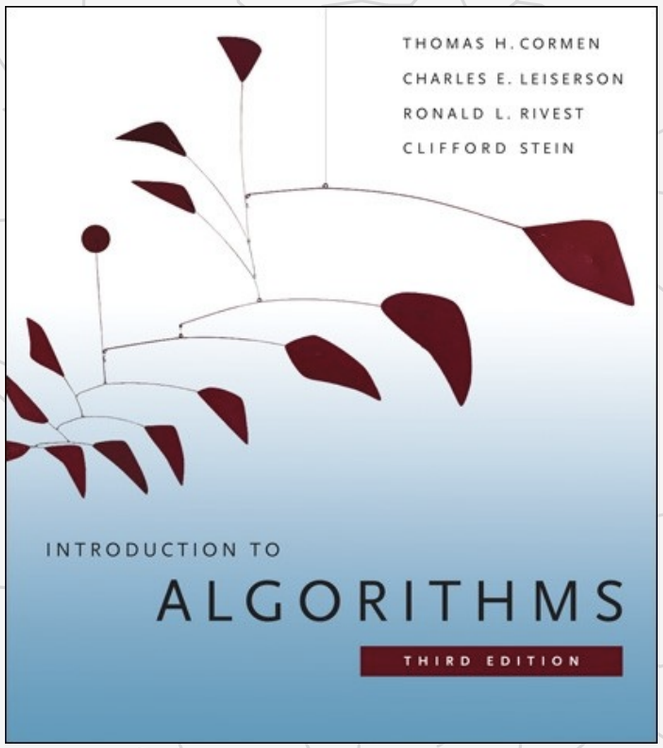
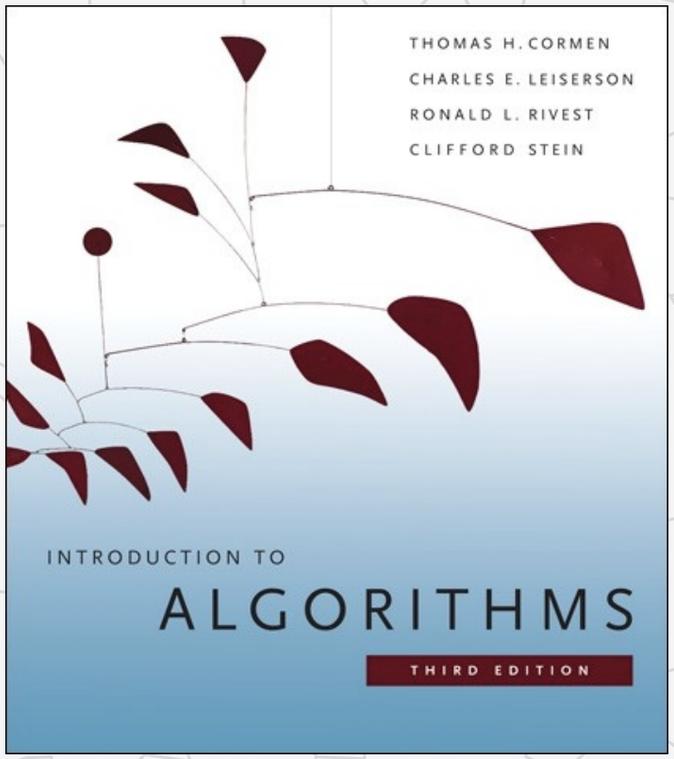


15 DYNAMIC PROGRAMMING

- Rod cutting
- Matrix multiplication
- Longest Common Subsequence
- Weighted Interval Scheduling
- Knapsack problem

HISTORY OF DYNAMIC PROGRAMMING

"We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities." — *Richard Ernest Bellman*



15 Dynamic Programming

- Rod cutting
- Matrix multiplication
- Longest Common Subsequence
- Weighted Interval Scheduling
- Knapsack problem
-

Rod cutting

The problem: Serling Enterprises buys long steel rods an cuts them into shorter rods, which it then sells at a profit

Each cut is free.

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	10	17	17	20	24	30

Given the price structure how could we make money?

Rod cutting visual

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

Given the price structure how could we make money?

1. Buy a rod of length 4 for 9

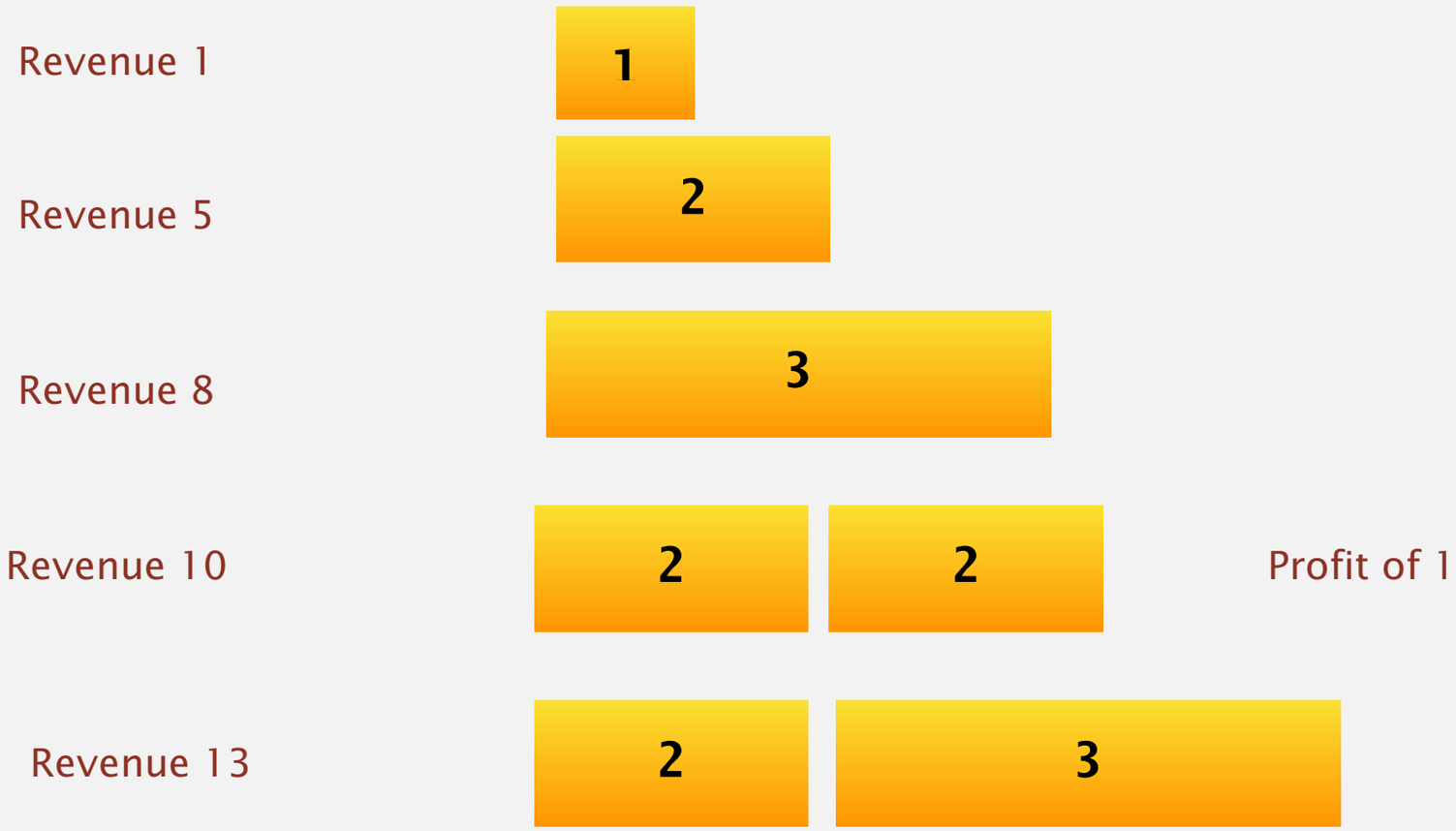
2. Cut into two piece of length 2 and sell for 5 each

3. Total for sale 5 + 5 = 10 : Profit 10 - 9 = 1

Revenue and Cuts

Revenue and Cuts

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



Rod cutting

Notation:

$i = 1, 2 \ldots\ldots n-1$ (length of the cut)

$n = i_1 + i_2 + i_3 \ldots. i_k$ (The length of the rod must equal the sum of its pieces)

$1 \leq k \leq n$ (the number of pieces between 1 and n)

$r_n = p_{i1} + p_{i2} + \ldots. p_{ik}$ (revenue is the sum of all the pieces)

$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2} \ldots\ldots, r_{n-1} + r_1)$

Price of rod Uncut

Every other combination.

Where represents the revenue for two pieces $r_1 + r_{n-1}$

Write a program to find the cut that maximizes revenue

Challenge: Since we don't know ahead of time which value of i optimizes revenue we have to consider all values of i including the case where we don't make a cut.

Insight: Once we make our first cut we consider can consider the two pieces as independent instances of the rod cutting problem.

What cut maximize revenue for this rod

$$r_n = \max(p_i + r_{n-1})$$

Example Run length of *4*

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

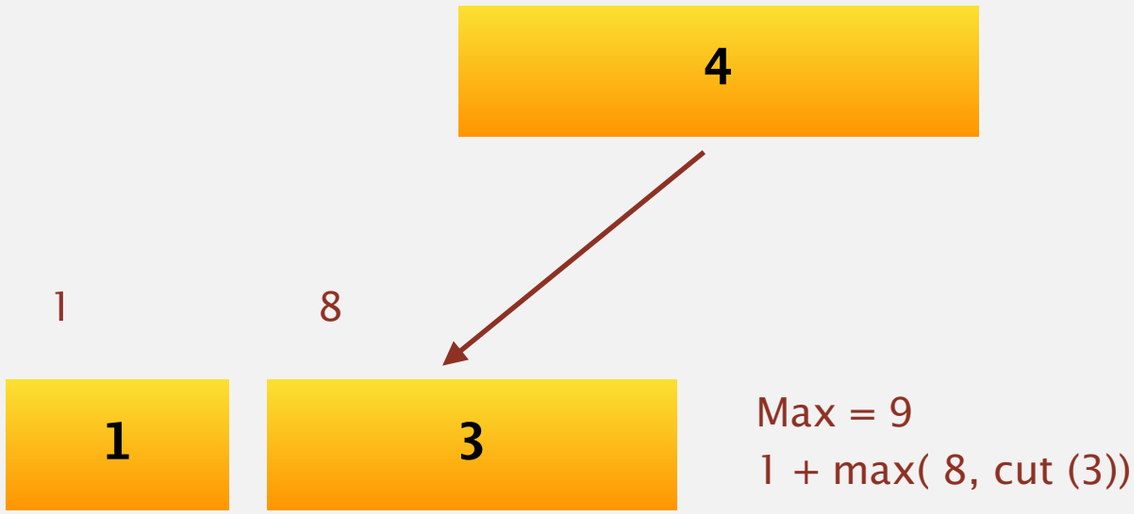
4

Max = -∞

Revenue 9

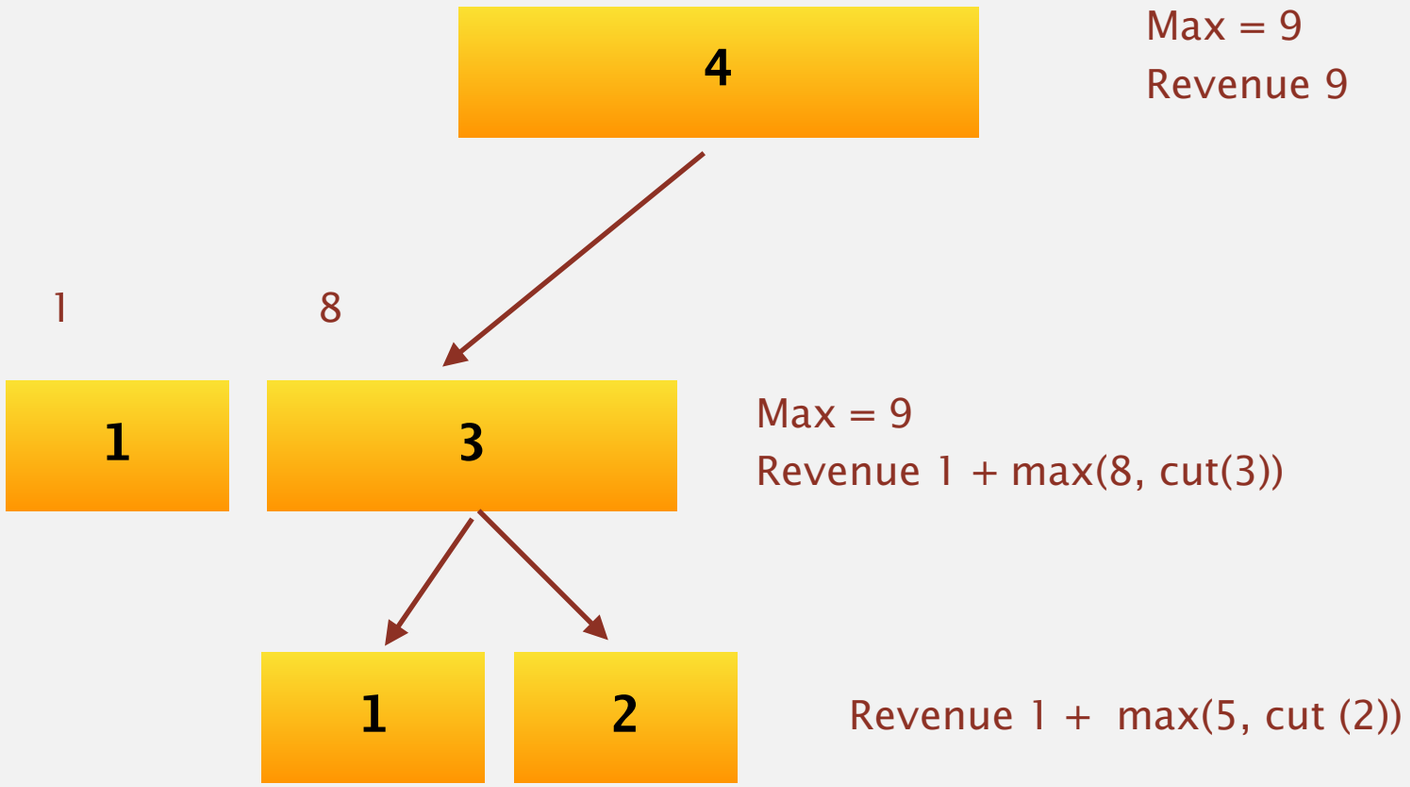
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



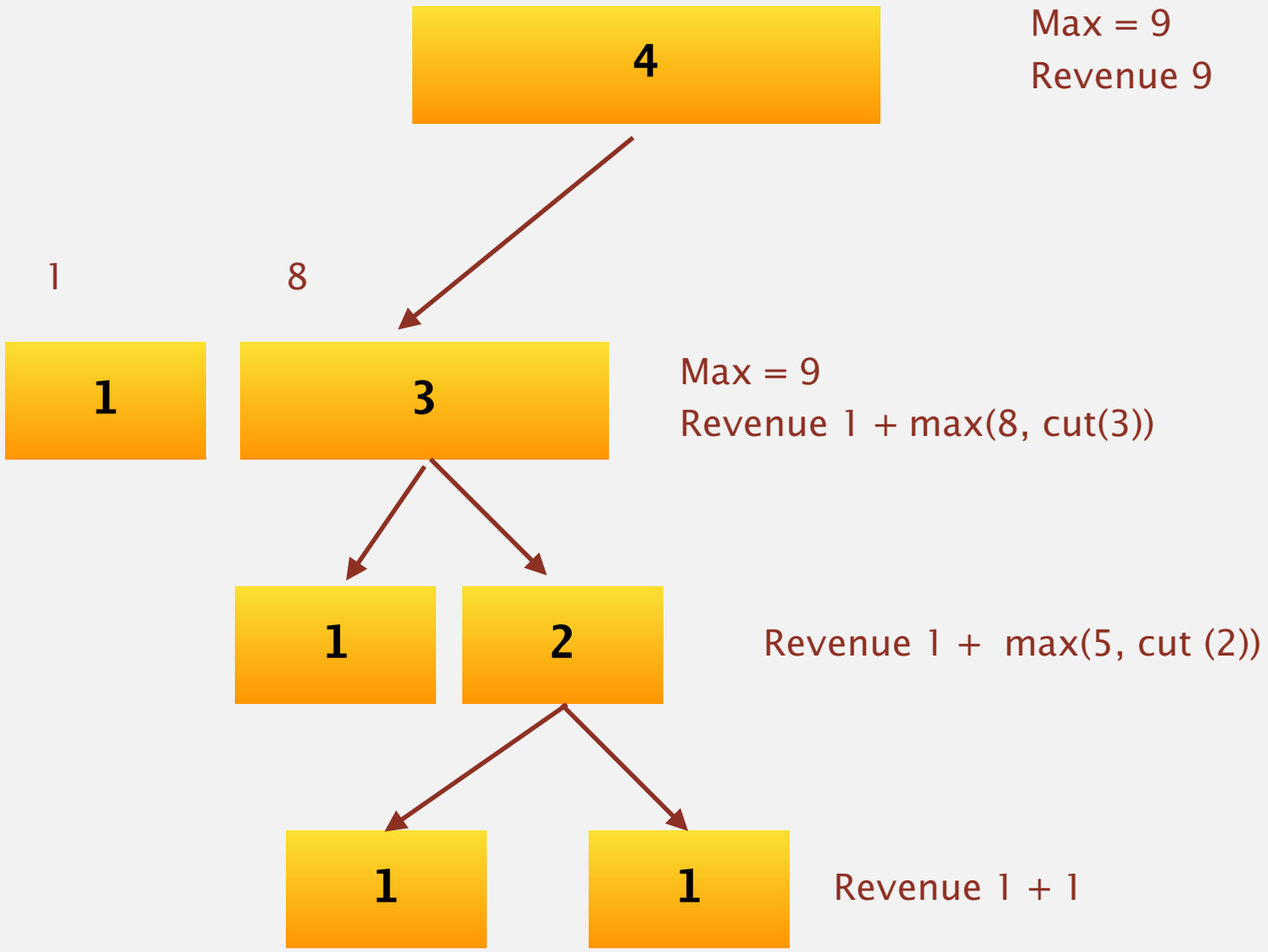
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



