

Dynamic Programming 1

Dynamic Programming

Problems we will look at

1. Rod cutting problem
2. Grid cost problem
3. Change making problem
4. Edit distance problem

Dynamic Programming

- First, don't get confused by the name “dynamic programming”.
- Think of it as a technique where u avoid repeated computation by storing values that have already been computed.
- Naïve example (non computer science)
 - If $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$
 - Then how much is $100 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$?
 - To answer, u would add 100 to the earlier computed result of 55 ... u would not add all of these again.

Dynamic Programming

- First, don't get confused by the name “dynamic programming”.
- Think of it as a technique where u avoid repeated computation by storing values that have already been computed.
- Naïve example (non computer science)
 - If $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$
 - Then how much is $100 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$?
 - To answer, u would add 100 to the earlier computed result of 55 ... u would not add all of these again.
- Another example:
 - Compute $2 * 394$.
 - Unless u r a math whiz, u may take a few seconds to do this mentally.
 - But if u r given: $2 * 393$ is 786, then u would simply add 2 to it to get $2 * 394 = 788$.

Dynamic Programming

- They say the following about dynamic programming:
 - Those who cannot remember the past are condemned to repeat it 😊
- Now, where have we use dynamic programming earlier?

Dynamic Programming

Dynamic programming is used to solve *optimization* and *combinatorial* problems:

- the former means that u need to find a solution for optimizing some value,
 - and the latter is the number of ways of doing something.
-
- DP is an optimization technique:
 - cache results that are used later (instead of recomputing).
 - And hence improve run time.

Dynamic Programming

- DP criteria

1. Optimal substructure

- It means that the solution is such that we can use the results of smaller sub problems to solve the overall problem.

Dynamic Programming

- DP criteria

1. Optimal substructure

- It means that the solution is such that we can use the results of smaller sub problems to solve the overall problem.
- Or, put slightly differently:
 - Optimal solution to overall problem can be obtained from the optimal solution of the sub problems.

Dynamic Programming

- DP criteria

1. Optimal substructure

- It means that the solution is such that we can use the results of smaller sub problems to solve the overall problem.
 - Or, put slightly differently:
 - Optimal solution to overall problem can be obtained from the optimal solution of the sub problems.
-
- Example: Finding shortest path in a graph.
 - Lets say we take the shortest path from A to C.
 - Let B be a vertex on that path.
 - The shortest path from A to C will contain the shortest path from A to B and the shortest path from B to C.
 - These are the sub problems for which we use the optimal solution for the overall solution.

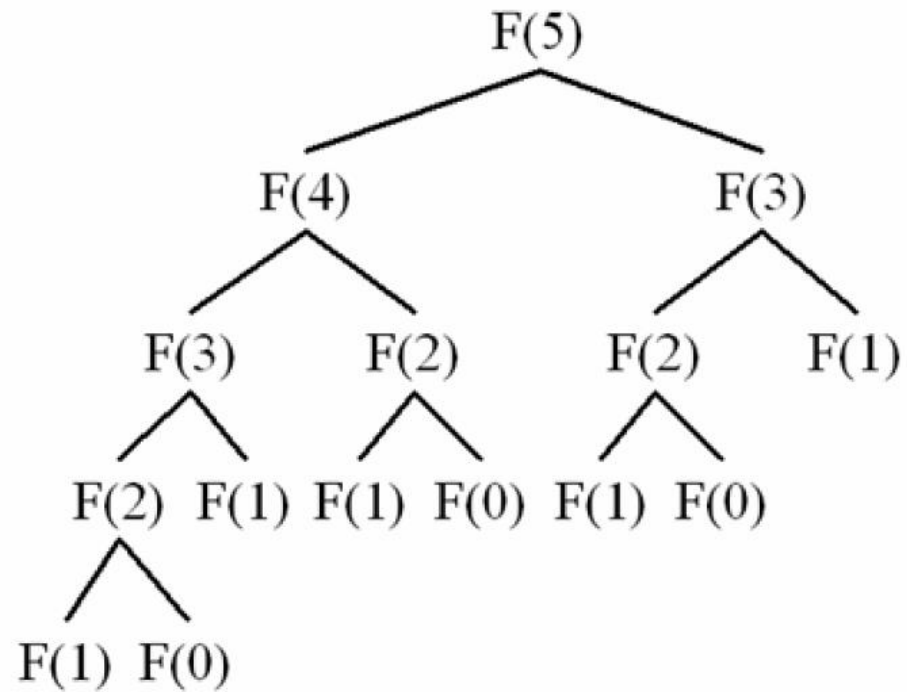
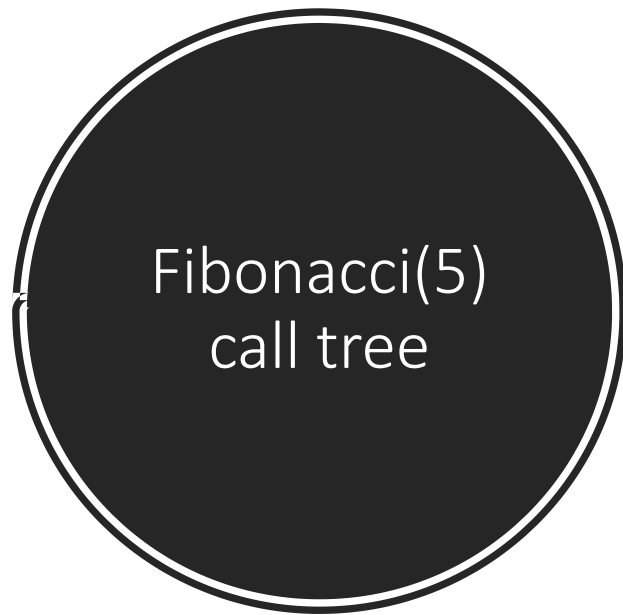
 - Floyd-Warshall and Bellman-Ford are shortest path algorithms where DP is used.

