)$ *versus* $\Theta(n)$) for top down.
- We will use bottom up dynamic programming in this class

The Rod Cutting Problem

Problem: Given a rod of length n and prices p_i for $i = 1, \dots, n$, 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 $r_1, r_2, \dots, r_{n-1}$

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

Sell in one piece

Revenue p_n

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

Rod Cutting: Another View

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

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

- 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
- …

Step 2 Recurrence: Solve the problem for rod length $1, 2, \dots, n$

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									

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

```
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	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 $n = 9$

$j = 9$ $\quad$ $s[j] = 3$

$j = 9 - 3 = 6$ $\quad$ $s[j] - 6$

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

Exercise on Maximum Sum problem

Recall the Max Sum problem:

Let A be a sequence of n positive numbers $a_1, a_2, \ldots, a_n$.

Find a subset S of A that has the maximum sum, provided that if we select a_i in S, then we cannot select a_{i-1} or a_{i+1}.

For example, for $A = 7, 8, 6, 3$.

Greedy would select the largest number, delete the two neighbors and continue

Greedy solution: $G = \{8,3\}$

Optimal solution: $O = \{7,6\}$

Design a DP Algorithm that always finds the optimal solution.

DP for Maximum Sum

General idea:

Let A_i be the subsequence of A containing the first i numbers ($i < n$):

$$a_1, a_2, \dots, a_i$$

Let S_i be the solution of problem A_i.

Let W_i be the sum of numbers in S_i.

Two possibilities for a_i :

$$a_i \in S_i \Longrightarrow a_{i-1} \notin S_i \;\; and \;\; W_i = W_{i-2} + a_i$$

$$a_i \notin S_i \Longrightarrow a_{i-1} \; can \; be \; in \; S_i \;\; and \;\; W_i = W_{i-1}$$

Step 1: (Recurrence): Solution is larger of the two cases:

$$W_i = \max\{W_{i-2} + a_i, W_{i-1}\}$$

Step 2: Solve the problem incrementally from smaller to larger,

i.e. for $A_1, A_2, \dots, A_n$. The final solution is A_n.

DP pseudocode

$W[1] = a_1;\ \ b[1] = true$ // in general $b[i] = true$ means that $a_i \in S_i$
If $a_2 > a_1$
    $W[2] = a_2;\ \ b[2] = true$ // $a_2 \in S_2$
  else
    $W[2] = a_1;\ \ b[2] = false$ // $a_2 \notin S_2$
for $i = 3\ to\ n$
    If $W[i-2] + a_i > W[i-1]$
        $W[i] = W[i-2] + a_i$
        $b[i] = true$ // $a_i \in S_i$
    else
        $W[i] = W[i-1]$
        $b[i] = false$ // $a_i \notin S_i$

Cost: $\Theta(n)$

Example: for $A = 1, 8, 6, 3, 7$, we have

$W[1] = 1, b[1] = true$
$W[2] = 8, b[2] = true$
$W[3] = 8, b[3] = false$
$W[4] = 11, b[4] = true$
$W[5] = 15, b[5] = true$

Printing the solution

$i = n$

while $i > 0$

    if $b[i]$ is true

        Output a_i

        $i = i - 2$

    else

        $i = i - 1$

Example: for $A = 1, 8, 6, 3, 7$, we have

$b[5] = true$; therefore, we print $a_5 = 7$ and set $i = 3$

$b[3] = false$; therefore, we set $i = 2$

$b[2] = true$; therefore, we print $a_2 = 8$ and set $i = 0$

Exercise on Minimum number of Coins

Input: Amount n and k denominations $d_1, \ldots, d_k$

Output: Minimum number of coins for amount n

Greedy solution: Give as many coins as possible from the largest denomination, then from the from second largest, and so on.

Greedy solution is not optimal for arbitrary coin denominations

- $n = 30c, d_1 = 25c, d_2 = 10c, d_3 = 1c$,
- Greedy solution: 6 (1x25+5X1)
- Optimal solution: 3 (3x10)

DP approach

We will find a formula that expresses the solution for amount n, as a *recurrence* of solutions for smaller amounts

- This is the **most crucial step**

Then, we will start solving the small problems, gradually increasing their size, until we reach n.

We will keep all solutions in a table

- The last solution in the table corresponds to the minimum number of coins for amount n

If want to remember which denominations were used, we need an additional table

Recurrence

Let $C[n]$ be the minimum number of coins for amount n

Let d_i be the last denomination used

- Then: $C[n] = 1 + C[n - d_i]$
  - Because after using one coin, the amount left is $n - d_i$

But I have to consider all k denominations

Step 1 Recurrence:

$$C[n] = 1 + \min\{C[n - d_i] : i \in [1, k]\}$$

Step 2: Solve the problem for amount $1, 2, \ldots, n$

Algorithm

