CSCI-UA.0480-004
Algorithmic Problem Solving

Brett Bernstein and Sean McIntyre

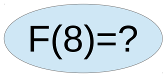
Lecture 09: Dynamic Programming

Fibonacci sequence

- Y'know, the famous one
  - 1, 1, 2, 3, 5, 8, 13, 21, ...
- As a function:
  - F(1) = 1
  - F(2) = 1
  - F(i) = F(i-1) + F(i-2) for i = 3, 4, ...

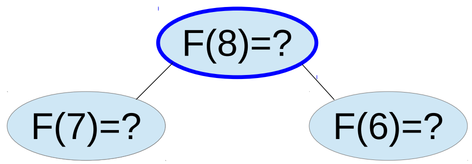
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



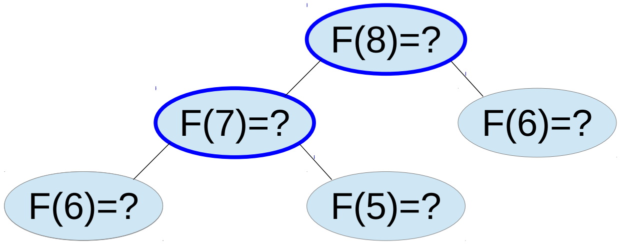
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



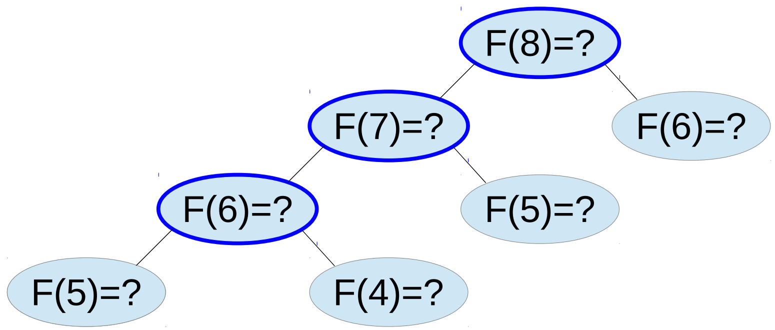
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



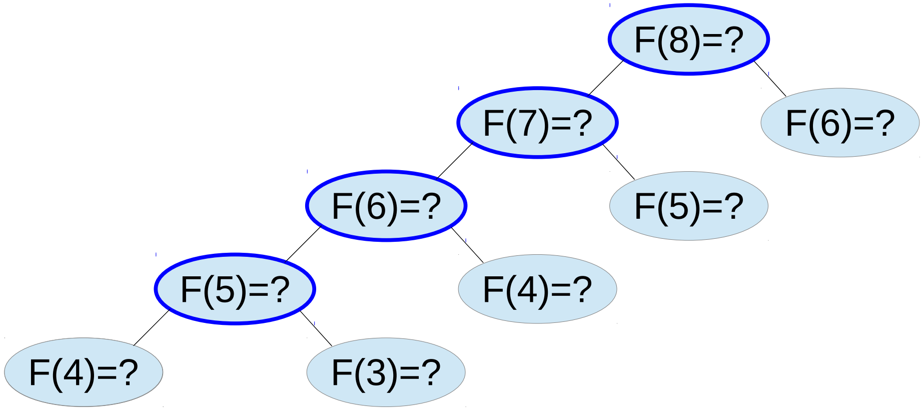
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



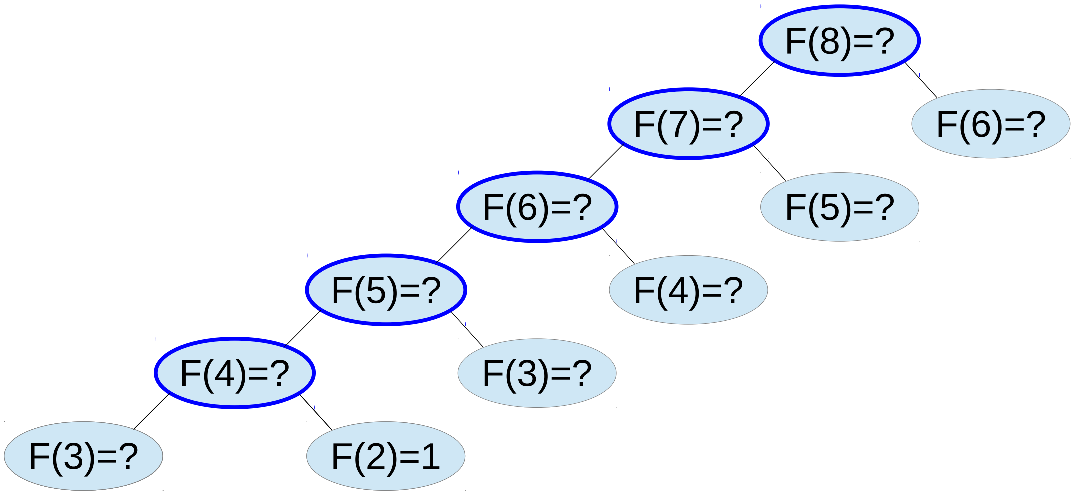
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