Dynamic Programming

- DP criteria

2. Overlapping sub problems

- The solution from the sub problems are cached, and we reuse these rather than recompute.
- By overlapping we mean that the sub problems show up many times (and hence are cached).
 - So, the total number of sub problems may be, say, exponential, but the total number of distinct sub problems should be much less (polynomial in input size).
- We saw Fibonacci as an example.
- Upcoming soon:
 - Compare with divide and conquer, where there are no overlapping sub problems.
- But first lets revisit Fibonacci



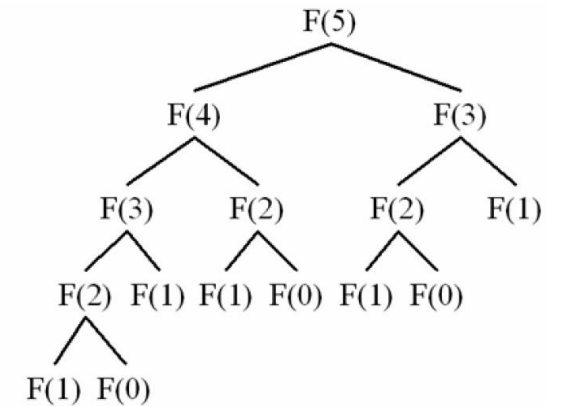
When computing Fibonacci(5), we have

- Two calls to Fibonacci(3)
- Three calls to Fibonacci(2)

Fibonacci

- There are a lot of repeated calls for the same value of N.
 - We will revisit this fact in a few slides.
- How many nodes do we have in a Fibonacci call tree:

Level	Nodes in level
0	1
1	2
3	8
4	16
N	2^N



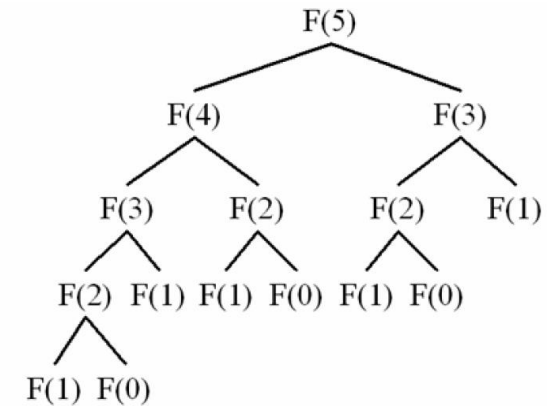
- **Note:** When we get towards the bottom, not all levels will be completely filled with nodes.
- Now, we know that Fibonacci(N) will have an N-level call tree.
- So, in general, we can say that a level N tree will have 2^N nodes.

So, naïve Fibonacci algorithm has a time complexity of ?

Fibonacci

- There are a lot of repeated calls for the same value of N.
 - We will revisit this fact in a few slides.
- How many nodes do we have in a Fibonacci call tree:

Level	Nodes in level
0	1
1	2
3	8
4	16
N	2^N



- **Note:** When we get towards the bottom, not all levels will be completely filled with nodes.
- Now, we know that Fibonacci(N) will have an N-level call tree.
- So, in general, we can say that a level N tree will have 2^N nodes.

So, naïve Fibonacci algorithm has a time complexity of $O(2^N)$

- Exponential algorithm ☹️... and as we saw before, that is not great.
- We had seen that memoization helps reduce the complexity.

Dynamic Programming

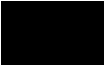
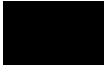

- Now, let's compare the “overlapping subproblems” with *Divide and conquer* approach.
 - We had used that in Quicksort and Mergesort.
- In that, the divisions are such that there are no overlaps.
 - And since no overlaps, no caching of solutions for sub problems.
 - If we are not going to reuse, no point caching.
 - The number of divisions (sub problems) typically increases exponentially with each step, but their size decreases exponentially.
- Each division is “solved” and then put together...forms the overall solution.

Dynamic Programming

Some examples of dynamic programming:

1. Fibonacci
2. Rod cutting problem
3. Floyd-Warshall (find the shortest path between all pairs of vertices in a directed graph)
4. Bellman-Ford (find the shortest path between a given vertex to all other vertices in a directed graph)
5. Coin change problem

Dynamic Programming

1. In case of Fibonacci:
 1. To start with, we were given $F(0)$ and $F(1)$.
 2. We first computed $F(2)$, which used $F(0)$ and $F(1)$
 3. Then computed $F(3)$, which used the results from $F(2)$ and $F(1)$
2. To improve run time, we used *memorization* where we added a cache to the recursive solution.
 1. This was the *top down* approach (more on this on next slide)
 2.  time and  space.
3. We also looked at an iterative solution where we kept track of just the last two Fib numbers.
 1. This was the *bottom up* approach. (more on this on next slide)
 2.  time and $O(1)$ space.