```
DP-Coin(n)
C[0] = 0                // initialization
For p = 1 to n                  // for each amount (problem size)
    min = ∞
    For i = 1 to k              // for each denomination
        if (p ≥ d_i)            // If amount is at least as large as the coin
            if (1 + C[p − d_i] < min)
                min = C[p − d_i] + 1
                coin = d_i
    C[p] = min                  // min number of coins for amount p
    S[p] = coin                 // last coin used for amount p
```

Cost: $\Theta(nk)$

Printing the solution

While $n > 0$

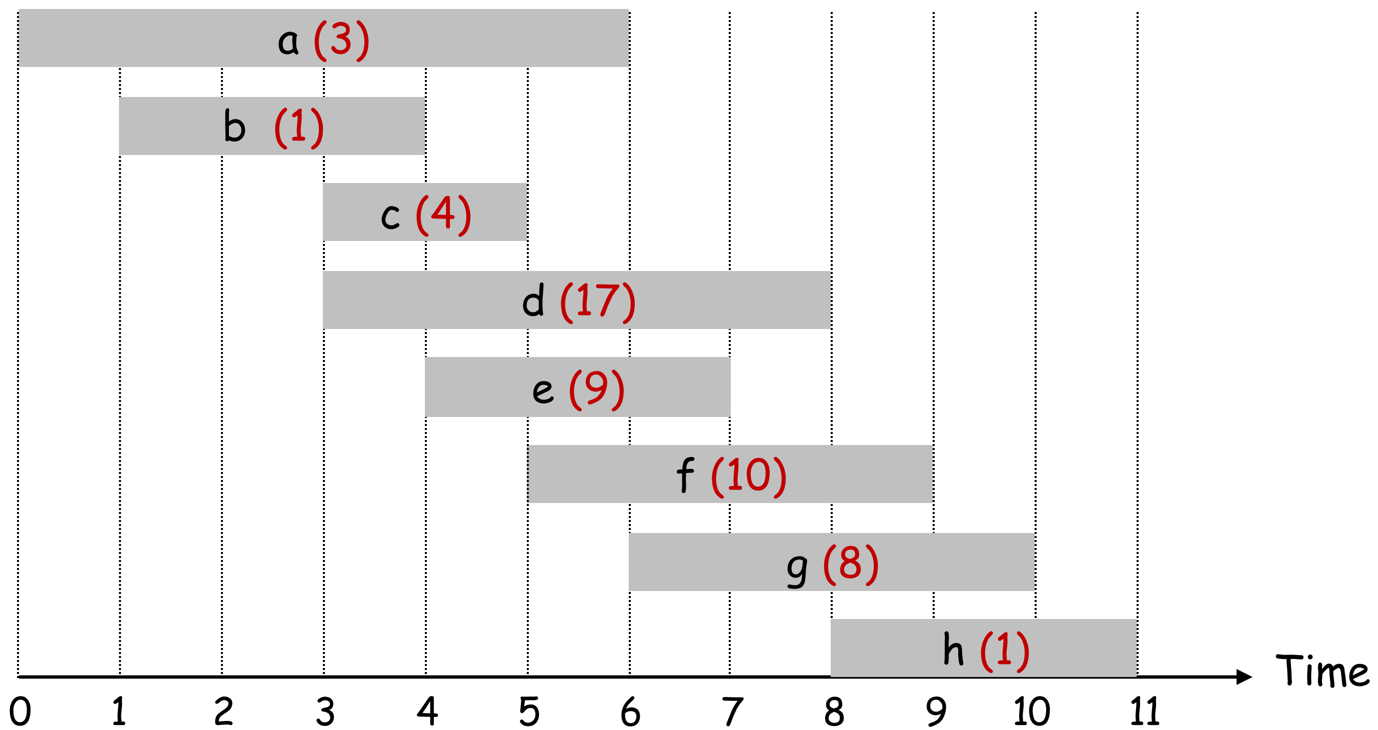
  print $S[n]$ // last coin used