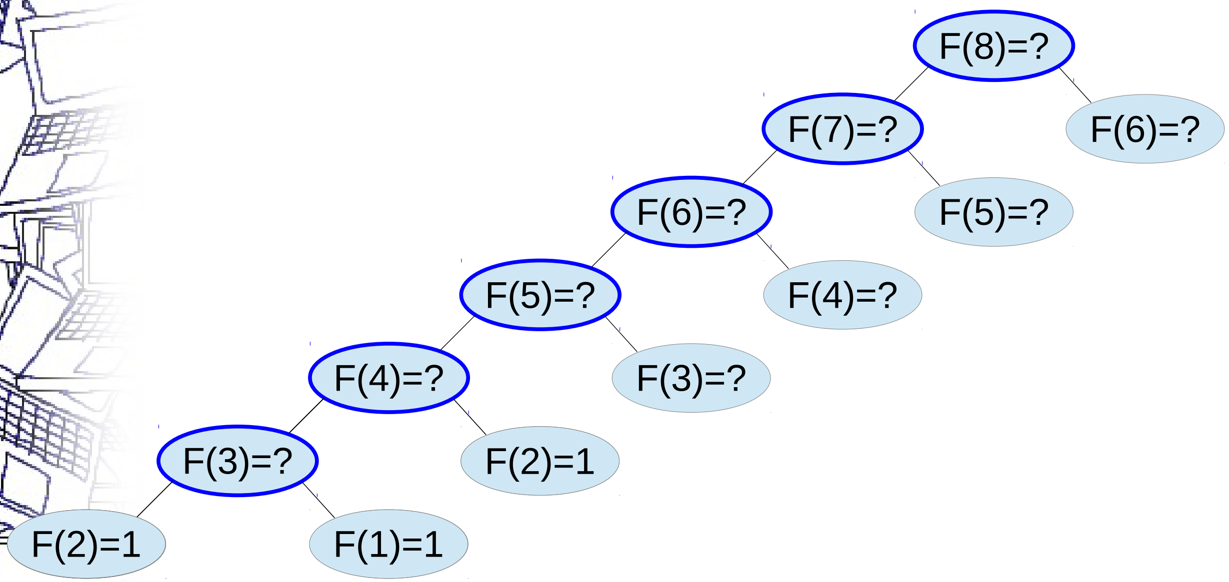
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



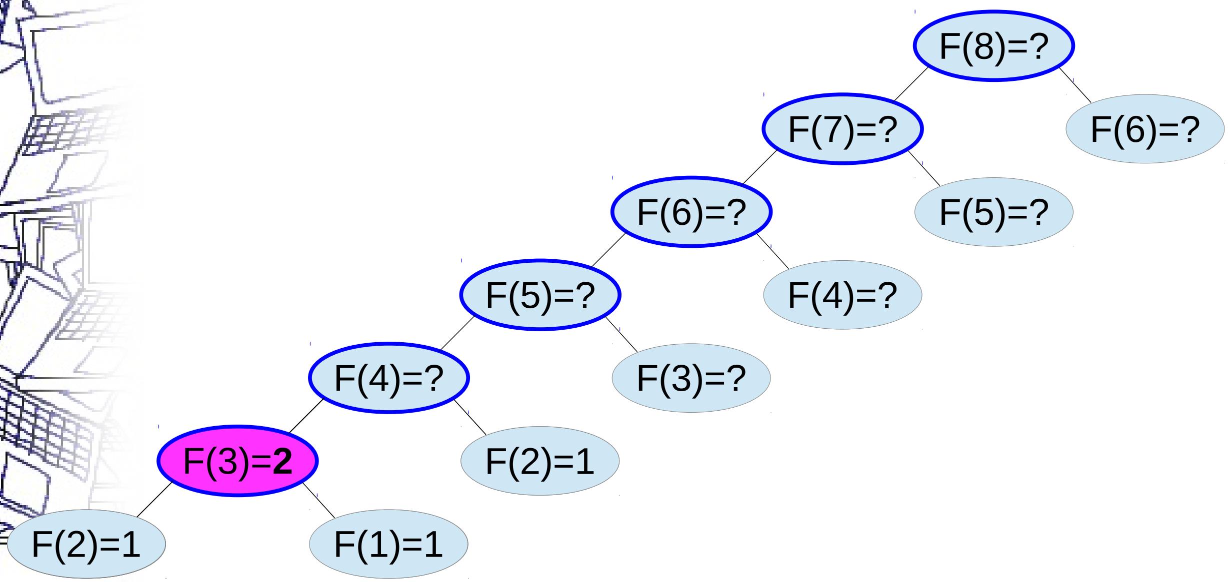
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



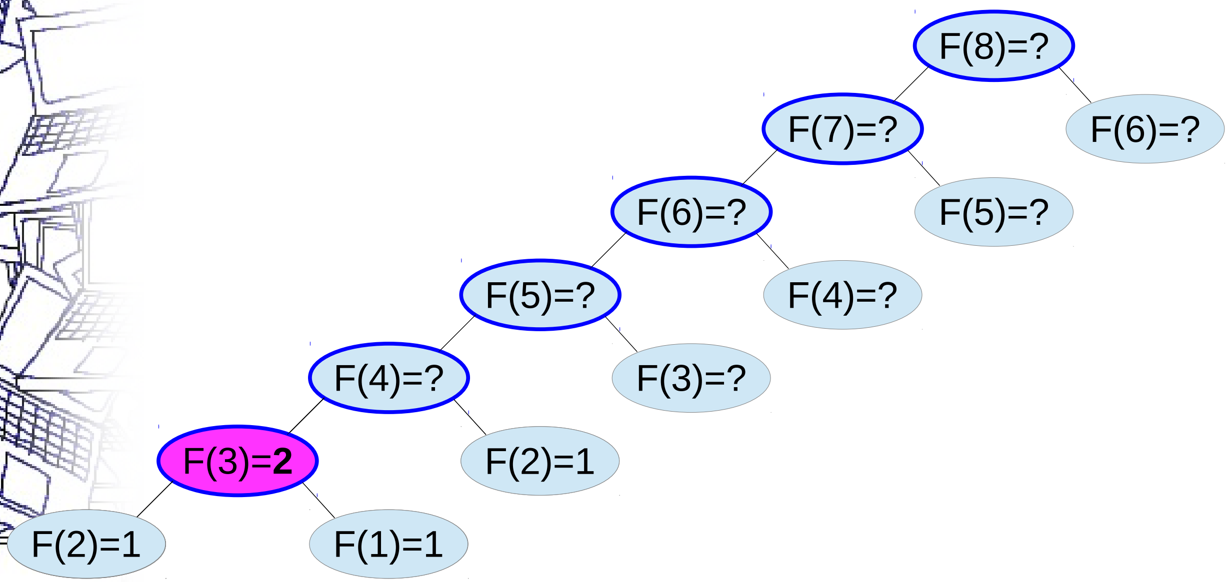
Fibonacci sequence

- Top-down DP memoized sequence for F(8)