Dynamic Programming

1. In case of Fibonacci:
 1. To start with, we were given $F(0)$ and $F(1)$.
 2. We first computed $F(2)$, which used $F(0)$ and $F(1)$
 3. Then computed $F(3)$, which used the results from $F(2)$ and $F(1)$
2. To improve run time, we used *memorization* where we added a cache to the recursive solution.
 1. This was the *top down* approach (more on this on next slide)
 2. $O(N)$ time and $O(N)$ space.
3. We also looked at an iterative solution where we kept track of just the last two Fib numbers.
 1. This was the *bottom up* approach. (more on this on next slide)
 2. $O(N)$ time and $O(1)$ space.

Dynamic Programming

Implementing DP solutions:

1. Top down
 1. Recursive.
 2. Not as efficient as iterative.
 3. Can be easier to implement
 4. Aka *memorization*
5. Think of it this way:
 1. In recursion, we start at the top value and go down towards the base case... top → down

Dynamic Programming

Implementing DP solutions:

2. Bottom up
 1. Iterative
 2. More efficient
 3. Solve for smaller problem first, then reuse that for figuring out the bigger problem.
 4. Aka *tabulation*.

5. Think of it this way:
 1. In iteration, we start at a small value, and go towards higher values ... bottom → top
 2. E.g.:
 1. for $i = 0$ to N // start at 0 and go up to N

Dynamic Programming

Implementing DP solutions (writing down both on one slide):

1. Top down
 1. Recursive.
 2. Not as efficient as iterative.
 3. Can be easier to implement
 4. Aka *memoization*
2. Bottom up
 1. Iterative
 2. More efficient
 3. Solve for smaller problem first, then reuse that for figuring out the bigger problem.
 4. Aka *tabulation*.

Backtracking vs DP

Similarities: look for a solution by searching the space.

Difference: Backtracking eliminates parts of the search space that won't lead to a solution.

- Backtracking examples:
 - N queen problem
 - Word board game

Fibonacci – storing all intermediate values

Fibonacci iterative where we store compute results of all Fibonacci numbers:
This technique falls under DP.

```
fib : array size N
```

```
fib [ 0 ] = 0
```

```
fib [ 1 ] = 1
```

```
while i <= N
```

```
    fib [ i ] = fib [ i - 1 ] + fib [ i - 2 ];
```

```
return fib [ N ] ;
```

If we do it this way, space used is $O (N)$.

Fibonacci – storing all intermediate values

Fibonacci iterative where we store compute results of **only last two** Fibonacci numbers:
This technique also falls under DP.

```
ulong    fibN_1 = 0;    // initialized to Fib(0), which is 0. Will eventually contain Fib(N-1) value
ulong    fibN_2 = 1;    // initialized to Fib(1), which is 1. Will eventually contain Fib(N-2) value
ulong    fibN = 0;      // Will eventually contain Fib(N) value
```

```
while ( N > 0 )
    fibN = fibN_1 + fibN_2;

    // get ready for next iteration of the loop (if there is one)
    fibN_2 = fibN_1;
    fibN_1 = fibN;
    --N;
```

```
return fibN
```

Space used is $O(1)$.

Rod Cutting problem

Next, we will look at the rod cutting problem and solve it using DP

- Given:
 - a rod of length N
 - a *prices* array that contains the price of a rod of length i .
- To do:
 - Cut the rod in a way that you can sell it for maximum profit.

Example:

- U r given a rod of length 8
- Price array { 0, 1, 6, 2, 4, 3, 3, 2, 8 }
- If u sell the rod in one piece of length 8, u make \$8.
- If u sell the rod in 8 pieces of length 1 each, u make \$8.
- If u sell the rod in 4 pieces of length 2 each, u make \$24.
- Next, lets look at how we would approach this problem.

Rod cutting

Given a rod of length 8.

Max price from rod of length 8

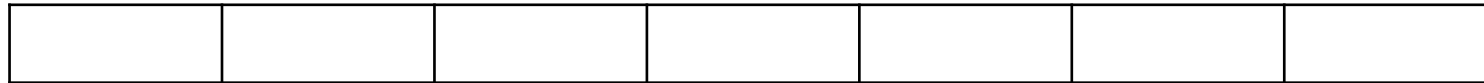


Either make a cut of size 1

prices [1] +



Max price from rod of length 7



Or make a cut of size 2

prices [2] +



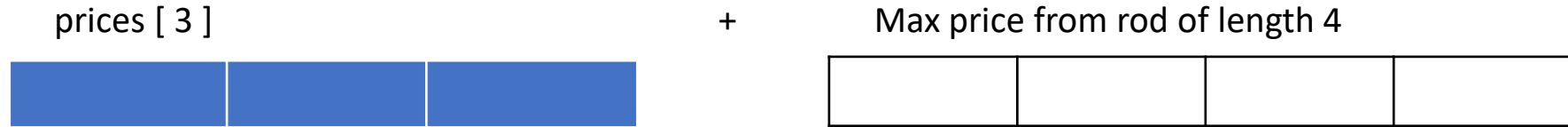
Max price from rod of length 6



Next slide

Rod cutting

OR make a cut of size 3



OR...

make a cut of size 7



OR make a cut of size of 0



Rod cutting

Repeating from previous slide:

prices [3] +



Max value from rod of length 4



OR...

prices [7] +



Max value from rod of length 1



OR

prices [8] +



Max value from rod of length 0 (which is 0).