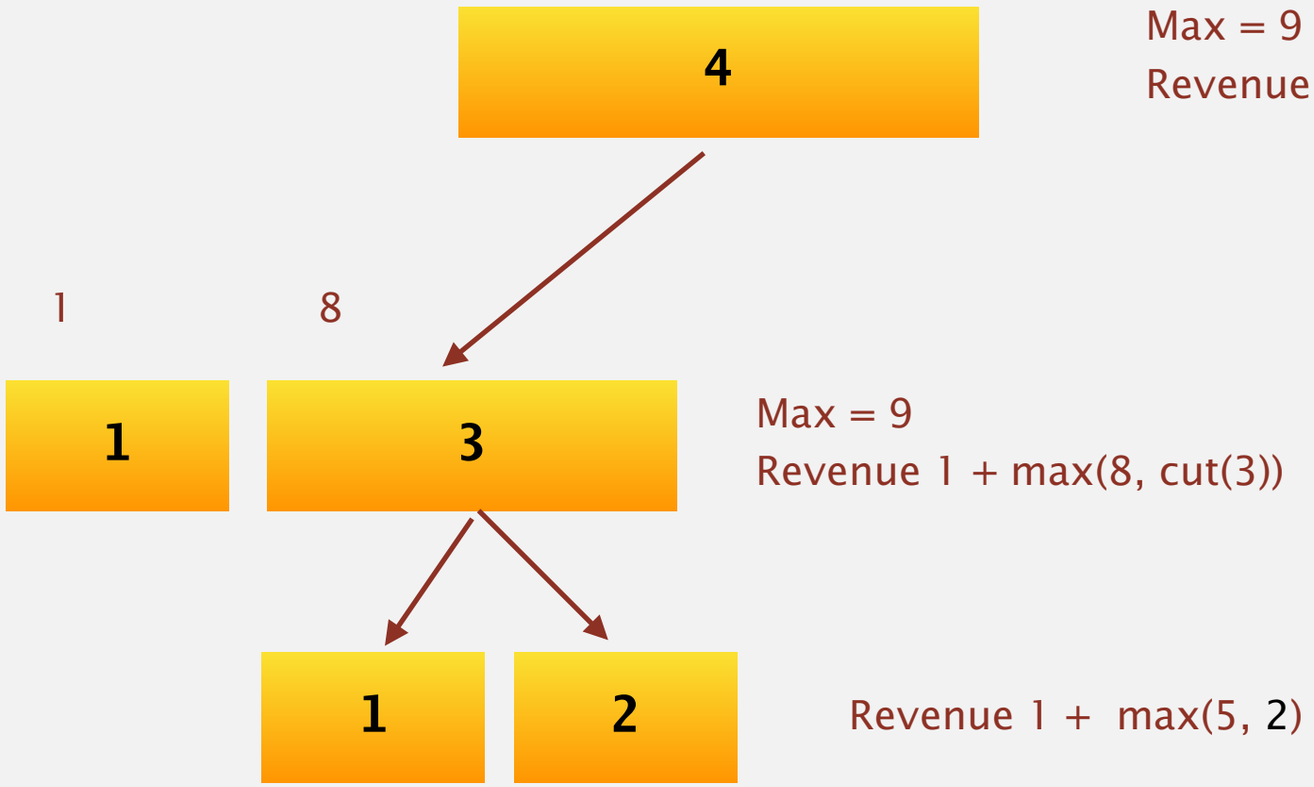
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



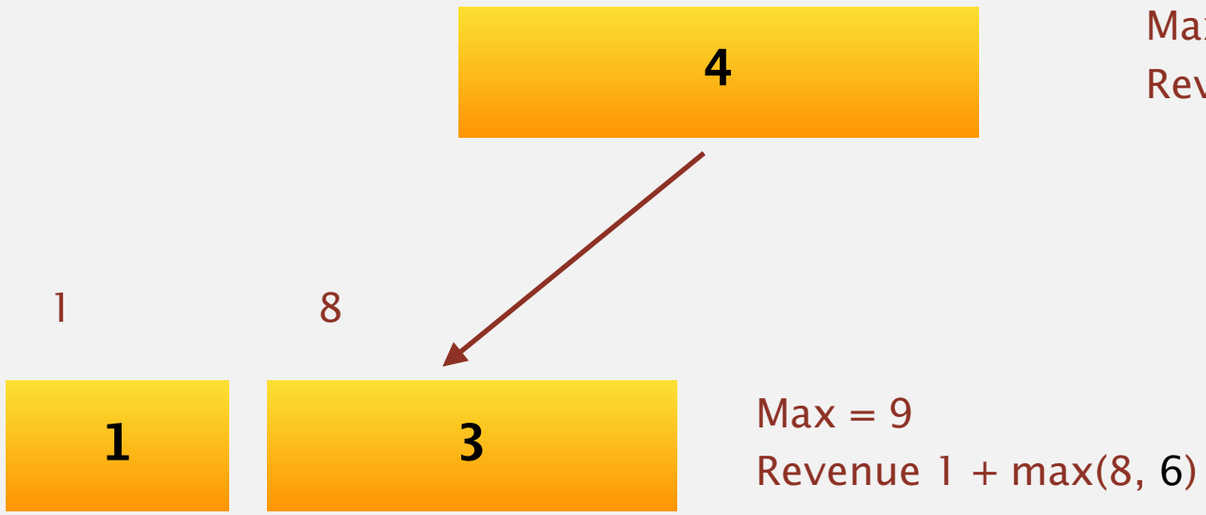
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



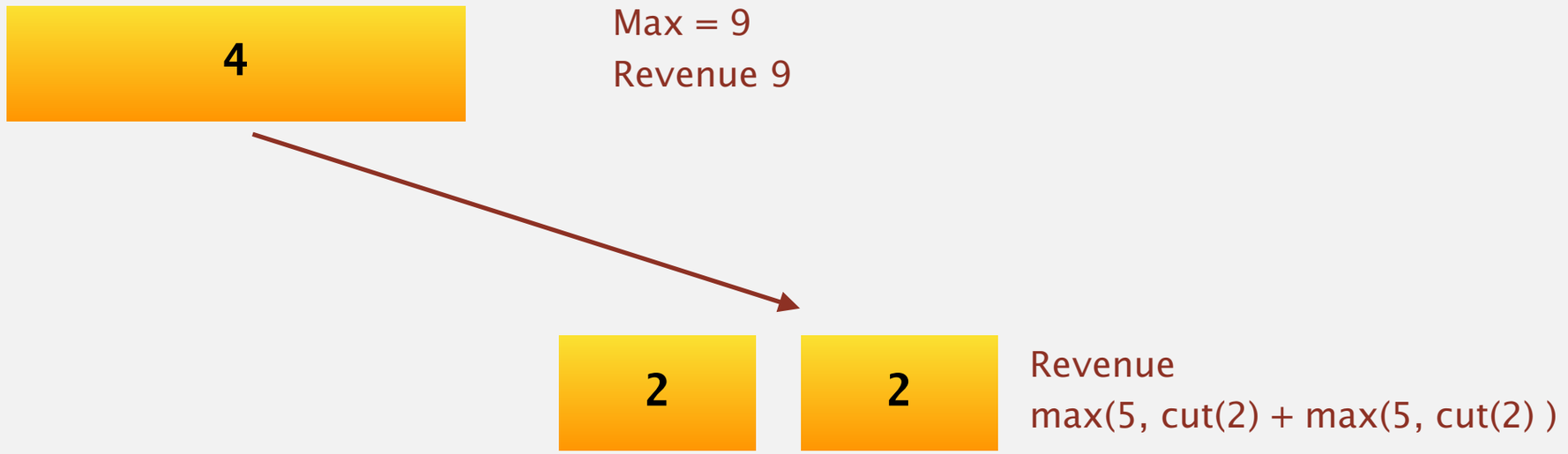
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



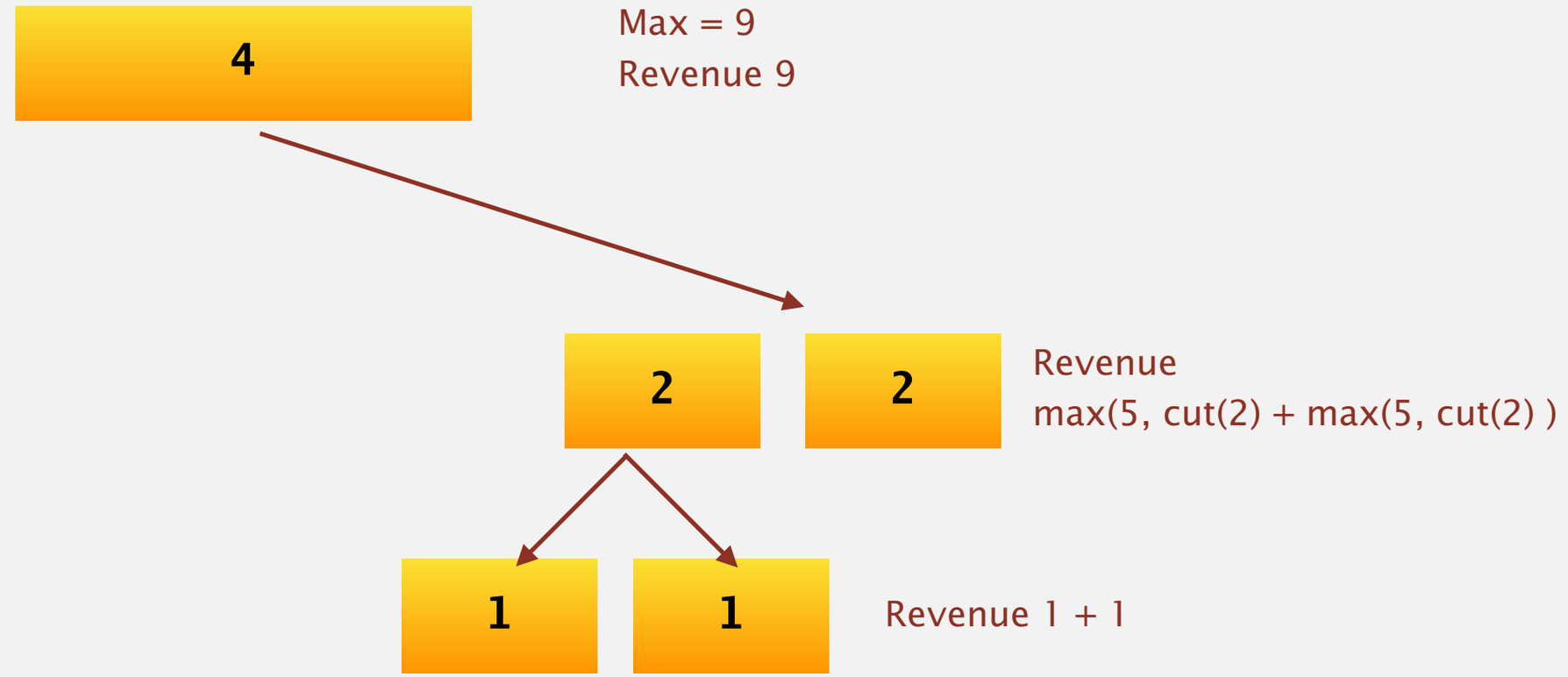
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



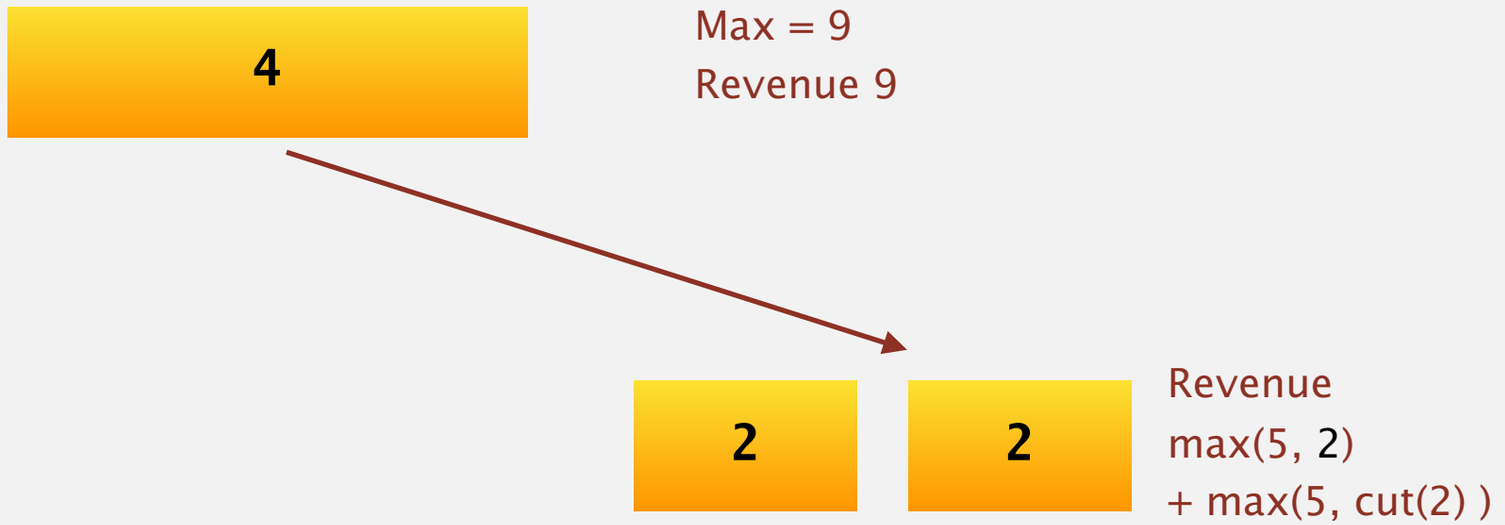
Example Run length of 4

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



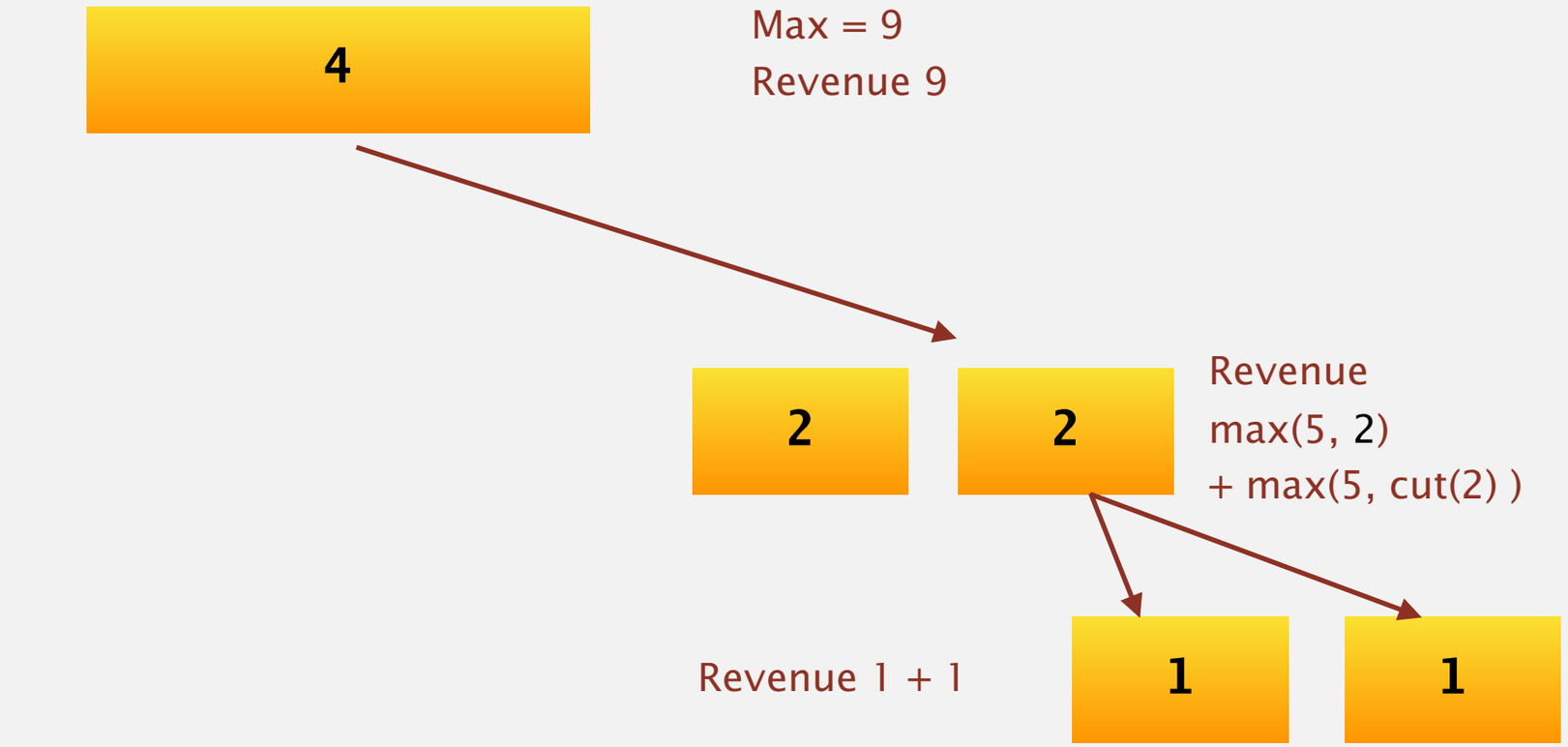
Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