Fibonacci sequence

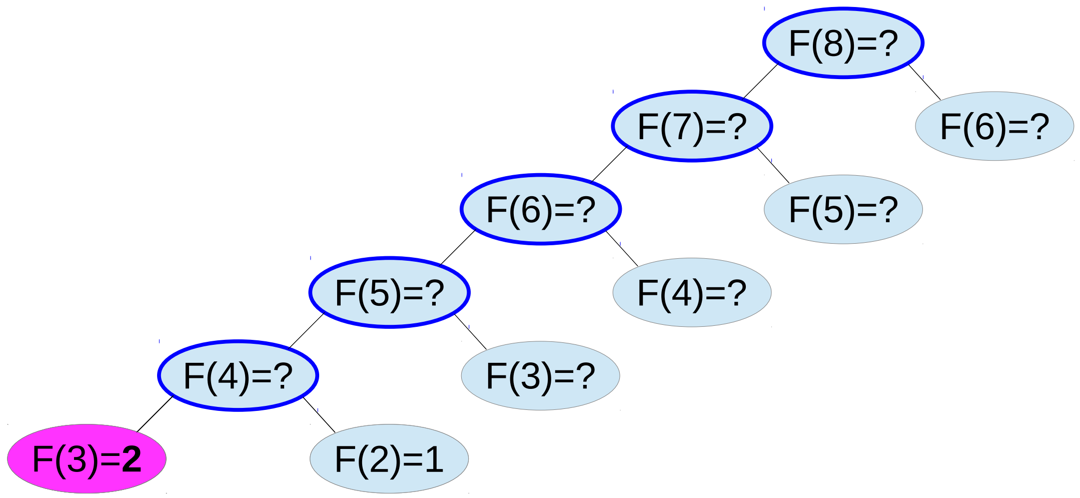
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2

Fibonacci sequence

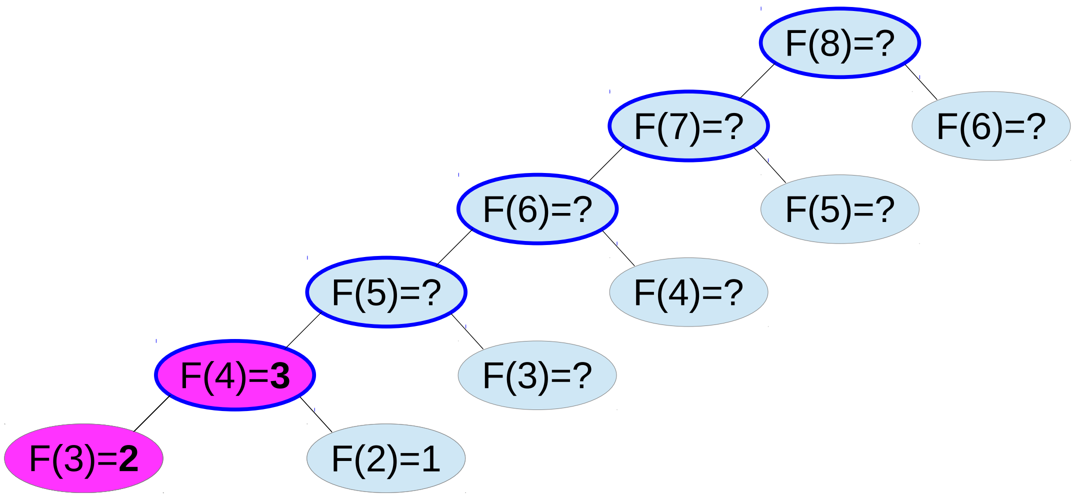
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2

Fibonacci sequence

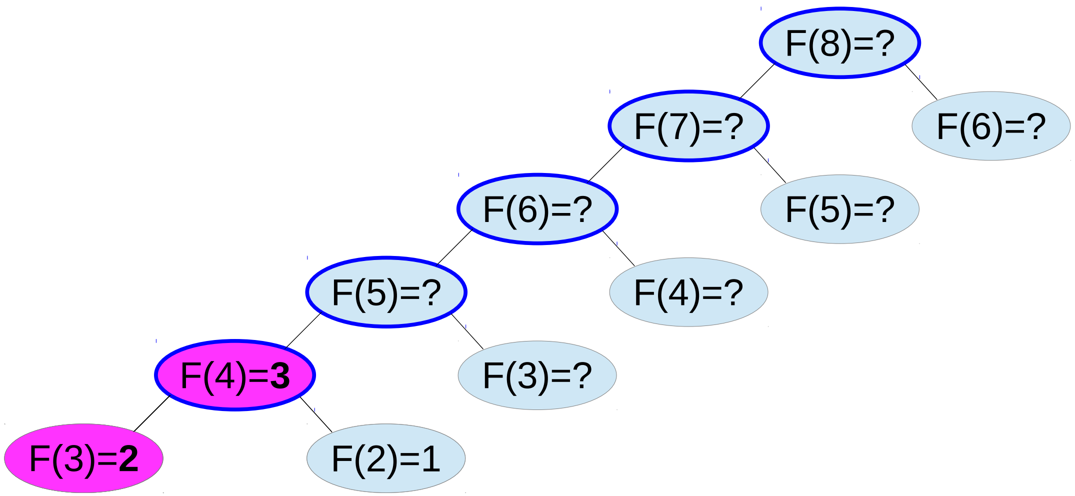
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2

Fibonacci sequence

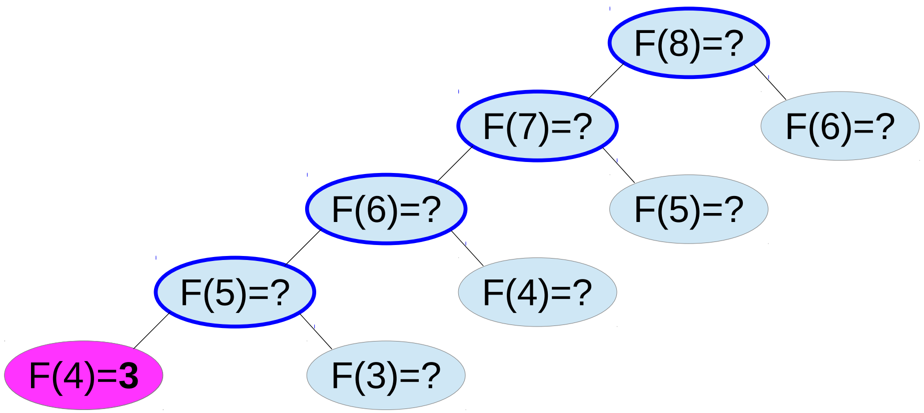
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3