NOTE: In this case, u r not making any cut

What pattern do u see above?

See next slide

Rod cutting

Repeating from prev slide:

prices [3] +



Max value from rod of length 4



OR...

prices [7] +



Max value from rod of length 1



OR

prices [8] +



Max value from rod of length 0 (which is 0).

NOTE: In this case, u r not making any cut

What pattern do u see above?

Looks like we want the **max value** from the equation below for all possible sizes of *cutRod*

prices [*cutRod*] + Max value from the remaining rod of length $N - \text{cutRod}$

Now, see next slide for a more complete equation...

Rod cutting

What pattern do u see here?

Looks like we want the max value from the equation below for all possible sizes of *cutRod*

$$\text{prices}[\text{cutRod}] + \text{Max value from the remaining rod of length } N - \text{cutRod}$$

OR, put differently, we want to get

$$\text{MAX} (\text{prices}[\text{cutRod}] + \text{Max value from the remaining rod of length } N - \text{cutRod})$$

for all possible values of *cutRod*

... which would be from 1 to N

meaning, from a rod of length N:

the smallest piece we can cut is of size 1, and

the biggest piece would be N ... which means no cut

Rod cutting

So, our equation is

MaxValue from rod length N = MAX (prices [cutRod] + Max value from the remaining rod of length N – cutRod)

For cutRod values from 1 to N

```
MaxValue ( N )  
    for cutRod = 1 to N  
        MAX ( prices [ cutRod ] + MaxValue ( N – cutRod ) )
```

But we need to keep track of the max value obtained at each step, so we really want:

```
MaxValue ( N )  
    maxValueSoFar = -1;           // local variable  
  
    for cutRod = 1 to N  
        maxValueSoFar = Math.Max ( prices [ cutRod ] + MaxValue ( N – cutRod ),  
                                   maxValueSoFar )
```

What obvious thing are we missing in the recursion above?

Rod cutting

```
MaxValue ( N )  
    if ( N <= 0 )  
        return 0;  
  
    maxValueSoFar = -1;  
    for cutRod = 1 to N  
        maxValueSoFar = MAX ( prices [ cutRod ] + MaxValue ( N – cutRod ),  
                               maxValueSoFar )  
  
    return maxValueSoFar
```

// We will add a base case to the recursion:

// We could add a counter to count calls to MaxValue... ++ callCount

We see repeated calls ... so, quite a few of these are being recomputed... we had seen this repetition in naïve Fibonacci

MaxValue (5) calls MaxValue (4) and MaxValue (3) and MaxValue (2) and MaxValue (1) and MaxValue (0)
MaxValue (4) calls MaxValue (3) and MaxValue (2) and MaxValue (1) and MaxValue (0)
MaxValue (3) calls MaxValue (2) and MaxValue (1) and MaxValue (0)
MaxValue (2) calls MaxValue (1) and MaxValue (0)
MaxValue (1) calls MaxValue (0)

Time complexity: ?

Overlapping subproblems: We see that here (all the colors I added above 😊).

Optimal substructure: Optimal solution to overall problem can be obtained from the optimal solution of the sub problems

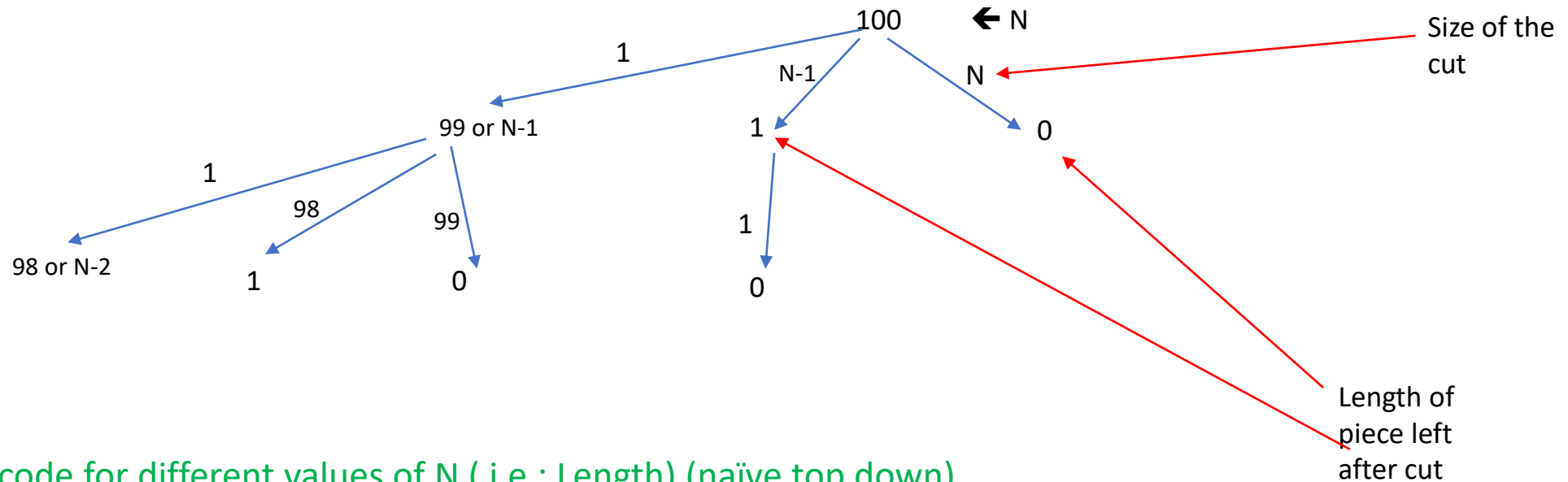
So, this implies we can / should use Dynamic programming (DP) to solve this.