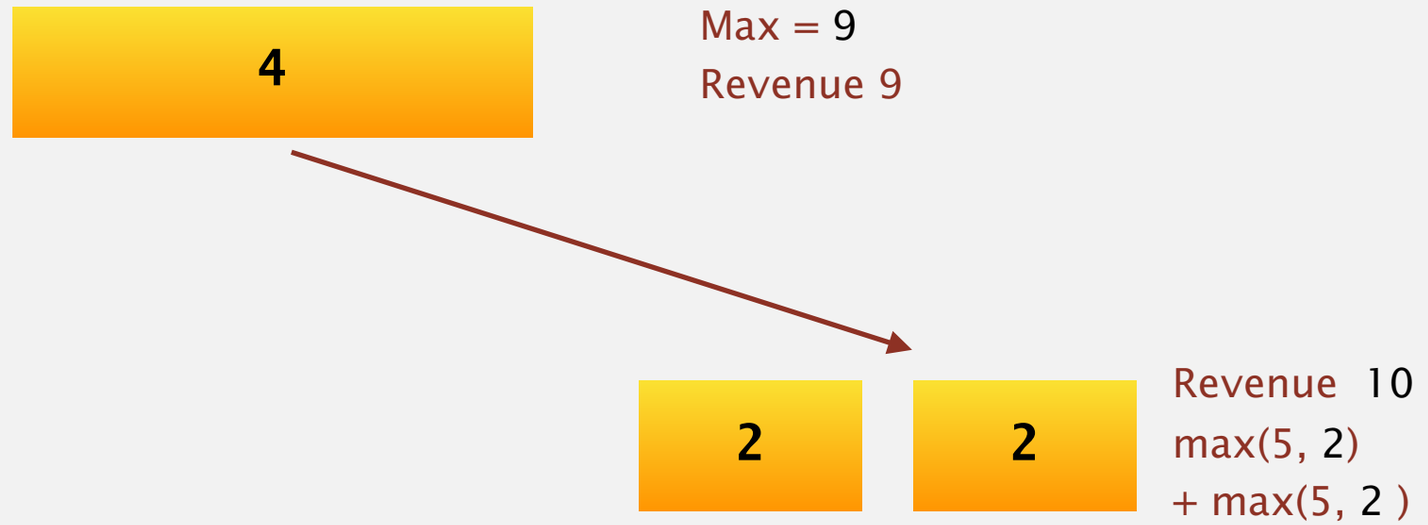
Example Run length of 4

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30



Example Run length of *4*

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

4

Max = 10
Revenue 9

Implementation

```
public class MaxProfit
{
   public static int cut(double[] prices, int length)
   {
      if( length == 0)
            return 0;
      double maxRev = Double.NegativeInfinity;
      for (int i = 1; i <= length;  i++)
      {
         maxRev = Math.max(maxRev, price[i]
                     + cut(price, length -i)
      }
   }
}
```

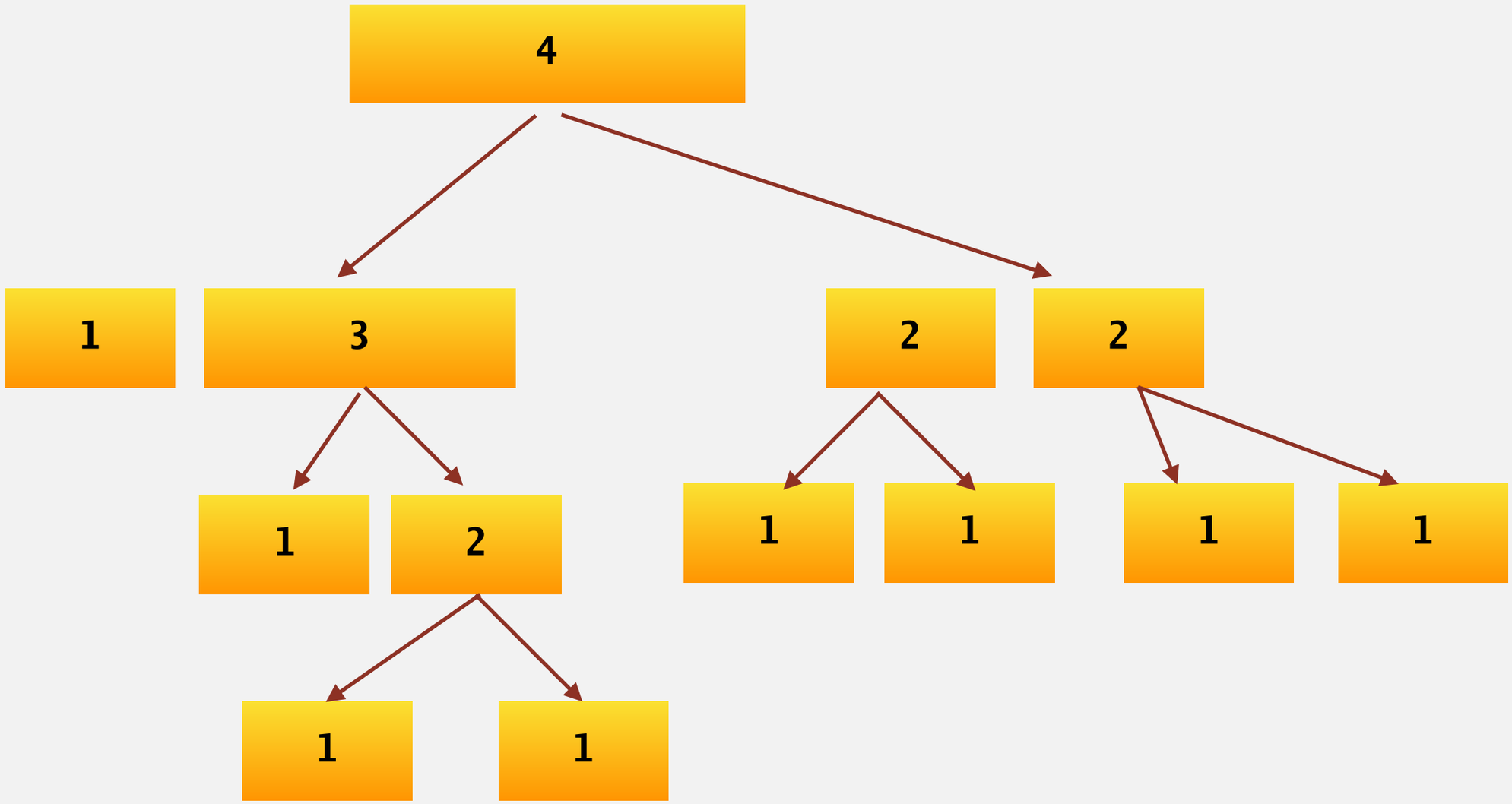
Base case rod of length 0

Recursive Call
Keep the max discovered

Run-time analysis

This Recursion Tree has 2^n node and 2^{n-1} leaves runtime ~ 2^{n-1}

Optimization : How we optimize this?



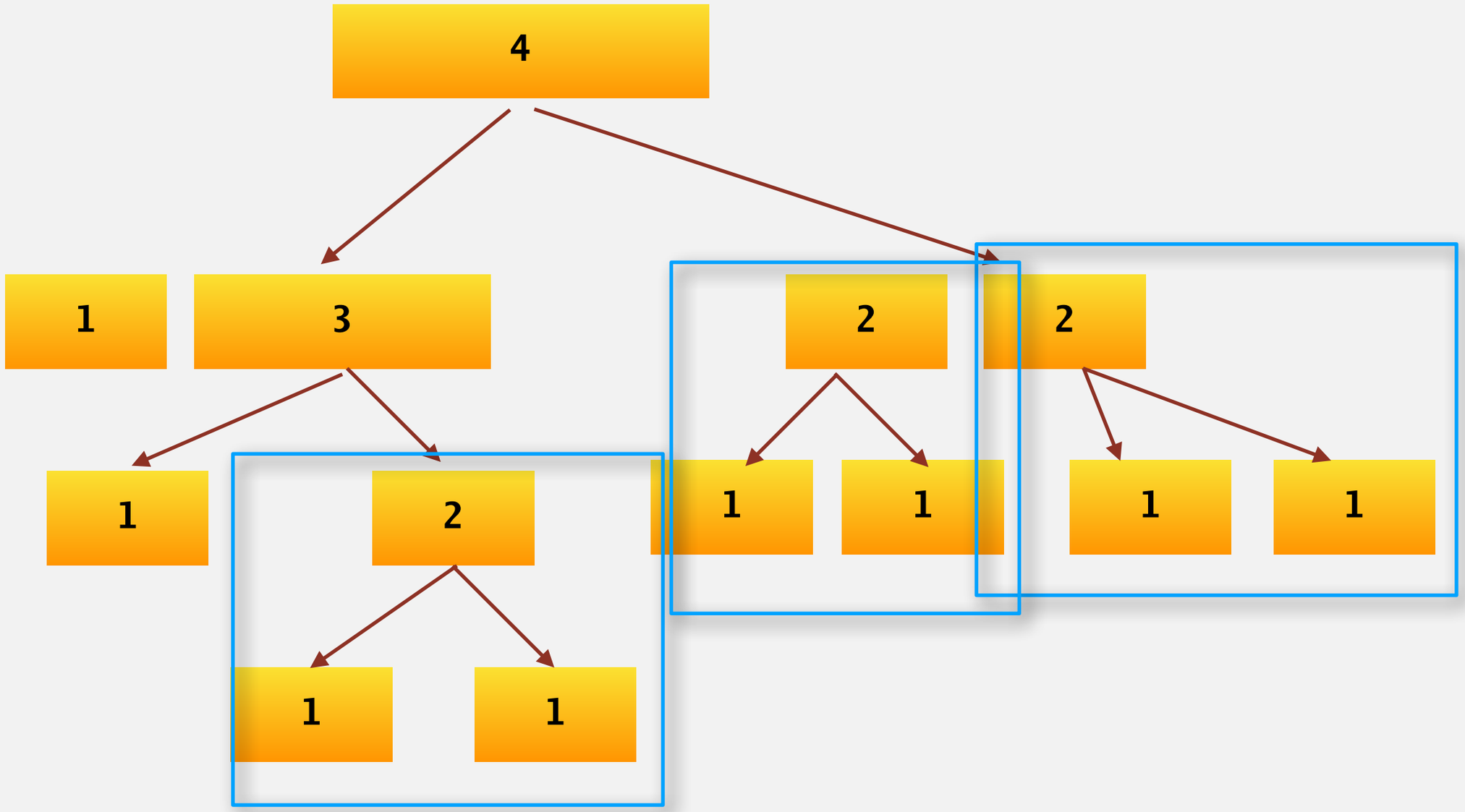
Run-time analysis

This Recursion Tree has 2^n node and 2^{n-1} leaves runtime ~ 2^{n-1}

Optimization : How we optimize this?

Simply store the result of recursive call reuse them:

Optimal substructure: optimal solutions to a problem incorporate optimal solutions to related problems



Dynamic programming & Optimal Substructure

Dynamic programming: arranges the subproblems so that they are only solved once and then looked up later. Trading memory for time.

```
public class MaxProfit
{
   public static int cut(double[] prices, int length)
   {
       double[] maxRev = new double[length + 1];
       for (int i = 1; i <= length;  i++)
       {
         maxRev[i] = Double.negativeInfinity;
       }
       return memCut(prices, length, maxRev);

   }
}
```

Array max revenue
for each length rod
Including zero length

Dynamic programming & Optimal Substructure

Dynamic programming: arranges the subproblems so that they are only solved once and then looked up later. Trading memory for time.

```
public static int memCut(double[] prices, int length, double[] maxRev)
{
     if maxRev[length] >= 0
          return max[length];
    if(length == 0)
          return 0;
    double pieceMaxRev = Double.negativeInfinity;
    for (int i = 1; i <= length;  i++)
    {
          pieceMaxRev = Math.max(pieceMaxRev, prices[i]
                    + menCut(prices, length - 1, maxRev));
     }
      maxRev[length] = pieceMaxRev;
     Return pieceMaxRev;
}
```