Fibonacci sequence

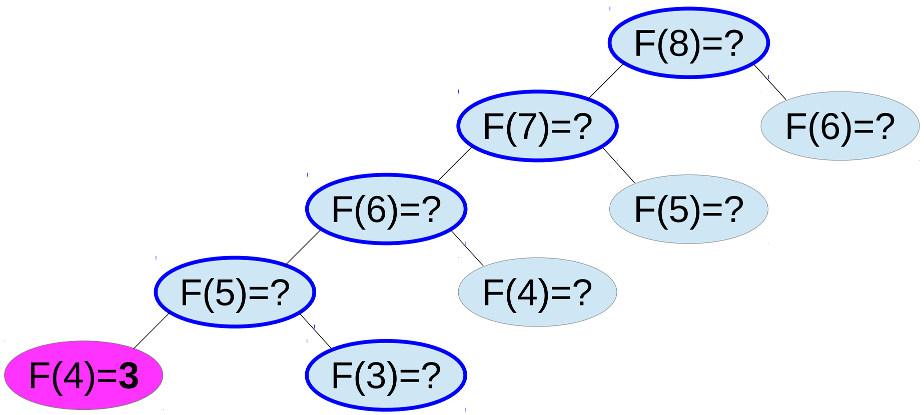
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3

Fibonacci sequence

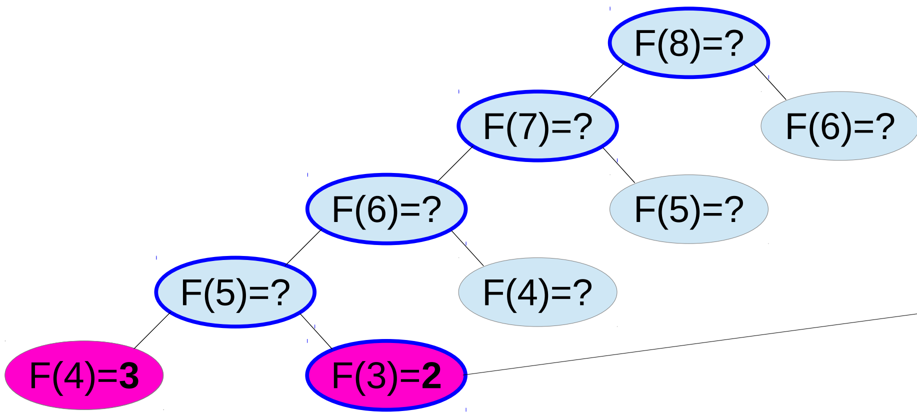
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3

Fibonacci sequence

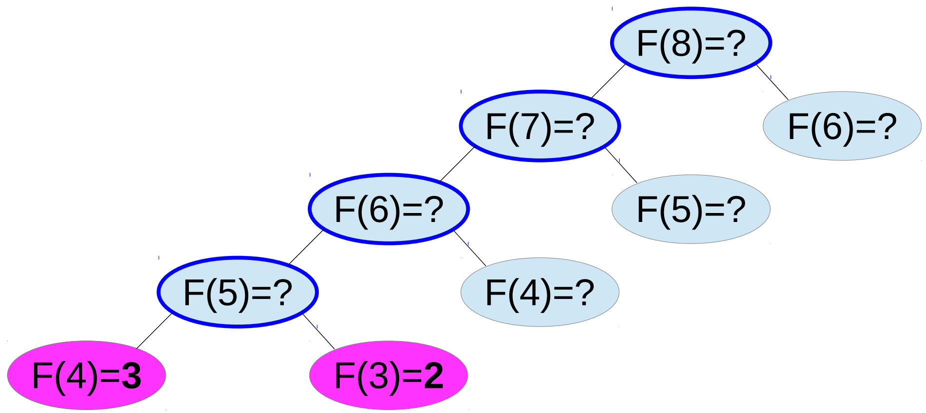
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3

Fibonacci sequence

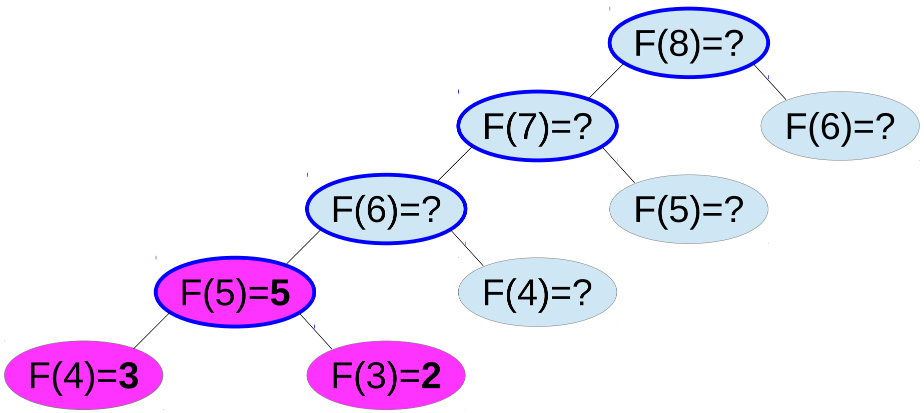
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3

Fibonacci sequence

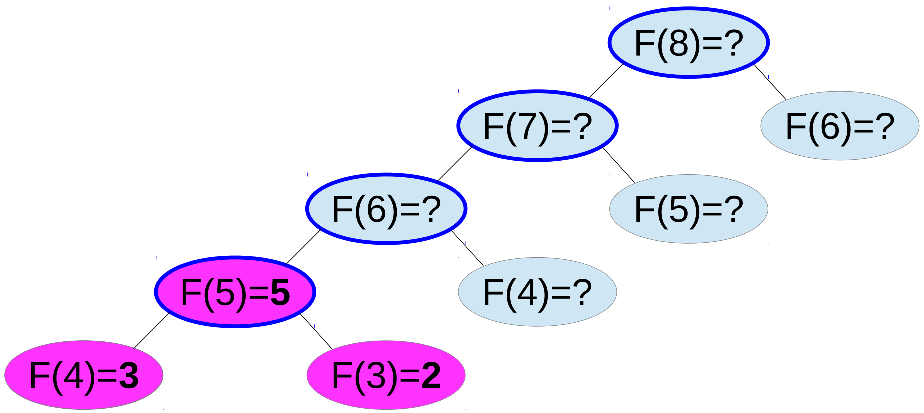
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3