Rod cutting

How many nodes do we have at each level.

Assume N is big enough.

At every level, one of the paths stops (rightmost in the diagram)



Running the code for different values of N (i.e.: Length) (naïve top down)

Length: 15

Call count: 16,384

Length: 30

Call count: 536,870,912

Rod cutting - Using Dynamic Programming

Now, in DP, we would want to prevent repeated computes of the overlapping subproblems.

Top-down approach using DP (**memoization**)

```
MaxValue ( N )
    if ( N <= 0 )
        return 0;

    // We could add a counter to count calls to MaxValue... ++ callCount

    if ( cache[ N ] >= 0 )
        return cache [ N ]

    // We could add another counter to count number of computes... ++ computeCount

    maxValueSoFar = -1;
    for cutRod = 1 to N
        maxValueSoFar = MAX ( prices [ cutRod ] + MaxValue ( N – cutRod ),
                               maxValueSoFar )

    cache [ N ] = maxValueSoFar
    return maxValueSoFar
```

Rod cutting - Using Dynamic Programming

Now, in DP, we would want to prevent repeated computes of the overlapping subproblems.

Top-down approach using DP (**memoization**)

```
MaxValue ( N )
    if ( N <= 0 )
        return 0;

    // We could add a counter to count calls to MaxValue... ++ callCount

    if ( cache[ N ] >= 0 )
        return cache [ N ]

    // We could add another counter to count number of computes... ++ computeCount

    maxValueSoFar = -1;
    for cutRod = 1 to N
        maxValueSoFar = MAX ( prices [ cutRod ] + MaxValue ( N – cutRod ),
                               maxValueSoFar )

    cache [ N ] = maxValueSoFar
    return maxValueSoFar
```

Rod cutting - Using Dynamic Programming

Bottom-up approach using DP (**tabulation**)

FindMaxValue (N)

cache [0] = 0

for rodLen = 1 to N

cache [rodLen] = **FindMaxValueForLen** (rodLen, cache) ← See this function on next slide

return cache [N];

Rod cutting

FindMaxValueForLen (rodLen, cache)

```
maxValue = -1           // since our value will never be negative.
```

```
for cut = 1 to rodLen  
    maxValue = Math.Max ( prices [ cut ] + cache [rodLen - cut ],  
                        maxValue );
```

```
return  maxValue
```

```
// For computing cache [ 1 ]:
```

```
// cut = 1, rodLen = 1,  prices[ 1 ]  + cache [ 1 - 1 ]
```

```
// cache [ 0 ] already known ( was initialized to 0)
```

```
// For computing cache [ 2 ]:
```

```
// cut = 1, rodLen = 2,  prices[ 1 ]  + cache [ 2 - 1 ]
```

```
// cache [ 1 ] already known
```

```
// cut = 2, rodLen = 2,  prices[ 2 ]  + cache [ 2 - 2 ]
```

```
// cache [ 0 ] already known
```

```
// For computing cache [ 3 ]:
```

```
// cut = 1, rodLen = 3,  prices[ 1 ]  + cache [ 3 - 1 ]
```

```
// cache [ 2 ] already known
```

```
// cut = 2, rodLen = 3,  prices[ 2 ]  + cache [ 3 - 2 ]
```

```
// cache [ 1 ] already known
```

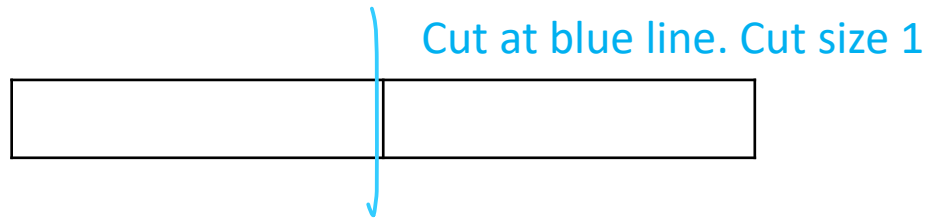
```
// cut = 3, rodLen = 3,  prices[ 3 ]  + cache [ 3 - 3 ]
```

```
// cache [ 0 ] already known
```