  $n = n - S[n]$ // remaining amount

Weighted Interval Scheduling

Weighted interval scheduling problem.

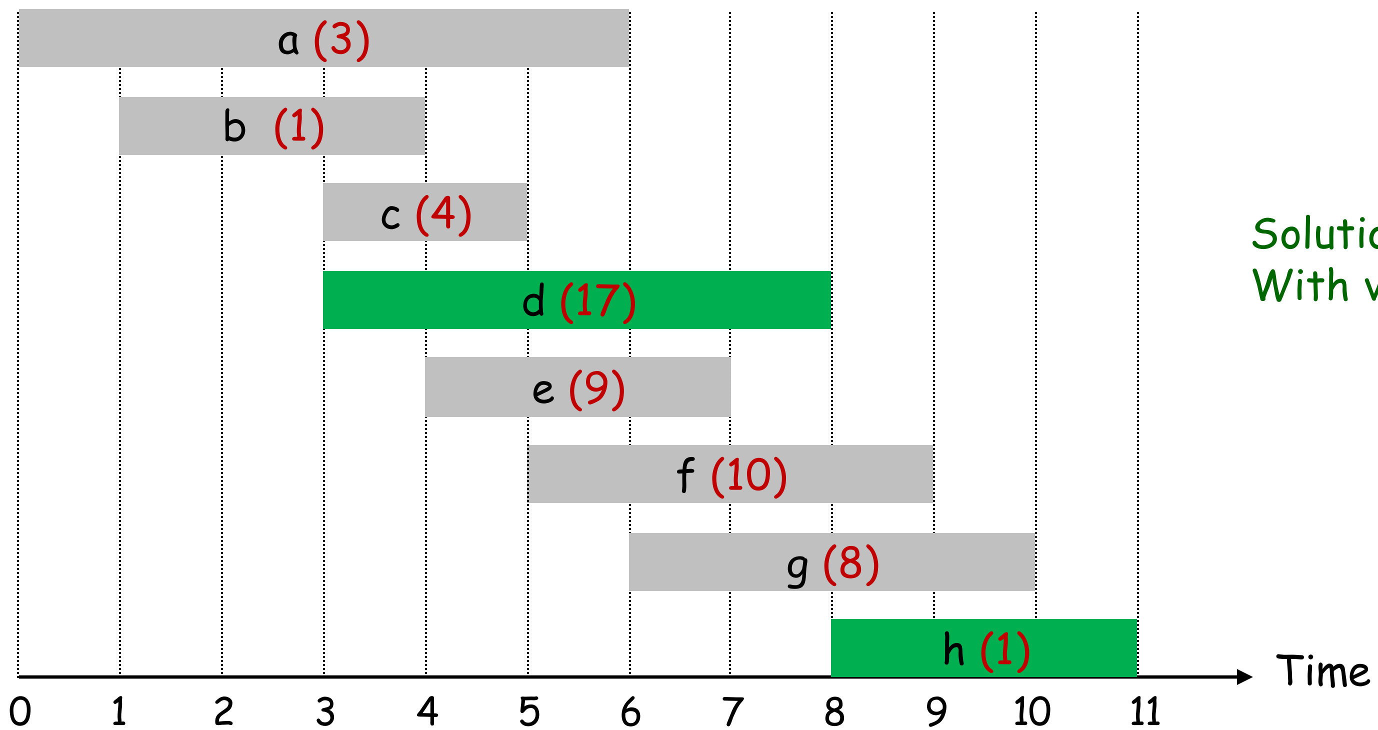
- Job j starts at s_j, finishes at f_j, and has weight (or value) v_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum-weight subset of mutually compatible jobs.



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.



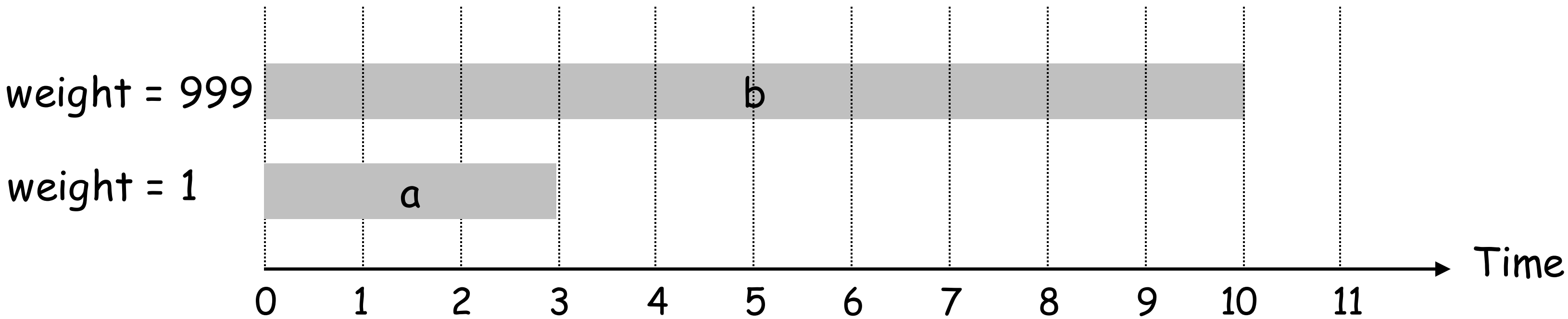
Solution is {d,h}
With value 18

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.

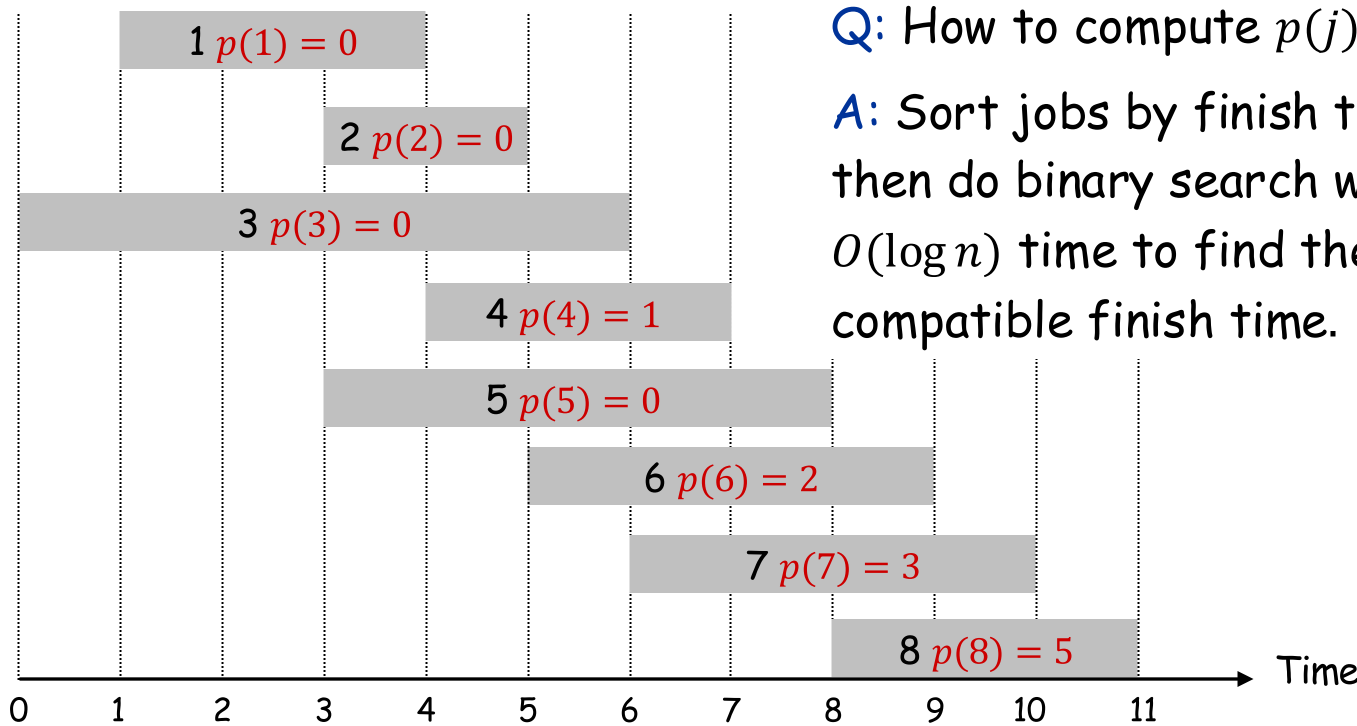


Weighted Interval Scheduling

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

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



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 the largest compatible finish time.

The Recurrence

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

Step 1 Recurrence: Solve the problem for jobs $\{1\}, \{1,2\}, \dots, \{1,2,\dots,n\}$.

Case 1: Select job j.

- can't use incompatible jobs $\{p(j)+1, p(j)+2, \dots, j-1\}$
- must include optimal solution to problem on jobs $1, 2, \dots, p(j)$

Case 2: Do not select job j.

- must include optimal solution to problem on jobs $1, 2, \dots, j-1$

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

Step 2: Solve the problem by incrementally including jobs $1, 2, \dots, n$