Fibonacci sequence

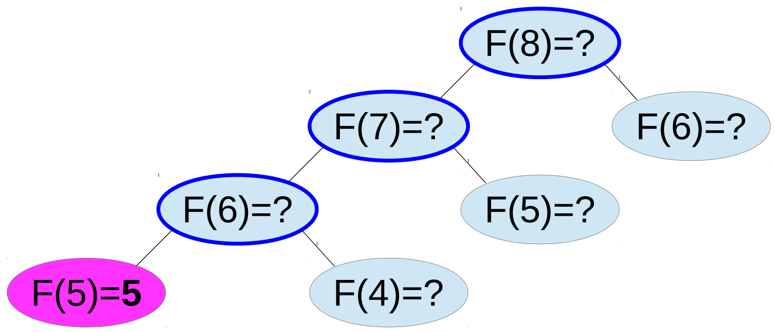
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5

Fibonacci sequence

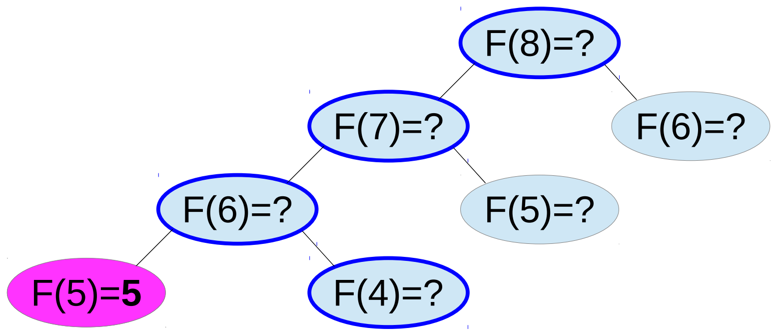
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5

Fibonacci sequence

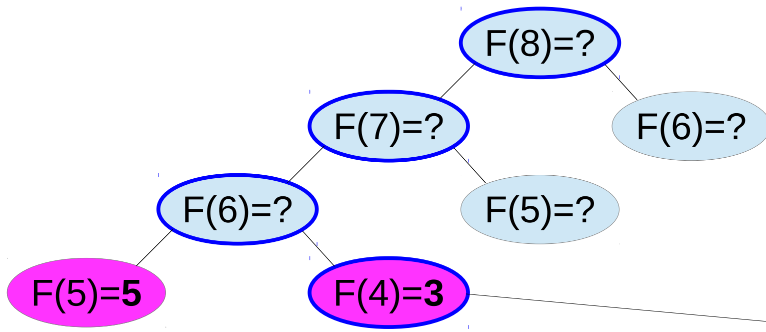
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5

Fibonacci sequence

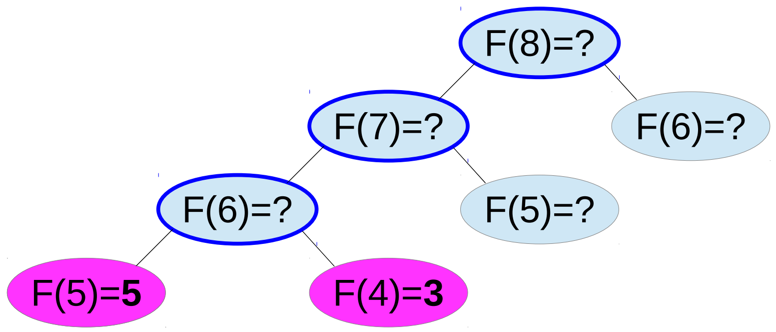
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5

Fibonacci sequence

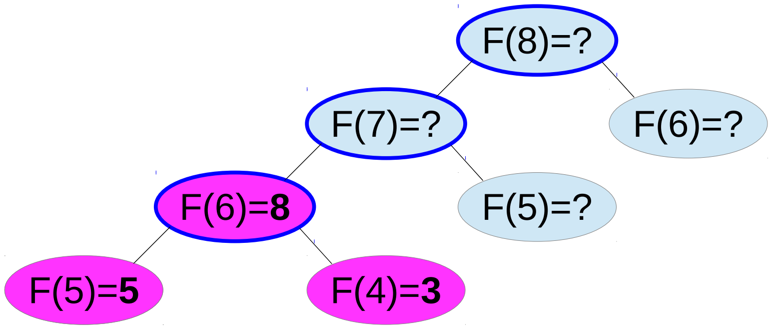
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5

Fibonacci sequence

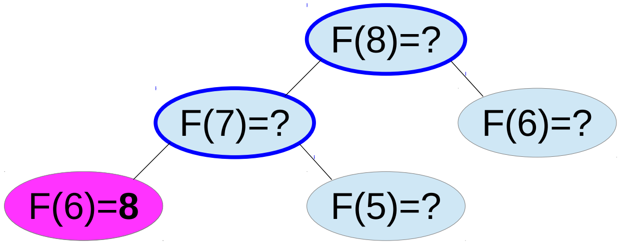
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8

Fibonacci sequence

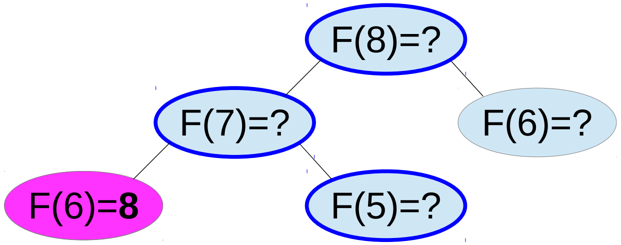
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8

Fibonacci sequence

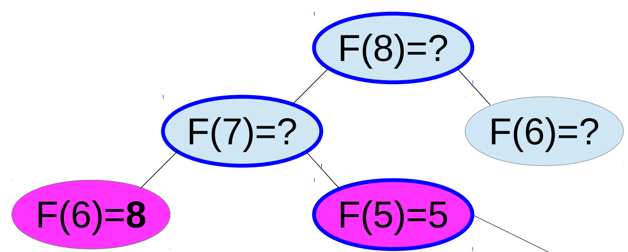
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8