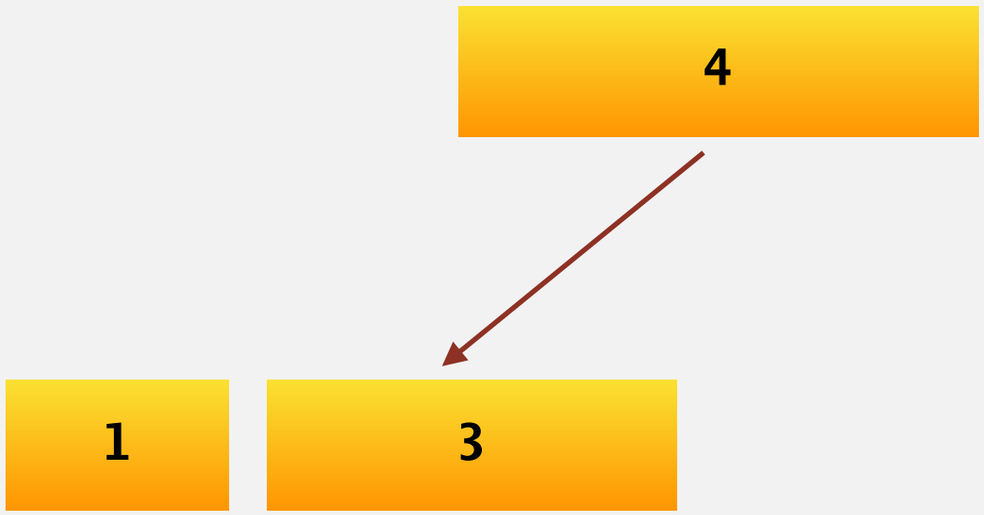
Example Memorization

4

Length	Revenue
4	∞
3	∞
2	∞
1	∞
0	∞

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

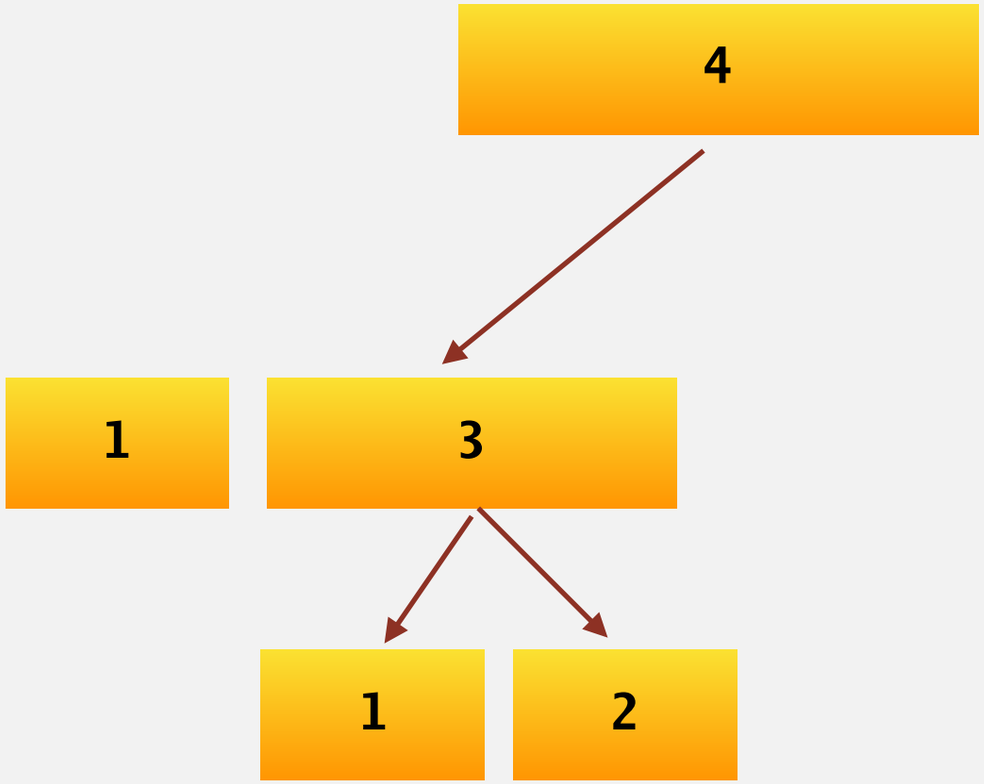
Example Memorization



Length	Revenue
4	∞
3	∞
2	∞
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

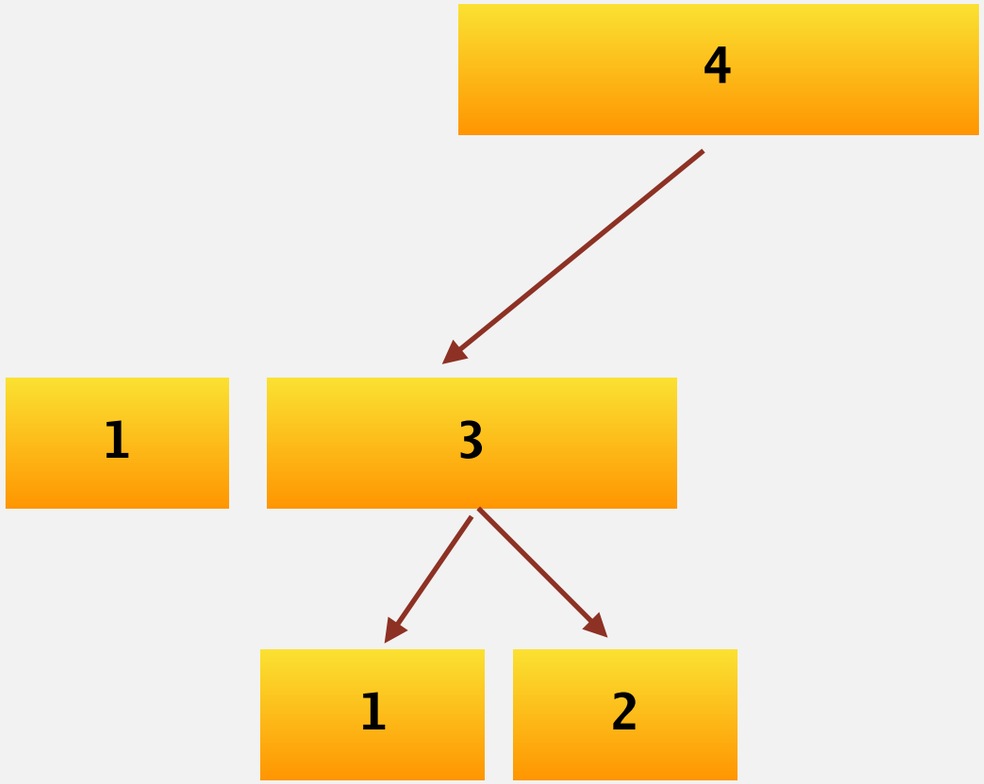
Example Memorization



Length	Revenue
4	∞
3	∞
2	∞
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

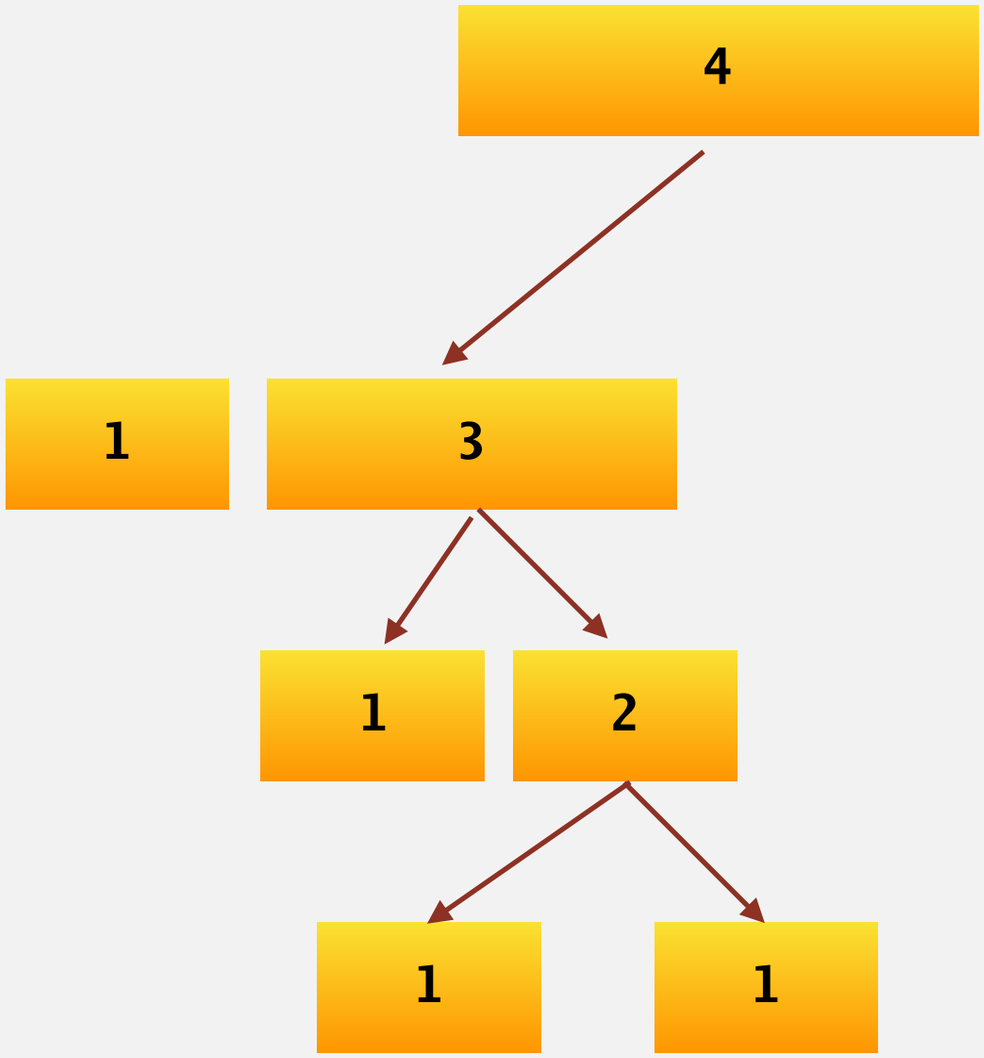
Example Memorization



Length	Revenue
4	∞
3	∞
2	∞
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

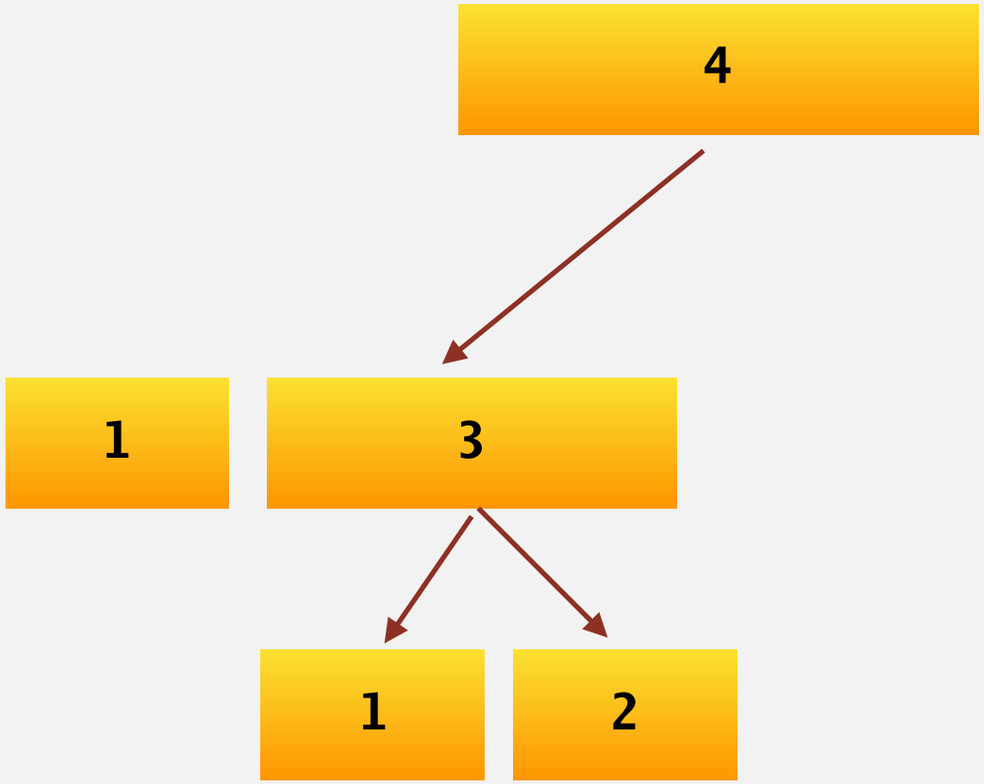
Example Memorization



Length	Revenue
4	∞
3	∞
2	5
1	1
0	0

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

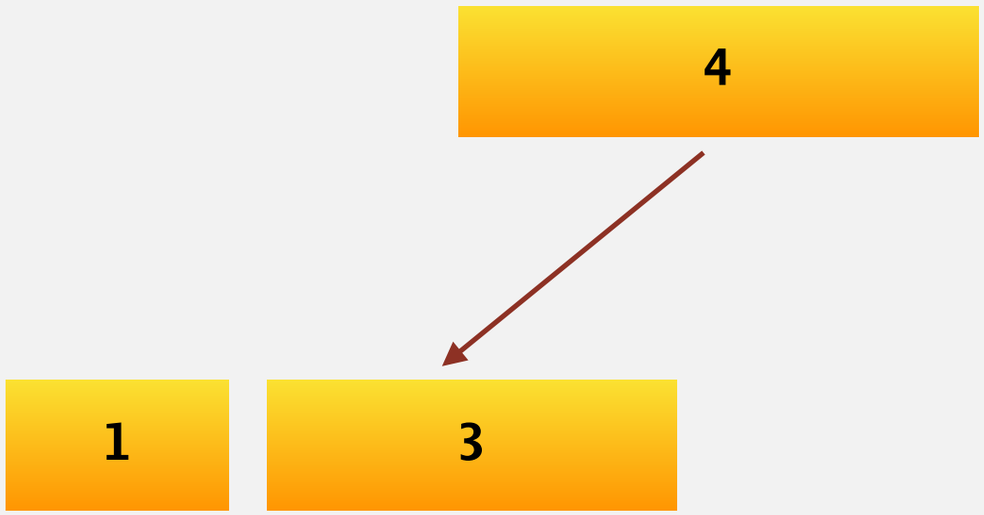
Example Memorization



Length	Revenue
4	∞
3	∞
2	5
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

Example Memorization



Length	Revenue
4	∞
3	8
2	5
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

Example Memorization

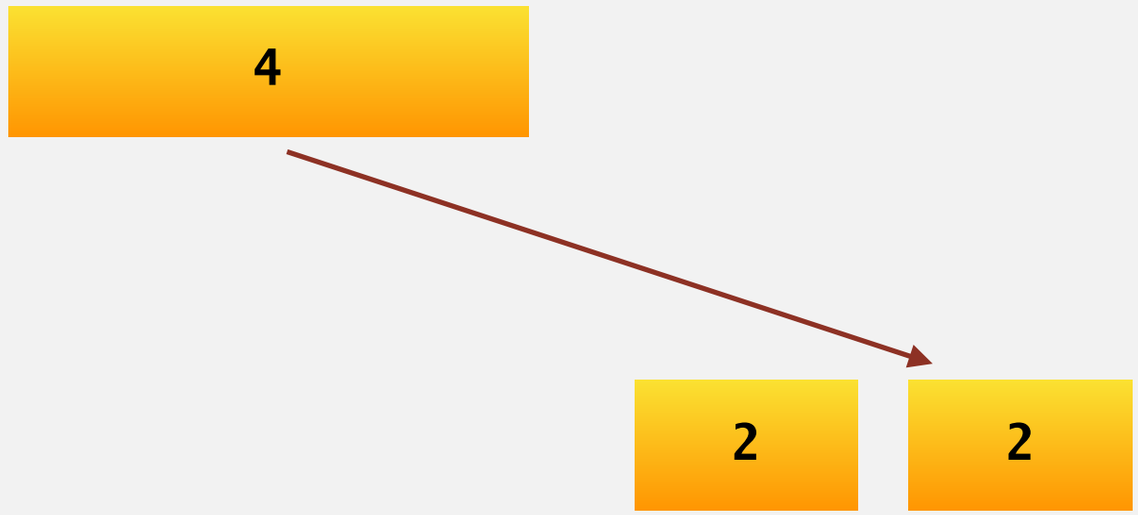
4

Note 4 is not set because the loop is still going →

Length	Revenue
4	∞
3	8
2	5
1	1
0	0

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

Example Memorization



Length	Revenue
4	∞
3	8
2	5
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

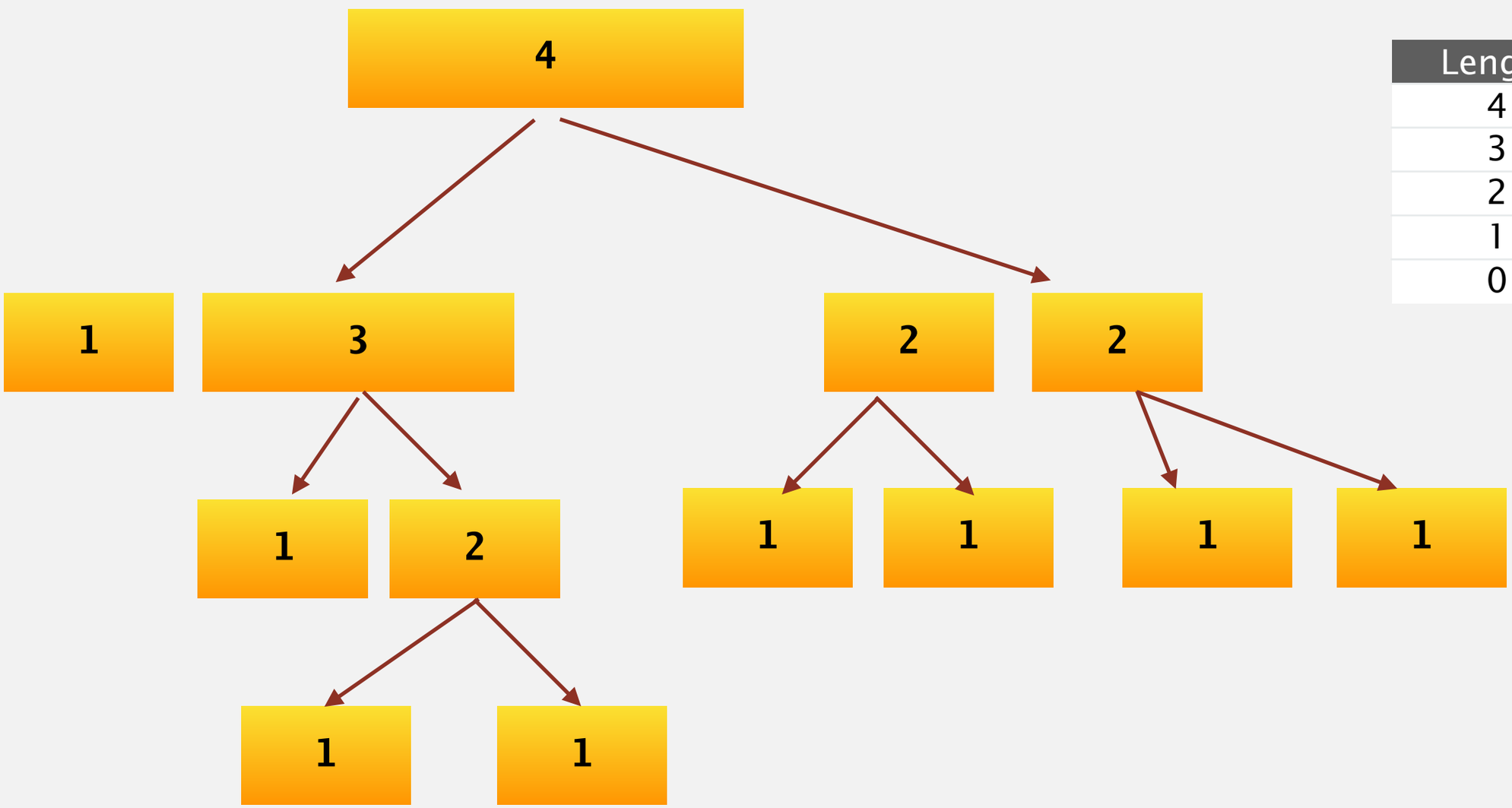
Example Memorization

Length	Revenue
4	10
3	8
2	5
1	1
0	0

Length I	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

Top down approach

Top Down: This recursive approach is often referred to as a top down approach



Length	Revenue
4	10
3	8
2	5
1	1
0	0

Bottom up approach

0

Revenue = 0

Already computed

1 | 1

maxRev[1]= max(-∞, price[1] + maxRev[0])

2 | 2

maxRev[2]= max(-∞, price[1] + maxRev[1],
price[2] + maxRev[0])

3 | 3

maxRev[2]= max(-∞, price[1] + maxRev[2],
price[2] + maxRev[1], price[3] + maxRev[0])

4 | 4

maxRev[4]= max(-∞, price[1] + maxRev[3],
price[2] + maxRev[2], price[3] + maxRev[1] ,
price[4] + maxRev[0])

Why not
maxRev[2] + maxRev[2]

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30

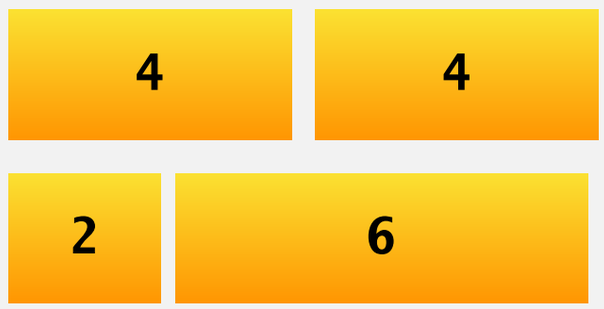
Question

Question: Why price[2] + maxRev[2], and not maxRev[2] + maxRev[2]

For each cut you must sell at least one of the rods at market price. It could be the cut of size one or a larger cut. For compound cuts they broken down into some final base piece that is sold at market price.

So this bottom up approach is finding which rod we sell at market price and the max revenue that we can get for remaining.

Key for bottom up: the remaining was computed in pervious step correctly.