```
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]
```

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: $\Theta(n \log n)$

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 $1, 2, \ldots, j$.

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

Exercise on Highway Billboards

Consider a highway from west to east. The possible sites for billboards are given by numbers $x_1, x_2, \ldots, x_n$, each in the interval $[0, M]$. If you place a billboard at location x_i, you receive revenue of $r_i > 0$.

Regulations imposed by the county's Highway Department require that every two of the billboards must be at least 5 miles apart.

You wish to place billboards at a subset of sites so as to maximize your total revenue, subject to this restriction.

Q1: Describe a greedy algorithm for the problem

A1: Problem and greedy algorithm is similar to the max sum problem.

(i) Place the first billboard at the site x_i with the maximum revenue, (ii) remove x_i and all sites within 5 miles from x_i , and (iii) repeat until not site is left.

Q2: Does the above greedy algorithm always find the optimal solution?

A2: No. For the same reason that the greedy algorithm does not give optimal solution for the max sum problem. You can create counter-examples, by taking suboptimal solutions for the max sum problem, and consider that (i) the numbers in the max sum correspond to revenues of sites, and (ii) the distance between neighboring sites is 4 miles (so if you select a site you cannot select the previous and the next one).

DP for Highway Billboards

Q: Describe the recurrence for dynamic programming

Solution is similar to Weighted Interval Scheduling

Sort the sites in increasing order of location $\{x_1, x_2, \ldots x_n\}$

Let $p(i)$ be the largest $j < i$ such that $x_i - x_j > 5$, i.e., x_j is the last site before that is compatible with x_i.

Define $R(i)$ be the revenue of the optimal solution for the first i sites.

For computing $R(i)$, I have 2 options:

1. If I do not use site x_i, the revenue is $R(i-1)$.
2. If I use site x_i, the revenue is $r_i + R(p(i))$.

Thus, $R(i) = \max\{R(i-1), r_i + R\big(p(i)\big)\}$, $R(0) = 0$;

Algorithm: Solve the problem for increasing number of sites - i.e., step i, solves the problem for the first i sites in the sorted order.

Exercise on Minimum Steps To 1

On a positive integer, you can perform any one of the following 3 steps.

1.) Subtract 1 from it ($n \leftarrow n - 1$)

2.) If its divisible by 2, divide by 2. ($n \leftarrow n/2$)

3.) If its divisible by 3, divide by 3. ($n \leftarrow n/3$).

Given a positive integer n, find the minimum number of steps that takes n to 1

Examples:

1.) For $n = 1$, output: 0

2.) For $n = 4$, output: 2 (4 /2=2 /2=1)

3.) For $n = 7$, output: 3 (7 -1=6 /3=2 /2=1)

Design a Greedy and a Dynamic Programming algorithm for the problem.

Greedy Minimum Steps To 1

Choose the step, which makes n as low as possible and continue the same, till it reaches 1.

```
Greedy-Min_steps(int n)
S ← 0 //counter for the number of steps
While n > 1
        S ← S + 1
        if (i%3 = 0) then n ← n/3
        else if (i%2 = 0) then n ← n/2
        else n ← n − 1
return S
```

Suboptimal.

Given $n = 10$, Greedy: 10 /2=5 -1=4 /2=2 /2=1 (4 steps).

Given $n = 10$, Optimal: 10 -1=9 /3=3 /3=1 (3 steps).

DP Minimum Steps To 1

Reccurence:

$$S(n) = 1 + \min\{S(n-1), S(n/2), S(n/3)\}$$
$$S(1) = 0$$

$S[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
  $S[i]$= $1 + S[i-1]$
  if $(i\%2 = 0)$ then $S[i] \leftarrow \min(S[i], 1 + S[i/2])$
  if $(i\%3 = 0)$ then $S[i] \leftarrow \min(S[i], 1 + S[i/3])$
return $S[n]$