Fibonacci sequence

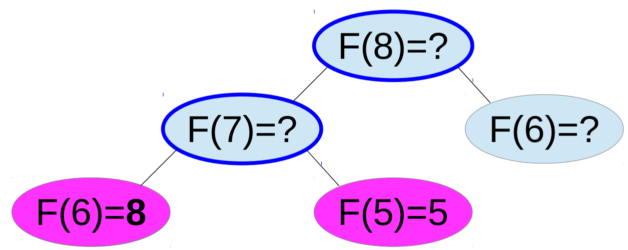
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8

Fibonacci sequence

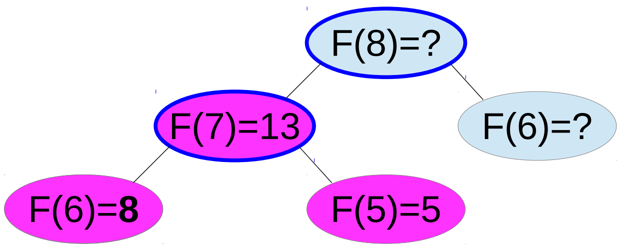
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8

Fibonacci sequence

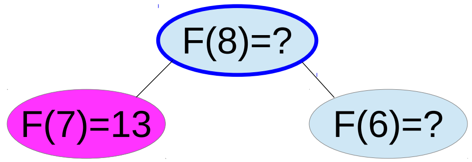
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8
7	F(7)=13

Fibonacci sequence

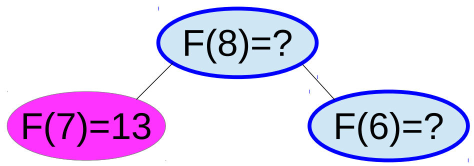
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8
7	F(7)=13

Fibonacci sequence

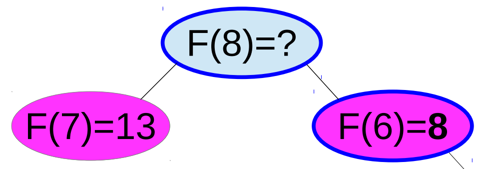
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8
7	F(7)=13

Fibonacci sequence

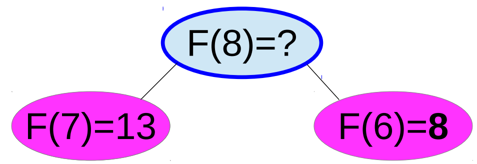
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8
7	F(7)=13

Fibonacci sequence

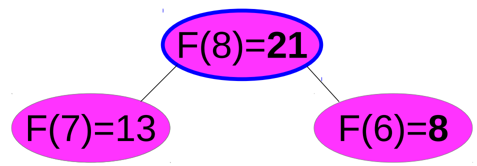
- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8
7	F(7)=13

Fibonacci sequence

- Top-down DP memoized sequence for F(8)



x	F(x)
3	F(3)=2
4	F(4)=3
5	F(5)=5
6	F(6)=8
7	F(7)=13

Fibonacci sequence

- These methods of computing the Fibonacci sequence use **recursion**
  - Referred to as **top-down**
  - “Find the answer for my problem”

Fibonacci sequence

- Another way of computing the Fibonacci sequence is by **iteratively building** the solution
  - Referred to as **bottom-up**
  - "Find the answers for all the sub problems, then find the answer for my problem"

Fibonacci sequence

- As a function:
    - $F(1) = 1$
    - $F(2) = 1$
    - $F(i) = F(i-1) + F(i-2)$ for $i = 3, 4, \ldots$

Fibonacci sequence

- As a table:
    - F = new int[9];
    - F[1] = 1
    - F[2] = 1
    - F[n] = F[n-1] + F[n-2]

Fibonacci sequence

- As a table:
  - F = new int[9];
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]

x	1	2	3	4	5	6	7	8
F[x]	1	1	2					

F[3] = F[2] + F[1]

Fibonacci sequence

- As a table:
  - F = new int[9];
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]

x	1	2	3	4	5	6	7	8
F[x]	1	1	2	3				

F[4] = F[3] + F[2]

Fibonacci sequence

- As a table:
  - F = new int[9];
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]

x	1	2	3	4	5	6	7	8
F[x]	1	1	2	3	5			

F[5] = F[4] + F[3]

Fibonacci sequence

- As a table:
  - F = new int[9];
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]

x	1	2	3	4	5	6	7	8
F[x]	1	1	2	3	5	8		

F[6] = F[5] + F[4]

Fibonacci sequence

- As a table:
  - F = new int[9];
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]

x	1	2	3	4	5	6	7	8
F[x]	1	1	2	3	5	8	13	

F[7] = F[6] + F[5]

Fibonacci sequence

- As a table:
  - F = new int[9];
  - F[1] = 1
  - F[2] = 1
  - F[n] = F[n-1] + F[n-2]

Here's F(8)!

x	1	2	3	4	5	6	7	8
F[x]	1	1	2	3	5	8	13	21

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3

- Given an array of numbers,
  - int a = new int[n];
- find the longest increasing subsequence
  - a subset s such that $a[s[0]] < a[s[1]] < a[s[2]] < \ldots < a[s[m-1]]$
  - And $|s|$ is maximal

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3

- Here's an increasing subsequence
    - Is it the longest increasing subsequence?
- To check, we can use bottom-up DP
    - Our algorithm will build the solution to all sub problems before finding the solution to the overall problem
    - Find the solution for a[0:1], then a[0:2], ..., then a[0:n]

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3

- Algorithm:
  - Build a table *s* so that *s*[*i*] is the longest increasing subsequence ending with *a*[*i*]
    - Includes *a*[*i*] in the sequence
  - $s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$

Builds on prior subsequences

Subsequence of itself

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]										

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1									

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2								

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2							

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1						

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2					

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3				

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3			

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	4		

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	4	3	

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	4	3	2

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	**4**	3	2

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

- Demo of the algorithm

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]										
p[i]										

$$s[i] = \max \begin{Bmatrix} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{Bmatrix}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1									
p[i]	-1									