Rod cutting

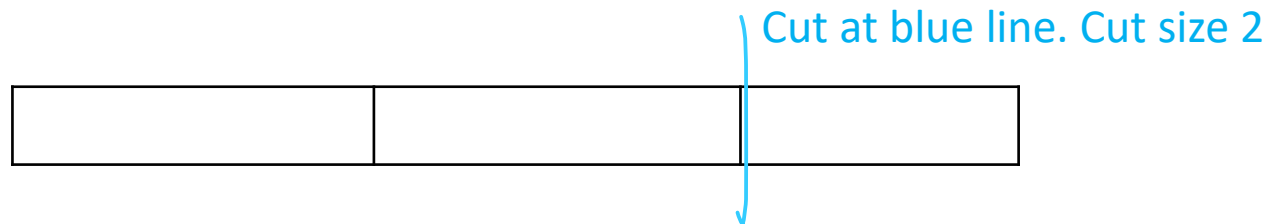
// For computing cache [2]:

// cut = 1, rodLen = 2, cache [2 - 1] + prices [1] // cache [1] already known



// For computing cache [3]:

// cut = 2, rodLen = 3, cache [3 - 2] + prices [2] // cache [1] already known



Rod cutting -- All code on one slide

Time complexity: ?

Space complexity: ?

See next slide for all code inside one function, in case u prefer it that way 😊

FindMaxValue (prices, N)

cache [0] = 0

for rodLen = 1 to N

cache [rodLen] = **FindMaxValueForLen** (prices, rodLen)

return cache [N];

FindMaxValueForLen (prices, rodLen)

maxValue = -1 // since our value will never be negative.

for cut = 1 to rodLen

maxValue = **Max** (maxValue, cache [rodLen – cut] + prices [cut])

return maxValue

Rod cutting -- not separating into another function

FindMaxValue (prices, N)

cache [0] = 0

for rodLen = 1 to N

maxValue = -1

for cut = 1 to rodLen

maxValue = **Max** (maxValue, cache [rodLen – cut] + prices[cut])

cache [rodLen] = maxValue

return cache [N];

Rod cutting - LAB 1 of 1

U r given a rod of length 8

Price array { 0, 1, 6, 2, 4, 3, 3, 2, 8 }

Write the functions below that return the maximum value for selling a rod of length N.

Once u have tested ur functions, u can call them in a loop for N from 1 to 8.

1. FindMaxValueTopDown(N)
2. FindMaxValueTopDownMemoization(N)
3. FindMaxValueBottomUp(N)

Example: First u can test like the following:

FindMaxValueTopDown (2) or FindMaxValueTopDown(3) or FindMaxValueTopDownMemoization(2) , etc.

Then u can call in a loop:

for len = 1 to 8

FindMaxValueTopDown (len) // or FindMaxValueTopDownMemoization(len) or FindMaxValueBottomUp(len)

Rod cutting

Here's a leetcode [problem](#) for some practice if u want.

Min cost path in a grid

Min cost path in a grid

Let's look at another problem.

- We are given a 2-dimensional grid.
- Each cell in the grid has a cost which represents the cost to be in the cell.
- The path to that cell (say, starting from top left of the grid) is obtained by
 - Traversing either downwards or
 - Traversing to the right.
- The cost of the path to a cell is the sum of costs of all cells in the path.
- Cost of **red** path below: 12
- Cost of **blue** path below: 14
- Cost of **yellow** path below: 10 (cheapest)
- **Goal:** Find cheapest path from starting cell (**top left**) to a destination cell (say, **bottom right**)

0	2	5
4	1	3
7	5	4

Min cost path in a grid

Now, our destination cell is the bottom right cell

Lets look at how we might arrive at that:

0	2	5
4	1	3
7	5	4

Option 1:

From the cell above

Min cost path in a grid

Now, our destination cell is the bottom right cell

Lets look at how we might arrive at that:

0	2	5
4	1	3
7	5	4

Option 1:

From the cell above (3)

Option 2:

From the cell to its left (5)

And how could we have arrived at the cells that are above(3) and to the left(5) of destination cell?

See next slide

Min cost path in a grid

Now, our destination cell is the bottom right cell

Lets look at how we might arrive at that:

0	2	5
4	1	3
7	5	4

Option 1:

From the cell above

Option 2:

From the cell to its left

And how could we have arrived at the cells that are above and to the left of destination cell?

For cell above(3), we could have come from one of its(3's) **two neighbors**.

And for the cell to the left(5)... next slide

Min cost path in a grid

0	2	5
4	1	3
7	5	4

The diagram shows a 3x3 grid with the following values:

0	2	5
4	1	3
7	5	4

Arrows indicate a path from the start cell (0) to the end cell (4):

- From (0,0) to (0,1) (vertical arrow)
- From (0,1) to (1,1) (horizontal arrow)
- From (1,1) to (1,0) (vertical arrow)
- From (1,0) to (2,0) (horizontal arrow)
- From (2,0) to (2,1) (vertical arrow)
- From (2,1) to (2,2) (vertical arrow)

So it looks like every cell has two possibilities in terms of the path that was taken to arrive at that cell.

What does that hint in terms of the possible paths ... or the time complexity?