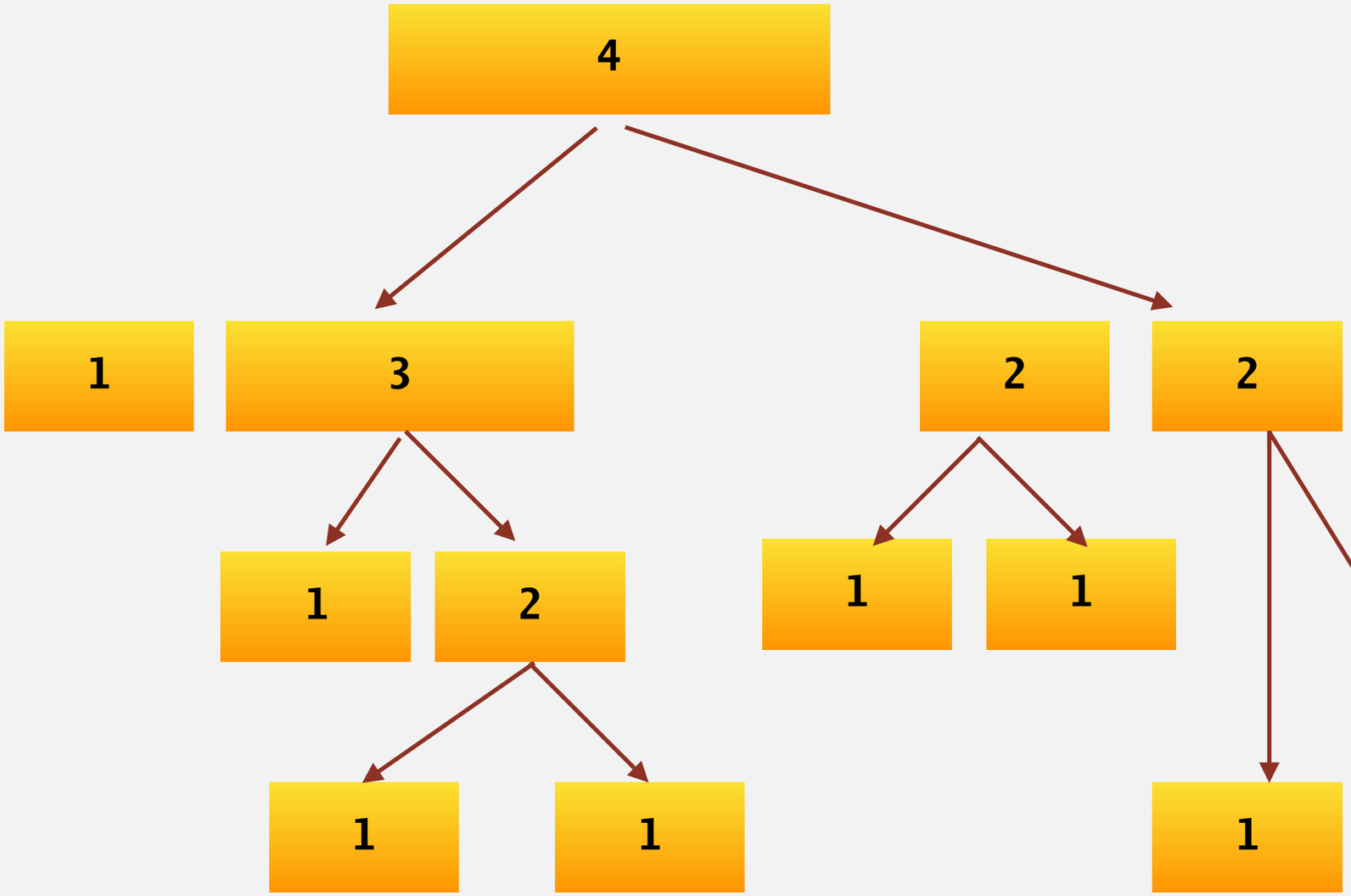


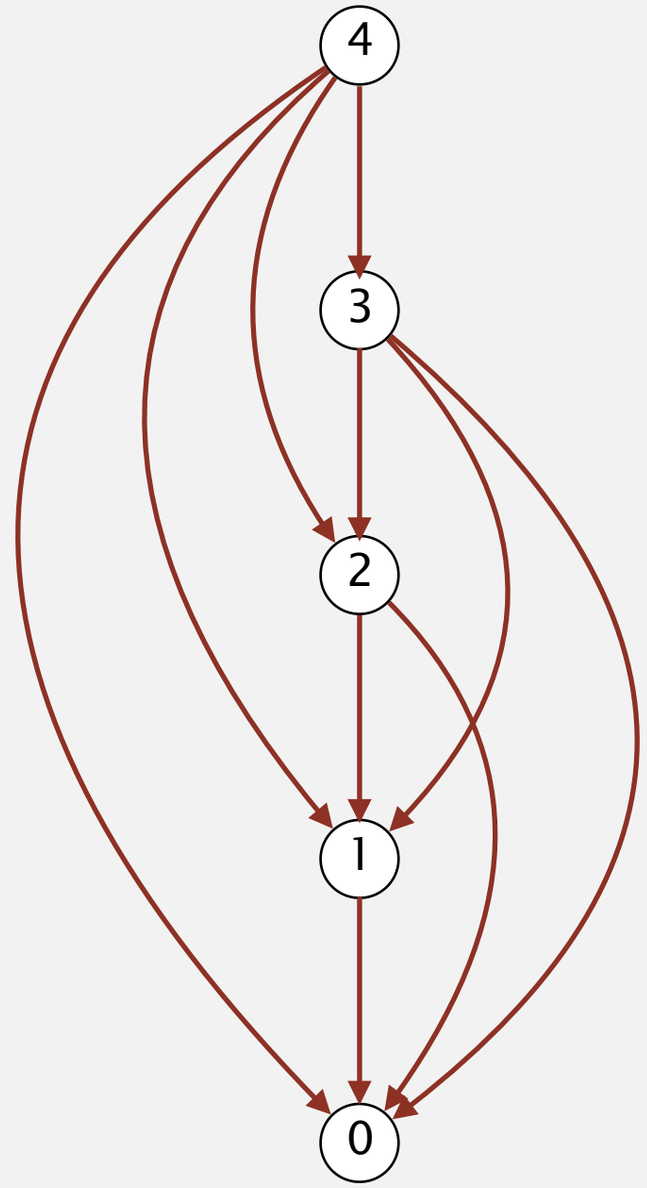
Bottom up approach

```
public static int cutBottomUp(double[] prices, int length)
{
    double[] maxRev = new double[length +1];
    maxRev = 0;
    for (int i = 1; i <= length;  i++)
    {
          double pieceMaxRev = Double.negativeInfinity;
          for(int j = 1; j <= i; j++)
          {
                pieceMaxRev = Math.max(pieceMaxRev, prices[i]
                      + maxRev[i-j]);
           }
         maxRev[i] = pieceMaxRev;
     }
    return maxRev[length];
}
```

Subproblem graph

Subproblem graph is a great way to understand how problems depend on each other





Reconstructing a solution

Challenge: Print out the sequence of cuts required to get the maximum revenue

```
public static int cutBottomUp(double[] prices, int length)
{
   double[] maxRev = new double[length +1];
   int[] cuts = new int[length + 1];
   maxRev = 0;
   for (int i = 1; i <= length;  i++)
    {
          double pieceMaxRev = Double.negativeInfinity;
          for(int j = 1; j <= i; j++)
          {
              if ( pieceMaxRev < price[i] + maxRev[i -j])
              {
                 pieceMaxRev = prices[i] + maxRev[j-1];
                 cut[i] = j;
              }
           }
        maxRev[i] = pieceMaxRev;
    }
   return new solution(maxRev,cuts) ;
}
```

Reconstructing a solution

Length l	1	2	3	4	5	6	7	8	9
Price p_i	1	5	8	9	10	17	17	20	30
Revenue	1	5	8	10	13	17	18	22	25
cut	1	2	3	2	2	6	1	2	3

```
public static void printCuts(double[] maxRev, int[] cuts, int length)
{
   while (length > 0)
   {
     System.out.println(cuts[length]);
     length = length - cuts[length];
   }
}
```

Quiz

Question: Calculate the n Fibonacci numbers using dynamic programming.

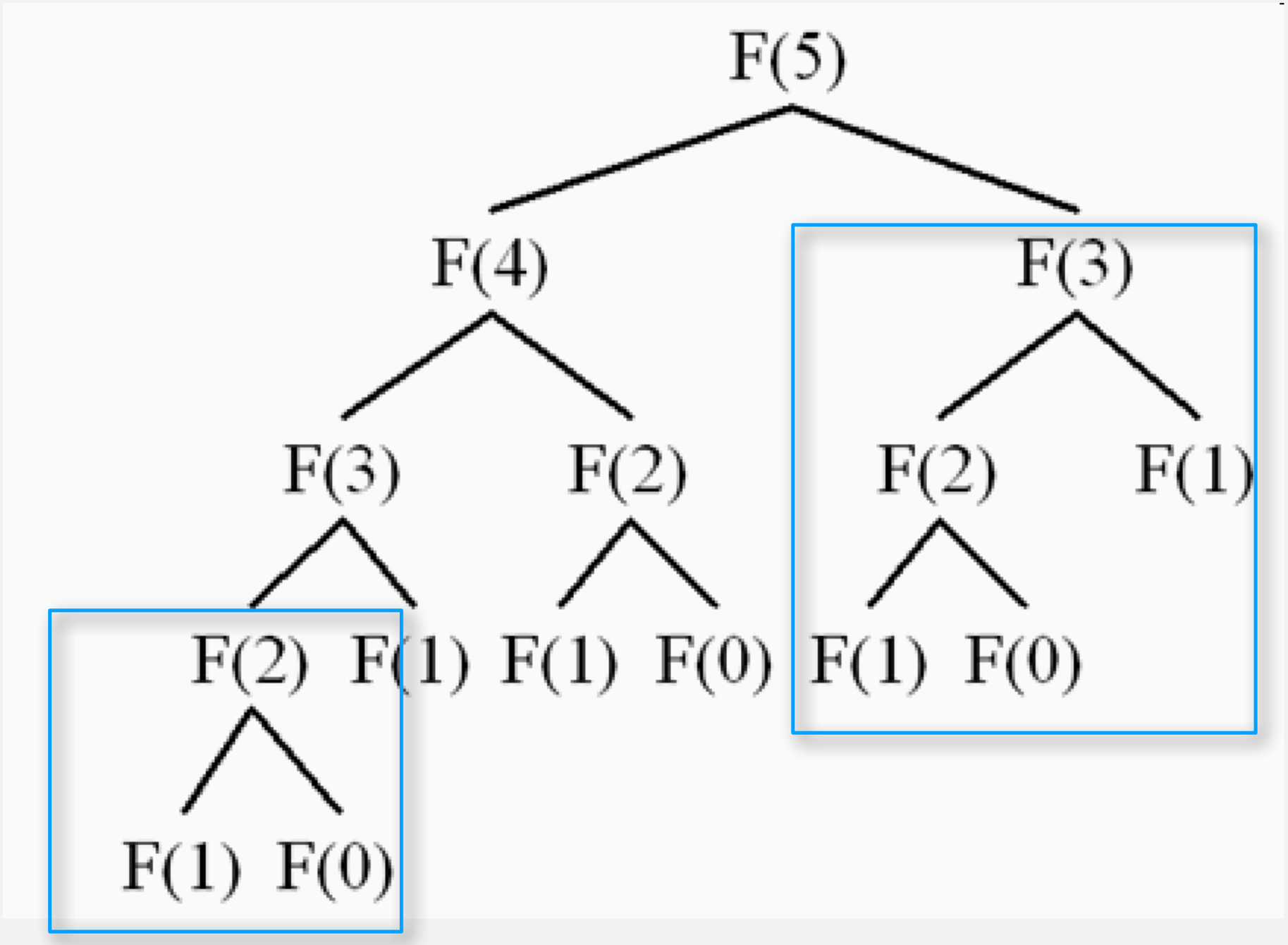
$$F_0 = 0, \quad F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2},$$

```
public static long fib(int n)
{
   if (n == 0) return 0;
   if( n == 1) return 1;

   return(fib(n-1) + fib(n-2));
}
```

Quiz

Question: Calculate the n Fibonacci numbers using dynamic programming.



Quiz

Question: Calculate the n Fibonacci numbers using dynamic programming.

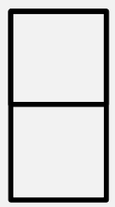
```
public static long fib_dp(int n)
{
  int i;
  long[]f = new long[n + 1];
  f[0] = 0;
  f[1] = 1;
   for( i = 2; i <= n; i++)
       f[i]= f[i-1] + f[1-2];

   return(f[n]);
}
```

Quiz two

Question: How many was can you tile 2xn board with dominions

How many ways to tile this:

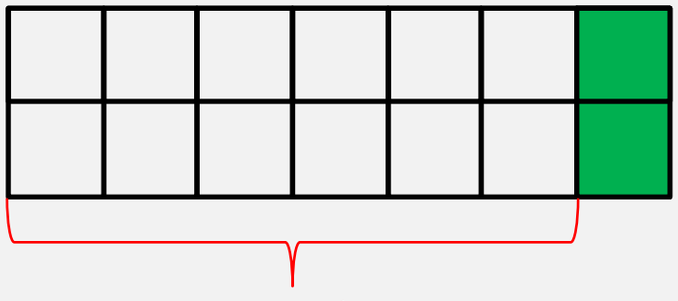
With these?



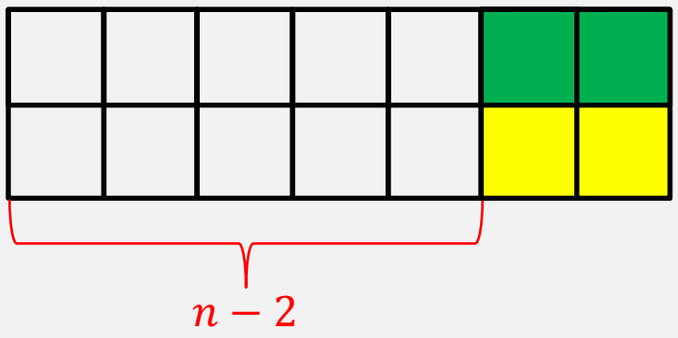
Quiz two

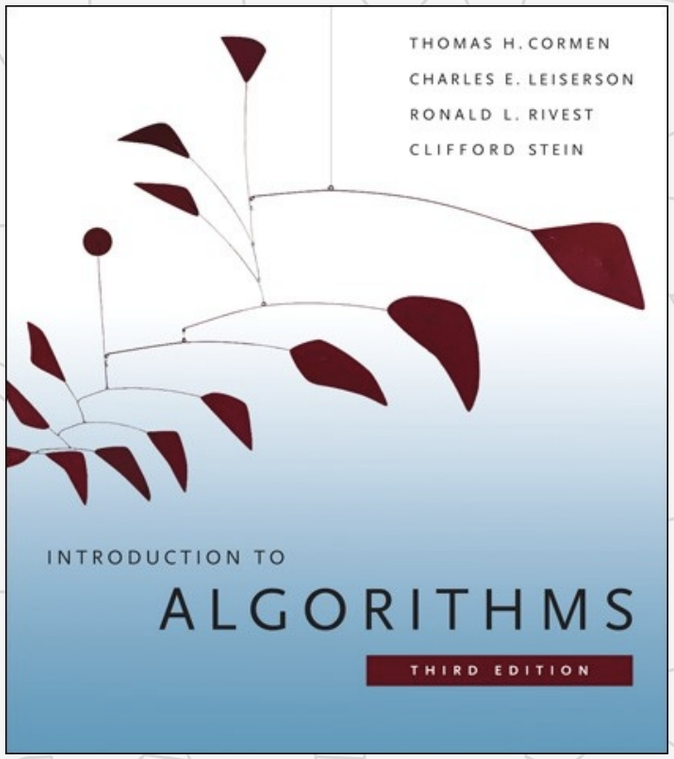
Question: How many was can you tile 2xn board with dominions

Two ways to fill the final column:



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





15 Dynamic Programming

- Rod cutting
- Matrix multiplication
- Longest Common Subsequence
- Weighted Interval Scheduling
- Knapsack problem
-

Longest Common subsequence

Problem: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

$X = ATCTGAT$

$Y = TGCATA$

$LCS = TCTA$

Brute force solution

- Calculate all subsets of Y (2^n)
- Check each subset against X to see number of matches

X = T A A T T A
Y = A T A
LCS = A T A

Analysis: Run time of brute force solution is M*(2^n-1) (Why?)