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2								
p[i]	-1	0								

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2							
p[i]	-1	0	0							

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1						
p[i]	-1	0	0	-1						

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2					
p[i]	-1	0	0	-1	0					

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3				
p[i]	-1	0	0	-1	0	1				

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3			
p[i]	-1	0	0	-1	0	1	2			

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	4		
p[i]	-1	0	0	-1	0	1	2	5		

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	4	3	
p[i]	-1	0	0	-1	0	1	2	5	4	

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i\text{-}1 \\ 1 \end{array} \right\}$$

p[i] = j, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	4	3	2
p[i]	-1	0	0	-1	0	1	2	5	4	3

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	**4**	3	2
p[i]	-1	0	0	-1	0	1	2	**5**	4	3

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	**4**	3	2
p[i]	-1	0	0	-1	0	**1**	2	**5**	4	3

$$s[i] = \max \left\{ \begin{array}{l} s[j] + 1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	**4**	3	2
p[i]	-1	**0**	0	-1	0	**1**	2	**5**	4	3

$$s[i] = \max \left\{ \begin{array}{l} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{array} \right\}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

i	0	1	2	3	4	5	6	7	8	9
a[i]	4	15	11	2	7	19	15	20	9	3
s[i]	1	2	2	1	2	3	3	**4**	3	2
p[i]	**-1**	**0**	0	-1	0	**1**	2	**5**	4	3

$$s[i] = \max \begin{Bmatrix} s[j]+1 \text{ if } a[i] > a[j] \text{ for } j = 1, 2, \ldots, i-1 \\ 1 \end{Bmatrix}$$

$p[i] = j$, or -1 if no j exists

- Demo of the algorithm
  - Keeping track of the longest increasing subsequence

Longest increasing subsequence

```
void lisLength(int a[]) {
    int N = a.length;
    int LIS[] = new int[N];
    int backIdx[] = new int[N];

    for (int i = 0; i < N; i++) {
        LIS[i] = 1;
        backIdx[i] = -1;

        for (int j = 0; j < i; j++) {
            if (a[i] > a[j]) {
                LIS[i] = Math.max(LIS[i], LIS[j]+1);
                backIdx[i] = j;
            }
        }
    }

    int LISlength = 0;
    for (int i = 0; i < N; i++) LISlength = Math.max(LISlength, LIS[i]);
    return LISlength;
}
```

Wedding Shopping

- You are going shopping for a wedding
  - There are $1 <= C <= 20$ types of garments
  - Each garment type has $1 <= K <= 20$ items
    - e.g., 2 types of shirts, 3 different belts, etc.
  - You have a budget of $1 <= M <= 200$
  - Task: Buy one of each type of garment, spending as much money as possible without going over budget
  - What is the maximum possible amount to spend?

Wedding Shopping

M = 100	Item 0	1	2	3
Shirt	8	6	4	
Pants	5	10		
Belt	1	3	3	7
Shoes	50	14	23	8

Answer: 75

Wedding Shopping

Item

M = 100	0	1	2	3
Shirt	8	6	4	
Pants	5	10		
Belt	1	3	3	7
Shoes	50	14	23	8

Answer: 75

Item

M = 20	0	1	2	3
Shirt	4	6	8	
Pants	5	10		
Belt	1	3	5	5

Answer: 19
(Multiple solutions)

Wedding Shopping

Item

M = 5	**0**	**1**	**2**	**3**
Shirt	6	4	8	
Pants	10	6		
Belt	7	3	1	7

Answer: No solution!

Wedding Shopping

- How did we solve this problem last time?
  - Dynamic programming (top-down)
  - State: (money remaining, garment index)
  - Goal: **Find all reachable states**
    - i.e., find out how much money we have remaining when we have selected some number of garments
    - The answer will be the least amount of money remaining when we have selected all garments

Wedding Shopping

- Bottom-up DP formulation
  - Define a table **dp[moneyRem][*i*]** where *i* is the garment index
  - **dp[moneyRem][*i*]** is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

Wedding Shopping

Example:

Item

M = 12	0	1	2	3
Shirt (1)	6	4	8	
Pants (2)	10	6		
Belt (3)	7	3	1	7

dp[moneyRem][*i*] is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

moneyRem

dp:

i (garment index)

	0	1	2	3	4	5	6	7	8	9	10	11	12
0													
1													
2													
3													

Wedding Shopping

Example:

Item

M = 12	0	1	2	3
Shirt (1)	6	4	8	
Pants (2)	10	6		
Belt (3)	7	3	1	7

dp[moneyRem][*i*] is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

moneyRem

dp:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0													T
1													
2													
3													

i
(garment index)

Wedding Shopping

Example:

Item

M = 12	**0**	**1**	**2**	**3**
Shirt (1)	6	4	8	
Pants (2)	10	6		
Belt (3)	7	3	1	7

dp[moneyRem][*i*] is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

moneyRem

dp:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0													T
1					T		T		T				
2													
3													

i
(garment index)

Wedding Shopping

Example:

Item

M = 12	0	1	2	3
Shirt (1)	6	4	8	
Pants (2)	10	6		
Belt (3)	7	3	1	7

dp[moneyRem][*i*] is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

moneyRem

dp:

i (garment index)

	0	1	2	3	4	5	6	7	8	9	10	11	12
0													T
1					T		T		T				
2	T		T										
3													

Wedding Shopping

Example:

Item

M = 12	0	1	2	3
Shirt (1)	6	4	8	
Pants (2)	10	6		
Belt (3)	7	3	1	7

dp[moneyRem][*i*] is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

moneyRem

dp:

i (garment index)

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0													T
1					T		T		T				
2	T		T										
3		T											

Wedding Shopping

- Bottom-up DP formulation
  - Define a table dp[moneyRem][*i*] where *i* is the garment index
  - dp[moneyRem][*i*] is true if it's possible to end up with moneyRem by choosing garments 1 through *i*

Wedding Shopping

- Bottom-up DP formulation
  - dp[moneyRem][i]
    = OR (dp[moneyRem – price_j][i-1]
    for j = 0, 1, ..., | items_i |-1)
    - items_i refers to all the items for the ith garment

Wedding Shopping

- This works because we're simply marking **all possible states that are reachable**
- The answer to “is it possible to have *x* remaining on *m* garments?” is yes if and only if **dp[x][m]** is true

Wedding Shopping

- The answer to “what is the most one can spend on all *n* garments without going over budget?” is:
  - min(moneyRem for all **dp[moneyRem][n]** that are true)

Wedding Shopping

- Runtime analysis:
  - It only takes as long as it takes to fill up each of the states in the dp table
  - (Amount of money) * (number of garments) = 200 * 20 = 4000 state space
  - (Number of items) = 20 operations to fill each state
  - 4000 * 20 = 80,000 operations, small!