Min cost path in a grid

Exponential time complexity

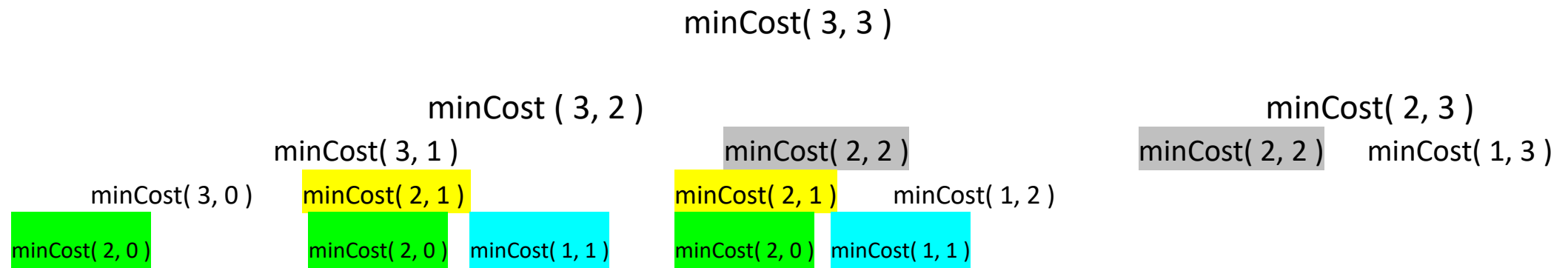
0	2	5
4	1	3
7	5	4

```
int FindMinCost( row, col )  
  
    if ( we are at cell 0 , 0 )  
    {  
        return cost[0, 0];  
    }  
  
    int comingFromLeft = FindMinCost( row, col - 1 );  
    int comingFromAbove = FindMinCost( row - 1, col );  
  
    // Cost of arriving at cell row, col is  
    // sum of cost[ row, col ] and minimum of arrival from left and arrival from above  
    return cost [ row, col ] + min (comingFromLeft, comingFromAbove);
```


Min cost path in a grid

How can we do better than exponential time complexity?

We can look at DP to help us out.
Let's look at that on next slide.



Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

First let's only look at the first column (col = 0):

To get to the cells in column 0, the cost can be said to be:

$$\text{Cost of path to } [R, 0] = \text{Cost of path to } [R-1, 0] + \text{Cost}[R, 0]$$

where R is the row.

So,

cost of path to [1,0] = cost of path to [0, 0] + Cost[1, 0]

Or,

put differently, we can say:

$$\text{PathCost}[1, 0] = \text{PathCost}[0, 0] + \text{Cost}[1, 0] = 1 + 1 = 2$$

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1			
1	1 + 1			
2				
3				

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

First let's only look at the first column (col = 0):

To get to the cells in column 0, the cost can be said to be:

$$\text{Cost of path to } [R, 0] = \text{Cost of path to } [R-1, 0] + \text{Cost}[R, 0]$$

where R is the row.

So,

$$\text{cost of path to } [1, 0] = \text{cost of path to } [0, 0] + \text{Cost}[1, 0]$$

Or,

put differently, we can say:

$$\text{PathCost}[1, 0] = \text{PathCost}[0, 0] + \text{Cost}[1, 0] = 1 + 1 = 2$$

$$\text{PathCost}[2, 0] = \text{PathCost}[1, 0] + \text{Cost}[2, 0] = 2 + 5 = 7$$

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1			
1	1 + 1			
2	2 + 5			
3				

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

First let's only look at the first column (col = 0):

To get to the cells in column 0, the cost can be said to be:

$$\text{Cost of path to } [R, 0] = \text{Cost of path to } [R-1, 0] + \text{Cost}[R, 0]$$

where R is the row.

So,

$$\text{cost of path to } [1,0] = \text{cost of path to } [0, 0] + \text{Cost}[1, 0]$$

Or,

put differently, we can say:

$$\text{PathCost}[1, 0] = \text{PathCost}[0, 0] + \text{Cost}[1, 0] = 1 + 1 = 2$$

$$\text{PathCost}[2, 0] = \text{PathCost}[1, 0] + \text{Cost}[2, 0] = 2 + 5 = 7$$

$$\text{PathCost}[3, 0] = \text{PathCost}[2, 0] + \text{Cost}[3, 0] = 7 + 2 = 9$$

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1			
1	1 + 1			
2	2 + 5			
3	7 + 2			

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

Look at the first row (row= 0):

To get to the cells in row 0, the cost can be said to be:

$$\text{Cost of path to } [0, C] = \text{Cost of path to } [0, C-1] + \text{Cost}[0, C]$$

where C is the col

So,

$$\text{PathCost}[0, 1] = \text{PathCost}[0, 0] + \text{Cost}[0, 1] = 1 + 2 = 3$$

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1	1 + 2		
1	2			
2	7			
3	9			

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

Look at the first row (row= 0):

To get to the cells in row 0, the cost can be said to be:

$$\text{Cost of path to } [0, C] = \text{Cost of path to } [0, C-1] + \text{Cost}[0, C]$$

where C is the col

So,

$$\text{PathCost}[0, 1] = \text{PathCost}[0, 0] + \text{Cost}[0, 1] = 1 + 2 = 3$$

$$\text{PathCost}[0, 2] = \text{PathCost}[0, 1] + \text{Cost}[0, 2] = 3 + 5 = 8$$