Don't check the empty set

A

T

A

A T

T A

A A

A T A (longest match)

Calculating the subsets

Problem: Calculate the number of subsets

{d, c, b, a}

Interesting abstraction

0 = [0000]	{}
1 = [0001]	{a}
2 = [0010]	{b}
3 = [0011]	{a, b}
4 = [0100]	{c}
5 = [0101]	{a, c}
6 = [0110]	{b, c}
7 = [0111]	{a, b, c}
…….	……..
2^n=[1111]	{a, b, c, d}

Brute force code Calculating the number of subsets

```
public static ArrayList<Sting> calSubsets(String X)
{









}
```

Brute force code Calculating the number of subsets

```
public static ArrayList<Sting> calSubsets(String X)
{
    int length = X.length();
    ArrayList<String> subsets = new ArrayList<>();
    for(int i = 0; i < (1 << length); i++)
    {
        String subset = “”;
        for(int index = 0; index < length; index)
        {
          if( 1<<index & i != 0)
             subset += X.charAt(index);
        }
       subsets.add(subset);
    }

   return subsets;
}
```

Thinking about the problem recursively

What if we try moving backwards

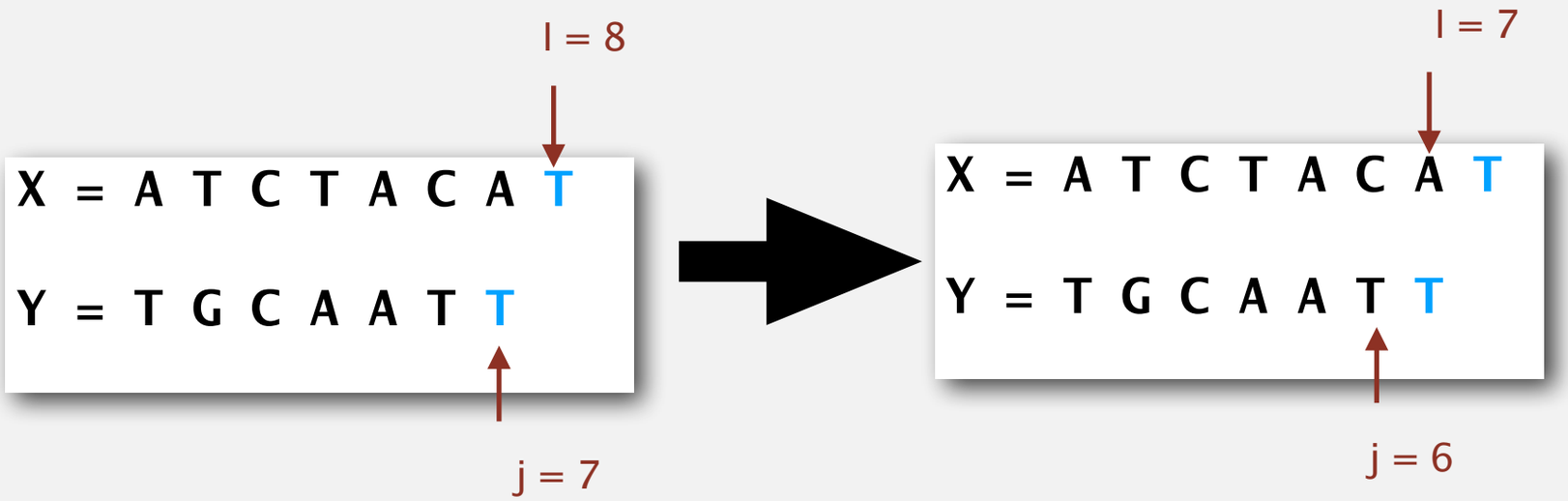
Let $LCS(i,j)$= length of the LCS for the first i characters of X, first j characters of Y

Find $LCS(i,j)$

Case 1

X[i] = Y[j]

$LCS(i,j) = LCS(i - 1, j - 1) + 1$



Thinking about the problem recursively

What if we try moving backwards

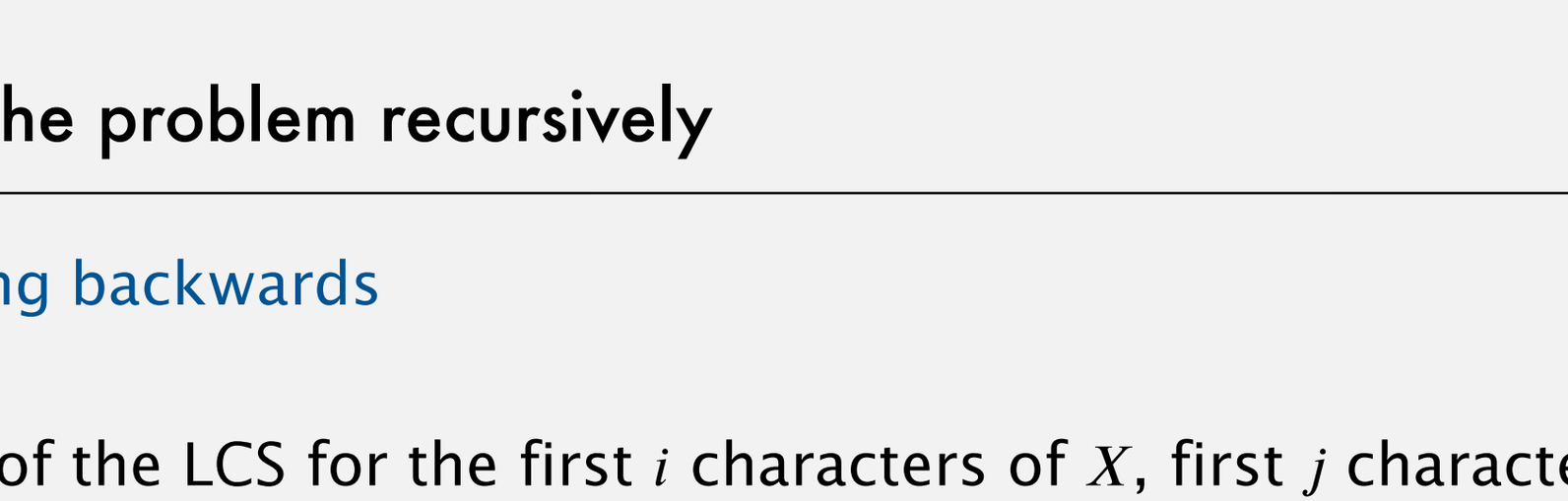
Let $LCS(i,j)$= length of the LCS for the first i characters of X, first j characters of Y

Find $LCS(i,j)$

Case 2:

X[i] ≠ Y[j]

$LCS(i,j)=LCS(i,j-1)$



Case 1

X[i] = Y[j]

$LCS(i,j)= LCS(i\text{ -1 },j\text{ - 1}) + 1$

Thinking about the problem recursively

What if we try moving backwards

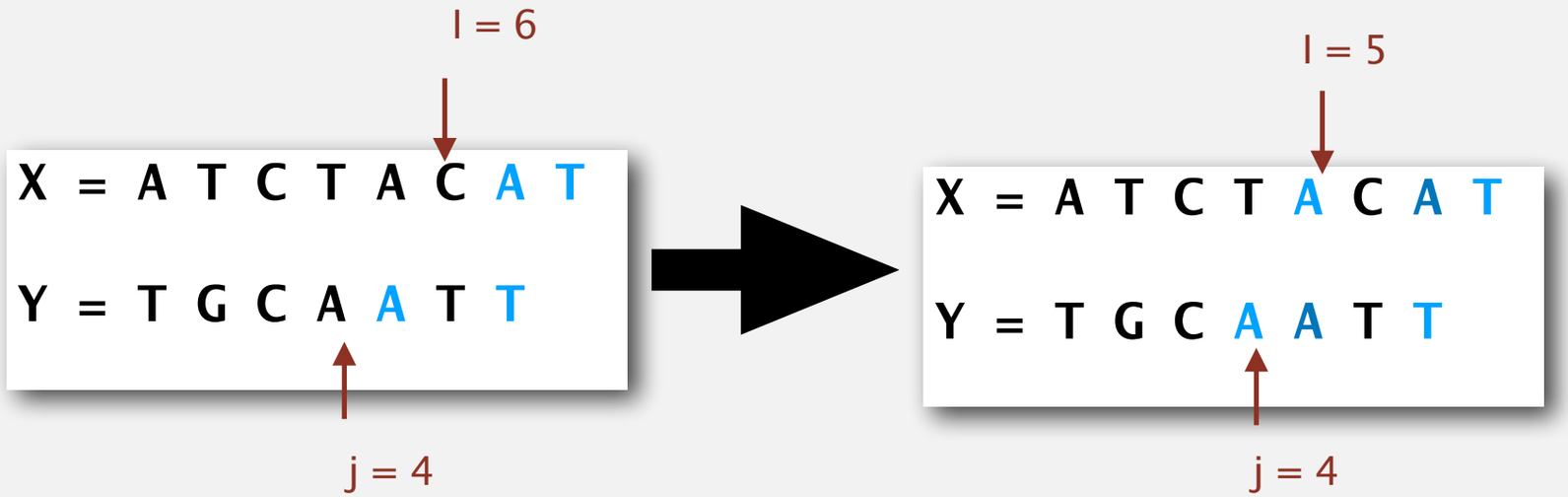
Let $LCS(i,j)$= length of the LCS for the first i characters of X, first j characters of Y

Find $LCS(i,j)$

Case 3:

X[i] ≠ Y[j]

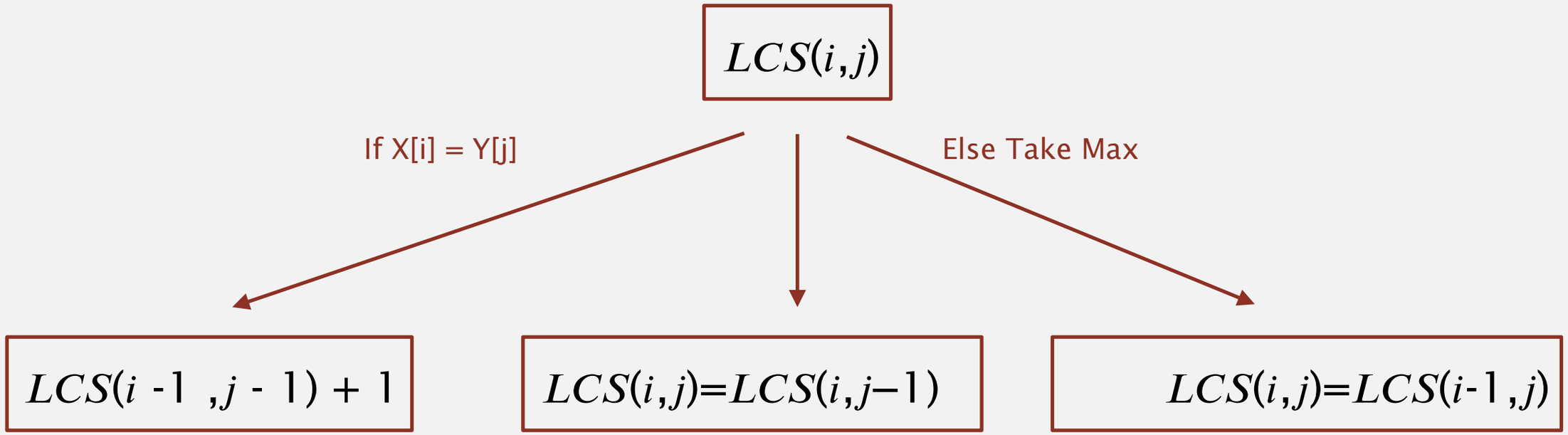
$LCS(i,j)=LCS(i\text{-}1,j)$

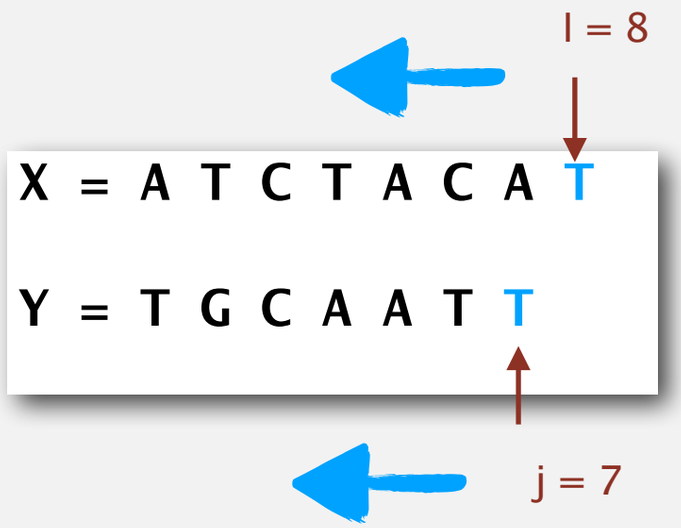


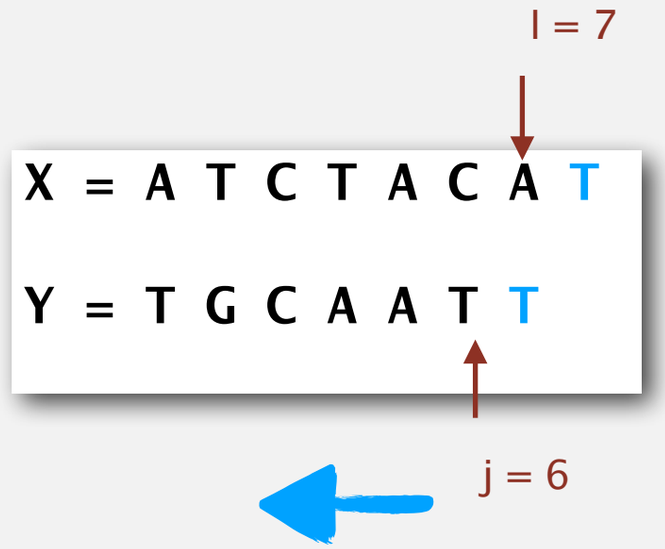
Recursive Call

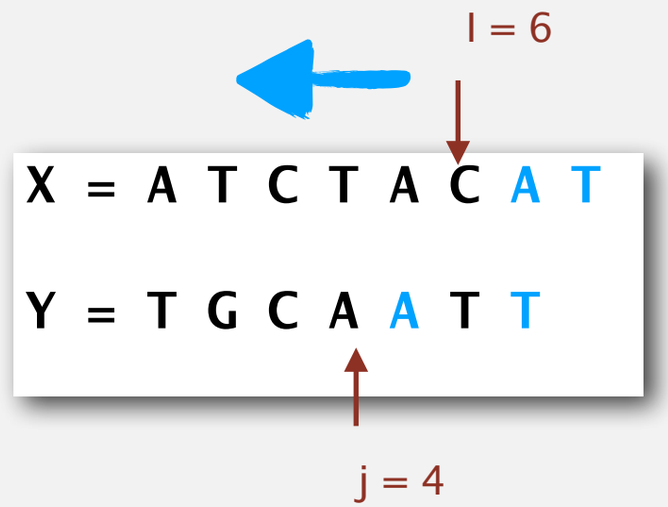
$$LCS(i,j) = \begin{cases} \text{Case 0} \\ \quad i = j = 0 \\ \quad LCS(i,j) = [] \\ \\ \text{Case 1} \\ \quad X[i] = Y[j] \\ \quad LCS(i,j) = LCS(i-1, j-1) + 1 \\ \\ \max \begin{cases} \text{Case 2:} \\ \quad X[i] \neq Y[j] \\ \quad LCS(i,j) = LCS(i, j-1) \\ \\ \text{Case 3:} \\ \quad X[i] \neq Y[j] \\ \quad LCS(i,j) = LCS(i-1, j) \end{cases} \end{cases}$$

Recursive Tree Structure









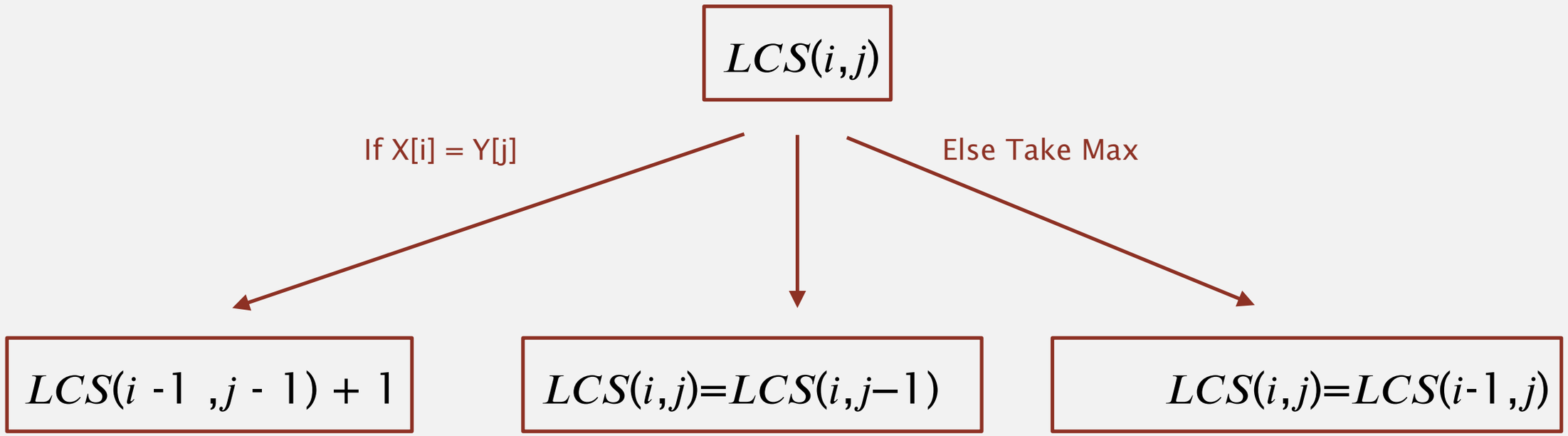
How would we implement this

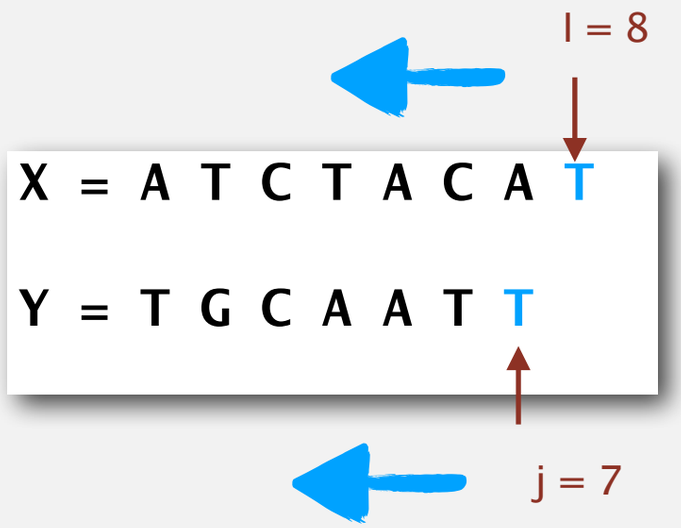
```
public static int lcs(int i, int j, String X, String Y)
{




}
```