Let Me Count The Ways

- Given the coin system { 1, 5, 10, 25, 50 }, how many different combinations of coins can make *n* cents?

Let Me Count The Ways

- Sketch of how to do this:
  - Create a 2D int array so that **dp[*amount*][*i*]** contains the number of ways to make *amount* using coins 0 through *i*
    - coins = [1, 5, 10, 25, 50] (an array)
    - This ensures we are counting combinations and not permutations

Let Me Count The Ways

- Sketch of how to do this:
  - Create a 2D int array so that **dp[*amount*][*i*]** contains the number of ways to make *amount* using coins 0 through *i*

“pick zero coins”

$$\text{dp[amount][} i \text{]} = \text{dp[amount][} i\text{-1]} + \text{dp[amount – coins[} i \text{]][} i \text{]}$$

“pick another coin”

Let Me Count The Ways

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
50	1	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Base case

Result will be here

$$dp[\ amount\ ][\ i\ ] = dp[\ amount\ ][\ i\text{-}1\ ] + dp[\ amount - coins[\ i\ ]][\ i\ ]$$

Let Me Count The Ways

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
50	1	0	0	0	0	0	0	0	0	0	0	0	0	0
25	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Pick zero coins

$$\text{dp[amount][} i \text{]} = \text{dp[amount][} i\text{-1]} + \text{dp[amount – coins[} i \text{]][} i \text{]}$$

Let Me Count The Ways

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
50	1	0	0	0	0	0	0	0	0	0	0	0	0	0
25	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Pick zero coins

Pick another coin

$$\begin{aligned} dp[\ amount\ ][\ i\ ] = &\quad dp[\ amount\ ][\ i\text{-}1\ ] \\ &+ \ dp[\ amount - coins[\ i\ ]][\ i\ ] \end{aligned}$$

Let Me Count The Ways

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
50	1	0	0	0	0	0	0	0	0	0	0	0	0	0
25	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	1	0	0	0
5	1	0	0	0	0	1	0	0	0	0	2	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Pick zero coins AND pick another coin

$$\begin{aligned} dp[\ amount\ ][\ i\ ] = \quad & dp[\ amount\ ][\ i\text{-}1\ ] \\ + \quad & dp[\ amount - coins[\ i\ ]][\ i\ ] \end{aligned}$$

Let Me Count The Ways

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
50	1	0	0	0	0	0	0	0	0	0	0	0	0	0
25	1	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	0	0	0	0	0	0	0	0	0	1	0	0	0
5	1	0	0	0	0	1	0	0	0	0	2	0	0	0
1	1	1	1	1	1	2	2	2	2	2	4	4	4	4

dp[amount][*i*] = dp[amount][*i*-1]
+ dp[amount – coins[*i*]][*i*]

Let Me Count The Ways

- Runtime analysis:
  - 5 types of coins, given values are at most 30,000 (as per problem statement)
  - O(1) to compute each cell
  - 5 * 30,000 = 150,000 operations, cool!

Dynamic Programming

- Top-down DP
  - Pro: Natural transformation from recursion
  - Pro: Computes subproblems only when necessary
  - Con: May be slower due to recursion overhead
  - Con: Uses exactly O(states) table size
- Bottom-up DP
  - Pro: Faster if many sub-problems visited, no recursion
  - Pro: Can save memory space
  - Con: May not be as intuitive
  - Con: Fills values for all the states, does not skip unreachable states

Book reference

- Readings:
  - Section 3.5

Vacation

- UVa 10192
- "Longest common subsequence"
  - You are given two strings, S_1 and S_2
  - dp[*i*][*j*] is the length of the longest common subsequence after *i* characters of S_1 and *j* characters of S_2
  - $$dp[\,i\,][\,j\,] = \max(\,dp[\,i-1\,][\,j\,],\; dp[\,i\,][\,j-1\,],\; dp[\,i-1\,][\,j-1\,] + 1 \text{ if } S_1[\,i\,] = S_2[\,j\,]\,)$$

Vacation

- Example:
  - abcd and acdb

	_	a	b	c	d
_	0	0	0	0	0
a	0	1	1	1	1
c	0	1	1	2	2
d	0	1	1	2	3
b	0	1	1	2	3

$$dp[\,i\,][\,j\,] = \max(\; dp[\,i-1\,][\,j\,],\; dp[\,i\,][\,j-1\,],\; dp[\,i-1\,][\,j-1\,] + 1 \text{ if } S_1[\,i\,] = S_2[\,j\,]\;)$$

Vacation

- Why this works (sketch):
  - dp[i][j] contains the largest common subsequence between S_1[0 : i] and S_2[0 : j]
  - At each step, “consume” a character from either string, incrementally build upon the best answer
    - If possible (and if the answer is better), “consume” a character from both and increment the subsolution
  - Therefore the largest common subsequence is in dp[n][m] where $|S_1| = n$ and $|S_2| = m$

Is Bigger Smarter?

- UVa 10131
  - Read it, then try to solve it! Hint in 5 minutes.
- Hint: Sort the elephants by weights increasing, if there's a tie then by IQ decreasing
  - This reduces the problem down to longest increasing subsequence
    - Discuss why?

Is Bigger Smarter?

6008 1300
6000 2100
500 2000
1000 4000
1100 3000
6000 2000
8000 1400
6000 1200
2000 1900

$\xrightarrow[\text{Then by IQ}]{\text{Sort by weight}}$

500 2000
1000 4000
1100 3000
2000 1900
6000 2100
6000 2000
6000 1200
6008 1300
8000 1400

$\xrightarrow{\text{Longest decreasing subsequence}}$ **Answer!**