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1	1 + 2	3 + 5	
1				
2				
3				

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

Look at the first row (row= 0):

To get to the cells in row 0, the cost can be said to be:

$$\text{Cost of path to } [0, C] = \text{Cost of path to } [0, C-1] + \text{Cost}[0, C]$$

where C is the col

So,

$$\begin{aligned}\text{PathCost}[0, 1] &= \text{PathCost}[0, 0] + \text{Cost}[0, 1] = 1 + 2 = 3 \\ \text{PathCost}[0, 2] &= \text{PathCost}[0, 1] + \text{Cost}[0, 2] = 3 + 5 = 8 \\ \text{PathCost}[0, 3] &= \text{PathCost}[0, 2] + \text{Cost}[0, 3] = 8 + 7 = 15\end{aligned}$$

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1	1 + 2	3 + 5	8 + 7
1				
2				
3				

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

We initialized the first column and first row of MinCost.

We can write pseudocode to compute the cost in other cells that could be in the path from starting cell to the destination cell.

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Path Cost

Indices	0	1	2	3
0	1	3	8	15
1	2			
2	7			
3	9			

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

We initialized the first column and first row of MinCost.

We can write pseudocode to compute the cost (actually, min cost) in other cells that could be in the path from starting cell to the destination cell.

for row = 1 to destinationRow

for col = 1 to destinationCol

costWhenComingFromCellToLeft = MinCost [row, col – 1];

(next step on next slide)

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Min Cost

Indices	0	1	2	3
0	1	3	8	15
1	2	ff		
2	7			
3	9			

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

We initialized the first column and first row of MinCost.

We can write pseudocode to compute the cost (actually, min cost) in other cells that could be in the path from starting cell to the destination cell.

for row = 1 to destinationRow

for col = 1 to destinationCol

costWhenComingFromCellToLeft = MinCost [row, col - 1];

costWhenComingFromCellAbove = MinCost [row - 1, col];

Figuring out the smaller of these two : on next slide

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Min Cost

Indices	0	1	2	3
0	1	3	8	15
1	2			
2	7			
3	9			

Min cost in a grid

Starting point is the top left cell, i.e., row 0 and col 0

We initialized the first column and first row of MinCost.

We can write pseudocode to compute the cost (actually, min cost) in other cells that could be in the path from starting cell to the destination cell.

for row = 1 to destinationRow

for col = 1 to destinationCol

costWhenComingFromCellToLeft = MinCost [row, col - 1];

costWhenComingFromCellAbove = MinCost [row - 1, col];

MinCost [row, col] = **Min** (costWhenComingFromCellToLeft,
costWhenComingFromCellAbove)
+ Cost [row, col];

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Min Cost

Indices	0	1	2	3
0	1	3	8	15
1	2			
2	7			
3	9			

Min cost in a grid

What if we also allow **diagonal** traversal ?
How will the pseudocode below change?

```
for row = 1 to destinationRow
```

```
  for col = 1 to destinationCol
```

```
    costWhenComingFromCellToLeft = MinCost [ row, col - 1 ];
```

```
    costWhenComingFromCellAbove = MinCost [ row - 1, col ];
```

```
    MinCost [ row, col ] = Min (costWhenComingFromCellToLeft,  
                                costWhenComingFromCellAbove )  
    + Cost [ row, col ];
```

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Min Cost

Indices	0	1	2	3
0	1	3	8	15
1	2			
2	7			
3	9			

Min cost in a grid

Allowing **diagonal** traversal:

for row = 1 to destinationRow

for col = 1 to destinationCol

costWhenComingFromCellToLeft = MinCost [row, col – 1];

costWhenComingFromCellAbove = MinCost [row - 1, col];

costWhenComingFromCellDiagonallyAbove = MinCost[row - 1, col – 1];

MinCost [row, col] = **Min** (costWhenComingFromCellToLeft,
costWhenComingFromCellAbove,
costWhenComingFromCellDiagonallyAbove)
+ Cost [row, col];

Min cost to destinationRow, destinationCol is

MinCost [destinationRow, destinationCol]

Cost grid

Indices	0	1	2	3
0	1	2	5	7
1	1	2	6	3
2	5	3	7	2
3	2	3	2	1

Min Cost

Indices	0	1	2	3
0	1	3	8	15
1	2			
2	7			
3	9			

Min cost in a grid

Time complexity: numRows X numCols or $O(M \times N)$

Space complexity: $O(M \times N)$

```
for row = 1 to destinationRow
  for col = 1 to destinationCol
    costWhenComingFromCellToLeft = MinCost [ row, col - 1 ];
    costWhenComingFromCellAbove = MinCost [ row - 1, col ];
    costWhenComingFromCellDiagonallyAbove = MinCost[ row - 1, col - 1 ];
    MinCost [ row, col ] = Min (costWhenComingFromCellToLeft,
                                costWhenComingFromCellAbove,
                                costWhenComingFromCellDiagonallyAbove )
    + Cost [ row, col ];
```