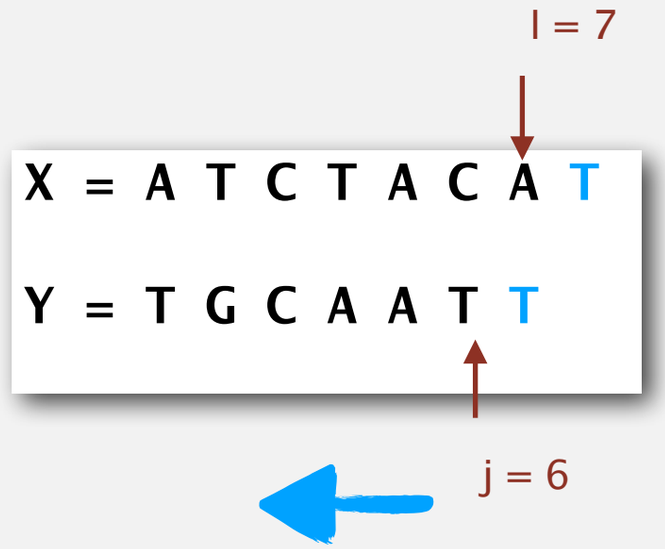
How would we implement this

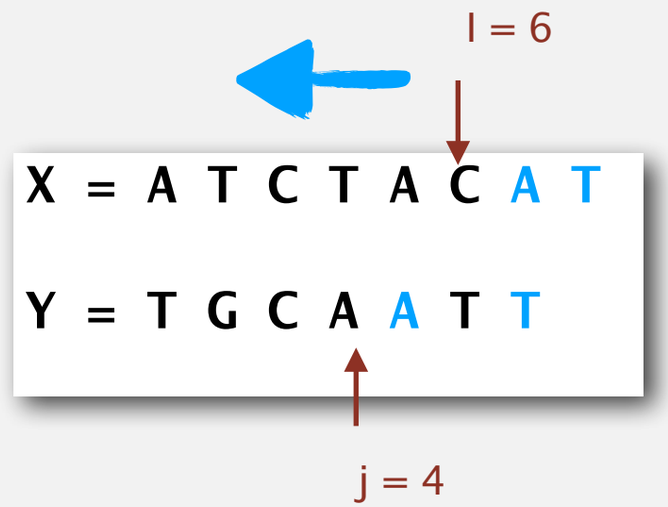
```
public static int lcs(int i, int j, String X, String Y)
{
    if( i == 0 || j == 0 )
        return 0;
    else{
       if(X.charAt(i) == Y.charAt(j))
          return 1 + lcs(i - 1, j - 1, X, Y);
        else{
            int moveJ = lcs(i, j-1, X, Y);
            int moveI = lcs(i-1, j, X, Y);
            return moveJ > moveI ? moveJ : moveI;
           }

        }
}
```

Recursive Tree Structure









Save recursive calls by storing the result LCS(i,j)

How would we implement this

```
public static int lcs(int i, int j, String X, String Y, int[][] mem)
{
    if( i == 0 || j == 0)
        return 0;
    else{
       if (mem[i][j] != 0)
           return men[i][j]
       if(X.charAt(i) == Y.charAt(j)){
          mem[i][j] =  1 + lcs(i - 1, j - 1, X, Y, mem);
          return mem[i][j]
          }
        else{
             int moveJ = lcs(i, j-1, X, Y);
             int moveI = lcs(i-1, j, X, Y);
             mem[i][j] = moveJ > moveI ? moveJ : moveI;
             return mem[i][j];
            }

        }
}
```

Bottom up approach

An iterative approach that goes from smallest to largest

To compute cell (i, j) in our memory

- We need (i-1, j-1)
- We also need (i, j-1)
- And we need (i-1, j)

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i\text{ -1 },j\text{ - 1}) + 1$$

X\Y	0	1	2
0			
1			
2			
3			

Case 2:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i,j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i\text{-}1,j)$$

Bottom up approach

An iterative approach that goes from smallest to largest

To compute cell (i, j) in our memory

```
X = T A
Y = A T A
```

- We need (i-1, j-1)
- We also need (i, j-1)
- And we need (i-1, j)

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

			T	A
	X\Y	0	1	2
	0			
A	1			
T	2			
A	3			

Case 2:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i,j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1,j)$$

Bottom up approach

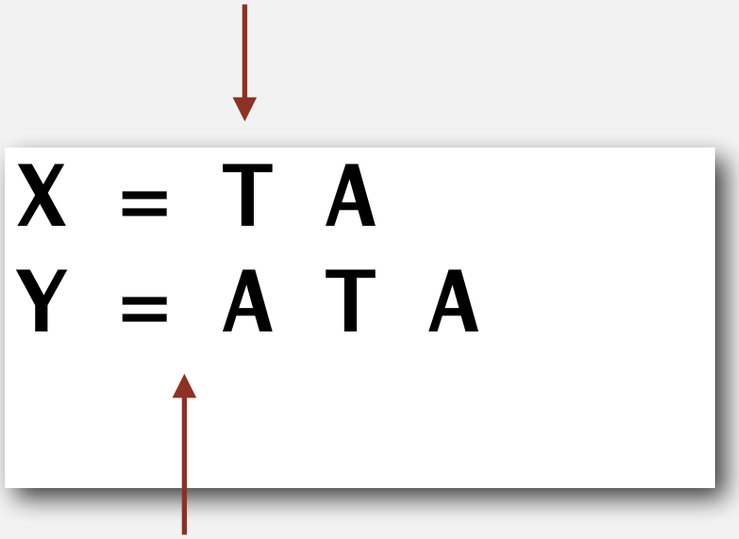
- We need (i-1, j-1) [1,0]
- We also need (i, j-1) [2, 0]
- And we need (i-1, j) [1,1]

Consider case were I = 2, j = 1

```
X = T A
Y = A T A
```

			T	A
	X\Y	0	1	2
	0	0		
A	1			
T	2			
A	3			

Bottom up approach



			T	A
	X\Y	0	1	2
	0	0	0	
A	1			
T	2			
A	3			

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

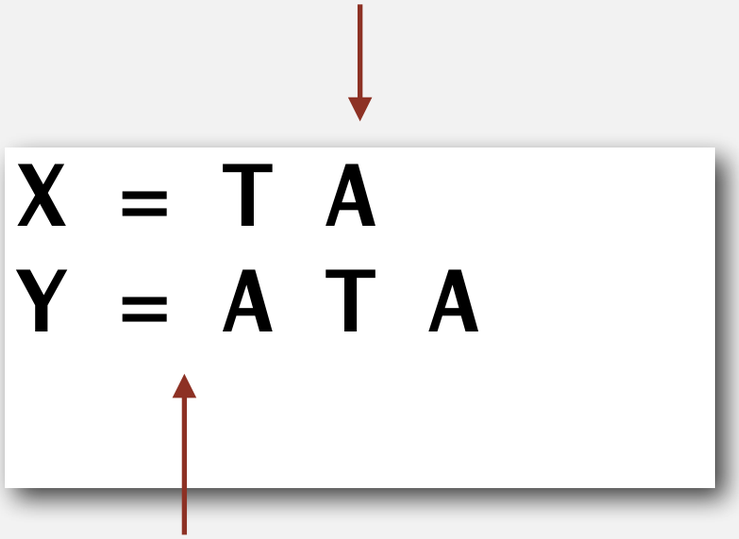
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



			T	A
	X\Y	0	1	2
	0	0	0	0
A	1			
T	2			
A	3			

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

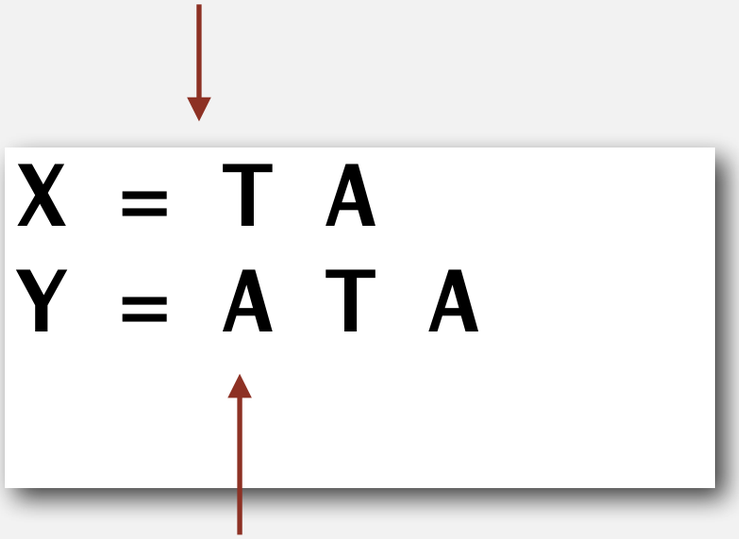
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0		
T	2			
A	3			

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

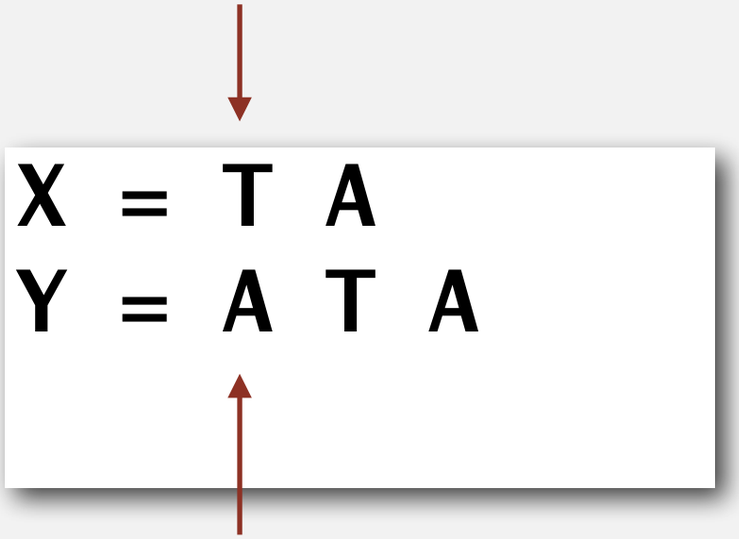
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



Don't Match

			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0	0	
T	2			
A	3			

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

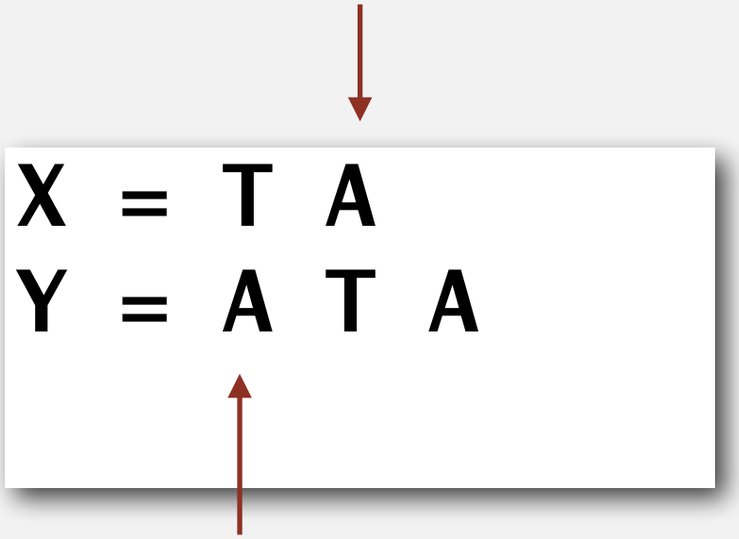
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Match 1 + LCS(i-1, j-1)

Case 2:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i, j-1)$$

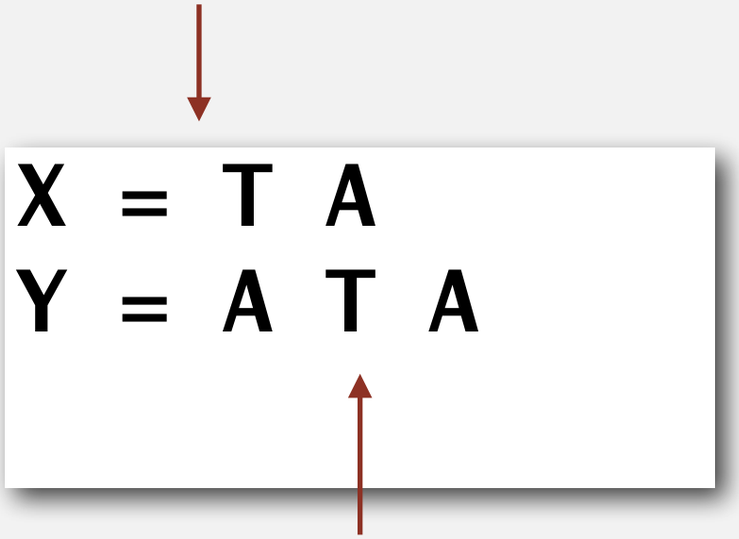
			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0	0	1
T	2			
A	3			

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



No Match

			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0		
A	3			

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

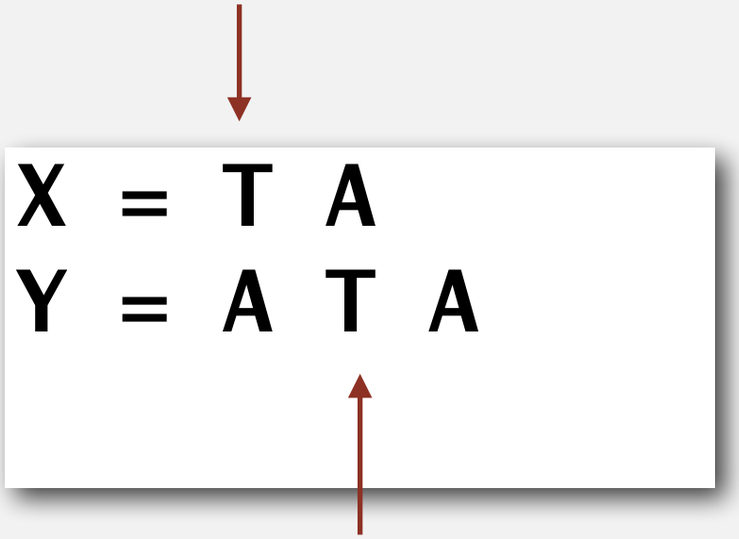
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	
A	3			

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Match 1 + LCS(i-1, j-1)

Case 2:

X[i] ≠ Y[j]

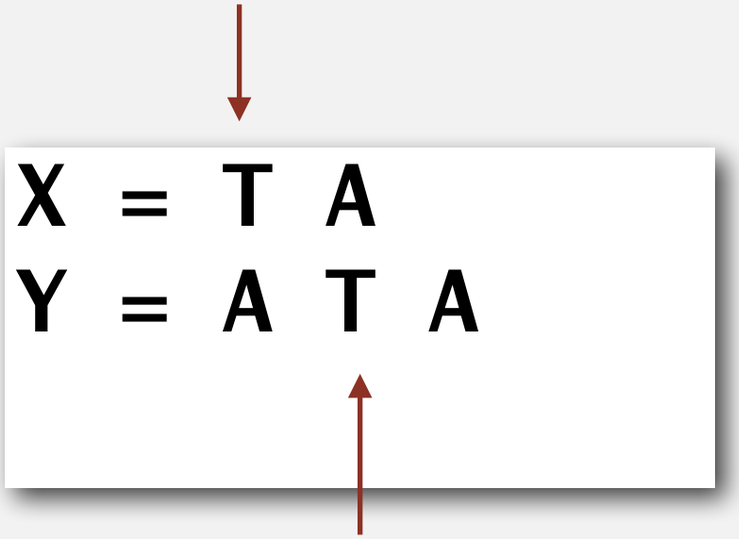
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Match 1 + LCS(i-1, j-1)

Case 2:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i, j-1)$$

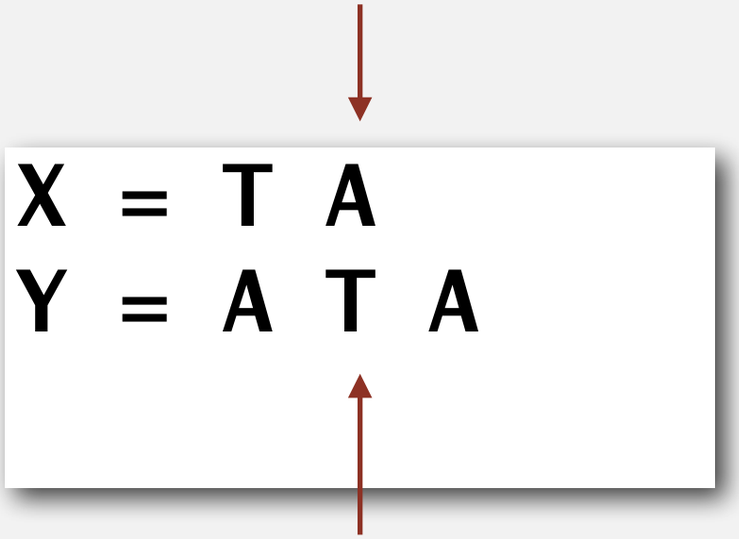
			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	
A	3			

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



No Match so

Pick the max

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i, j-1)$$

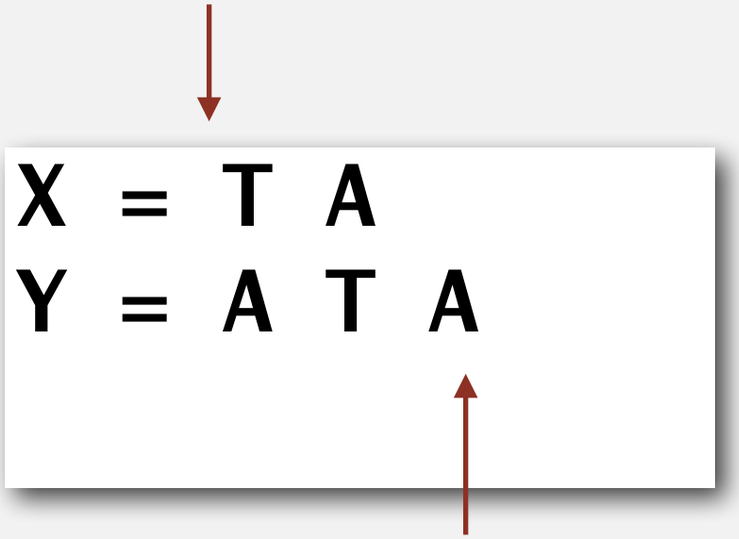
			T	A
	X\Y	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3			

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0		

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

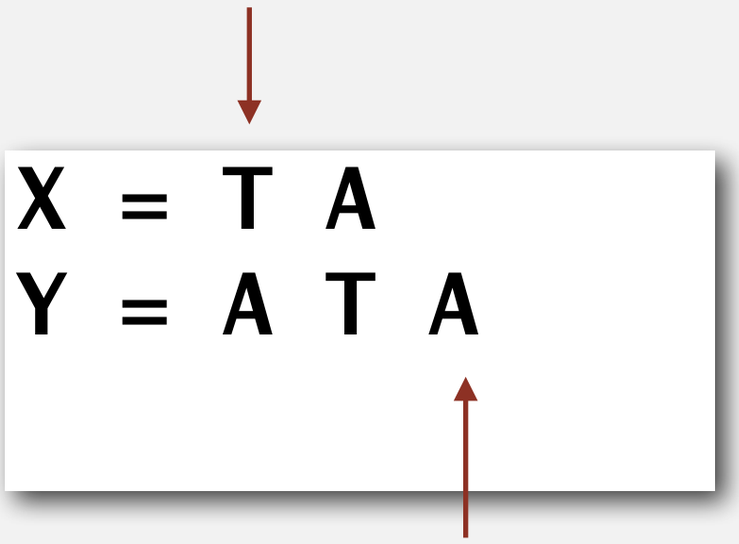
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



No Match so

Pick the max

			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0	1	

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

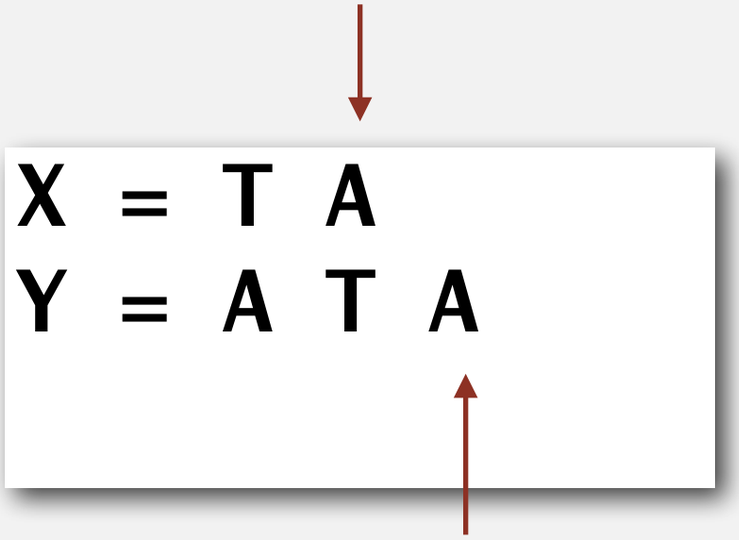
$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Bottom up approach



No Match so

Pick the max

			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0	1	2

Case 1

X[i] = Y[j]

$$LCS(i,j) = LCS(i-1, j-1) + 1$$

Case 2:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i, j-1)$$

Case 3:

X[i] ≠ Y[j]

$$LCS(i,j) = LCS(i-1, j)$$

Reconstruct the sequence

What about reconstructing the sequence

Repeat until it a zero:

- Start at the bottom right
- If chars match push to stack char and move diagonally
- Else go to largest adjacent (If equal pick one: means multiple sequences)

			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0	1	2

Reconstruct the sequence

What about reconstructing the sequence

Repeat until it a zero:

- Start at the bottom right
- If chars match push to stack and move diagonally
- Else go to largest adjacent (If equal pick one: means multiple sequences)

			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0	1	2

Reconstruct the sequence

What about reconstructing the sequence

Repeat until it a zero:

- Start at the bottom right
- If chars match push to stack and move diagonally
- Else go to largest adjacent (If equal pick one: means multiple sequences)

			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0	1	2

A

Reconstruct the sequence

What about reconstructing the sequence

Repeat until it a zero:

- Start at the bottom right
- If chars match print char and move diagonally
- Else go to largest adjacent (If equal pick one: means multiple sequences)

			T	A
	Y\X	0	1	2
	0	0	0	0
A	1	0	0	1
T	2	0	1	1
A	3	0	1	2

T A

Quiz

Consider the table below how many sequences are there

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

$Y =$		$X =$	A	T	C	T	G	A	T
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
T	1	0	0	1	1	1	1	1	1
G	2	0	0	1	1	1	2	2	2
C	3	0	0	1	2	2	2	2	2
A	4	0	1	1	2	2	2	3	3
T	5	0	1	2	2	3	3	3	4
A	6	0	1	2	2	3	3	4	4

Quiz

Consider the table below how many sequences are there

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1,j-1)+1 & \text{if } X[i] = Y[j] \\ \max(LCS(i,j-1), LCS(i-1,j)) & \text{otherwise} \end{cases}$$

			$X =$	A	T	C	T	G	A	T
			0	1	2	3	4	5	6	7
		0	0	0	0	0	0	0	0	0
	T	1	0	0	1	1	1	1	1	1
	G	2	0	0	1	1	1	2	2	2
	C	3	0	0	1	2	2	2	2	2
	A	4	0	1	1	2	2	2	3	3
	T	5	0	1	2	2	3	3	3	4
	A	6	0	1	2	2	3	3	4	4

Quiz

Consider the table below how many sequences are there

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } X[i] = Y[j] \\ \max(LCS(i, j-1), LCS(i-1, j)) & \text{otherwise} \end{cases}$$

			$X =$	A	T	C	T	G	A	T
		j	0	1	2	3	4	5	6	7
	i									
	0		0	0	0	0	0	0	0	0
T	1		0	0	1	1	1	1	1	1
G	2		0	0	1	1	1	2	2	2
C	3		0	0	1	2	2	2	2	2
A	4		0	1	1	2	2	2	3	3
T	5		0	1	2	2	3	3	3	4
A	6		0	1	2	2	3	3	4	4

http://algs4.cs.princeton.edu

4.4 Shortest Paths

- APIs
- shortest-paths properties
- Dijkstra's algorithm
- edge-weighted DAGs
- negative weights

Content-aware resizing

Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.

http://www.youtube.com/watch?v=vIFCV2spKtg

Content-aware resizing

Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.

In the wild. Photoshop CS 5, Imagemagick, GIMP, ...

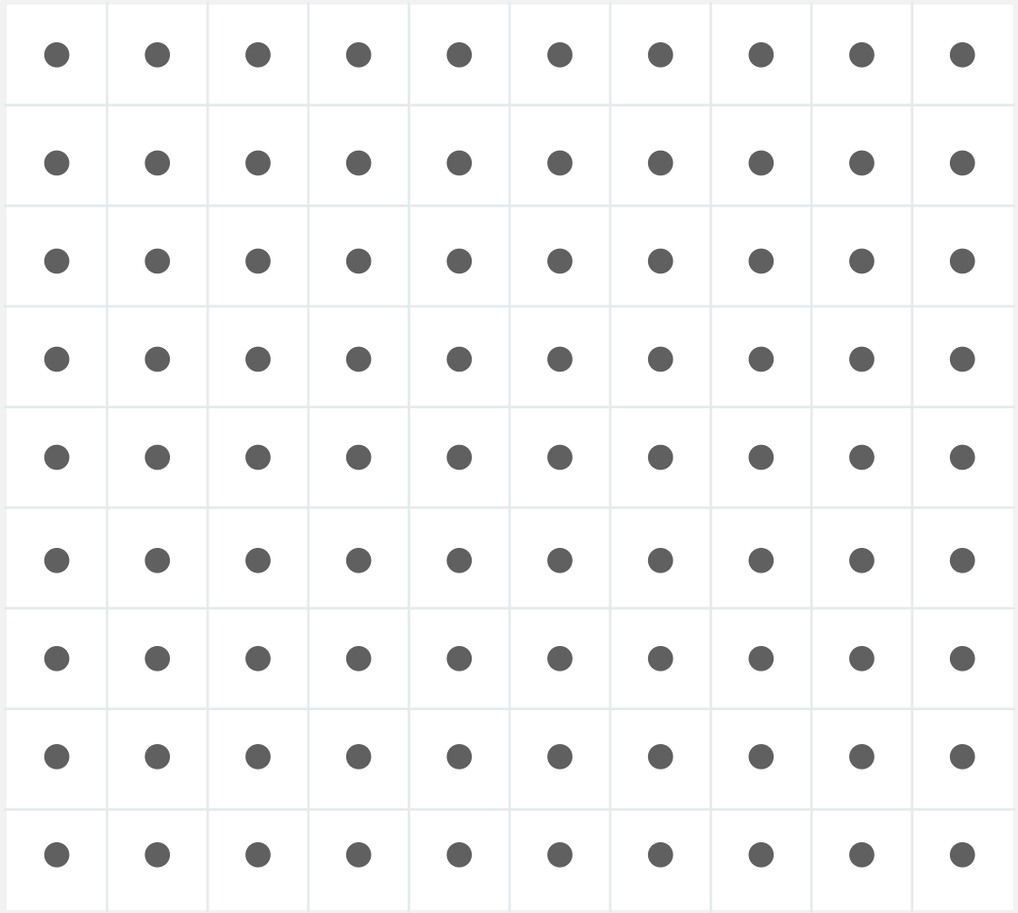
Cropped vs Scaled vs Seam Carved

Cropped

Content-aware resizing

To find vertical seam:

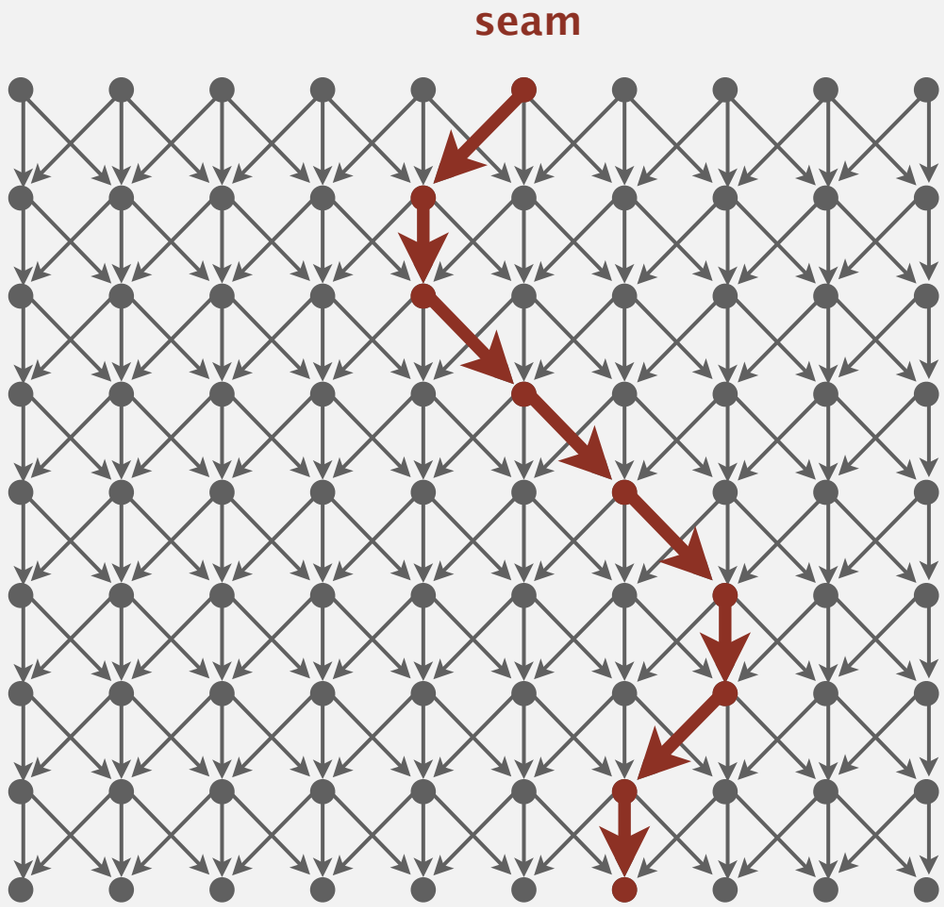
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To find vertical seam:

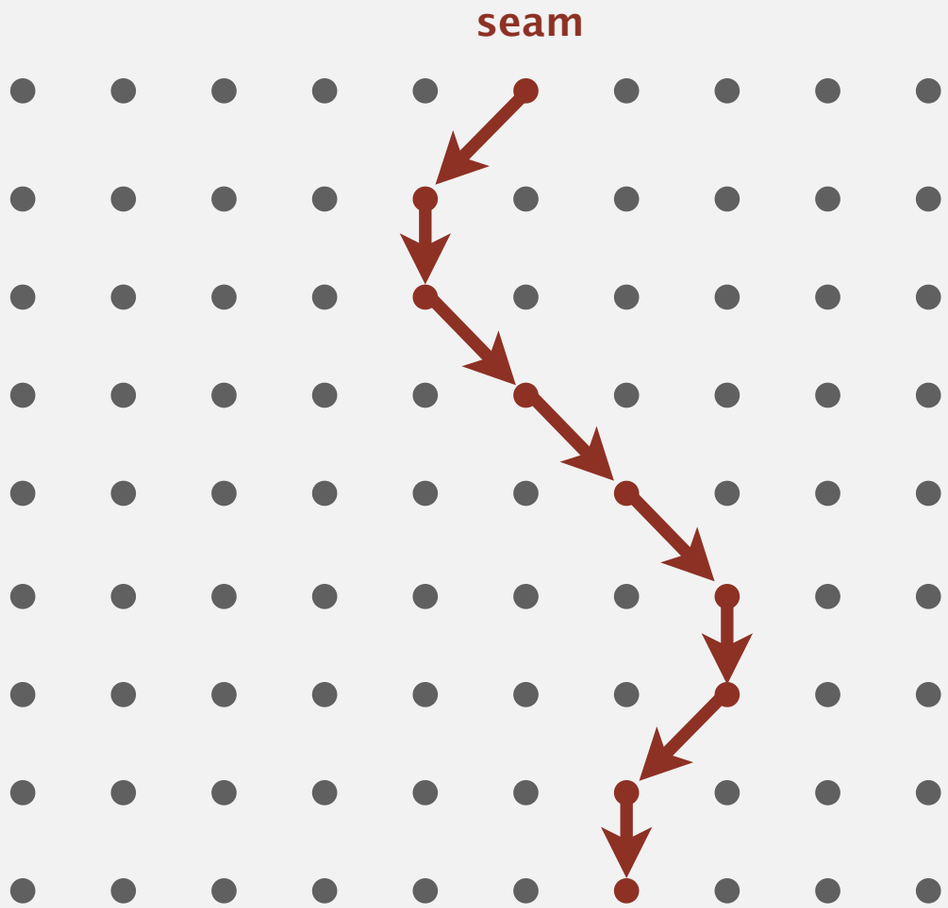
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To remove vertical seam:

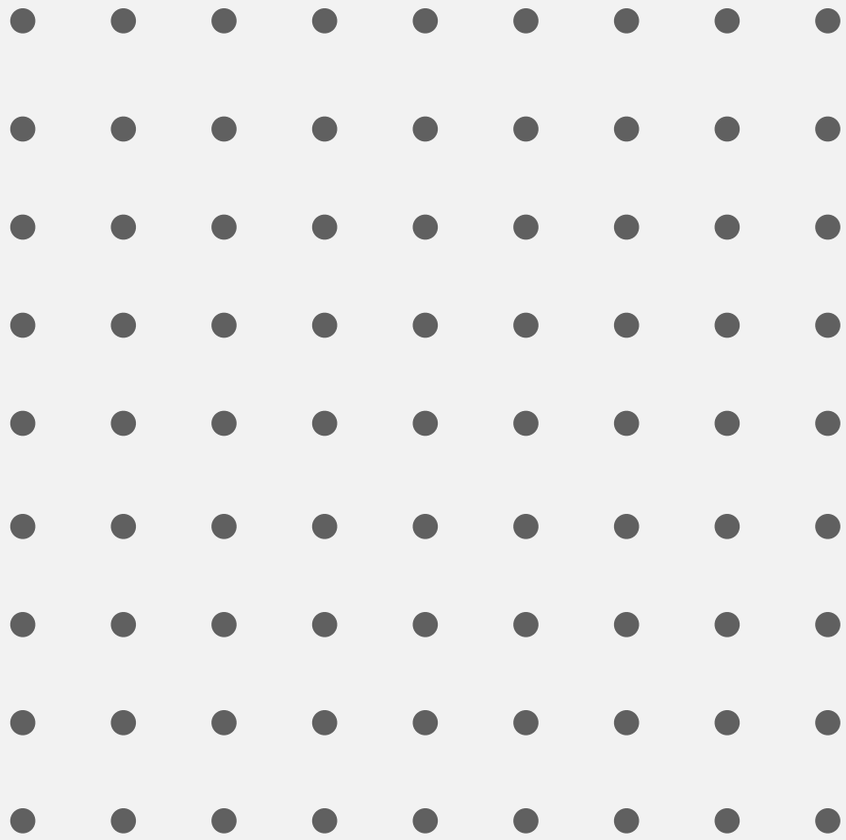
- Delete pixels on seam (one in each row).



Content-aware resizing

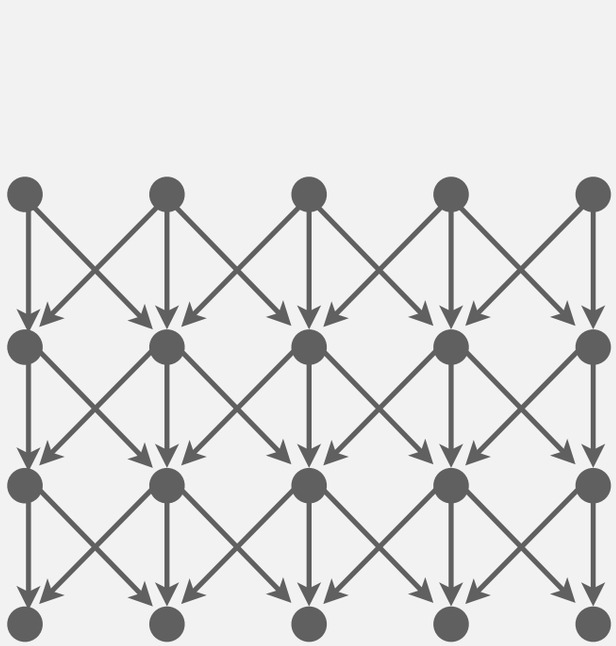
To remove vertical seam:

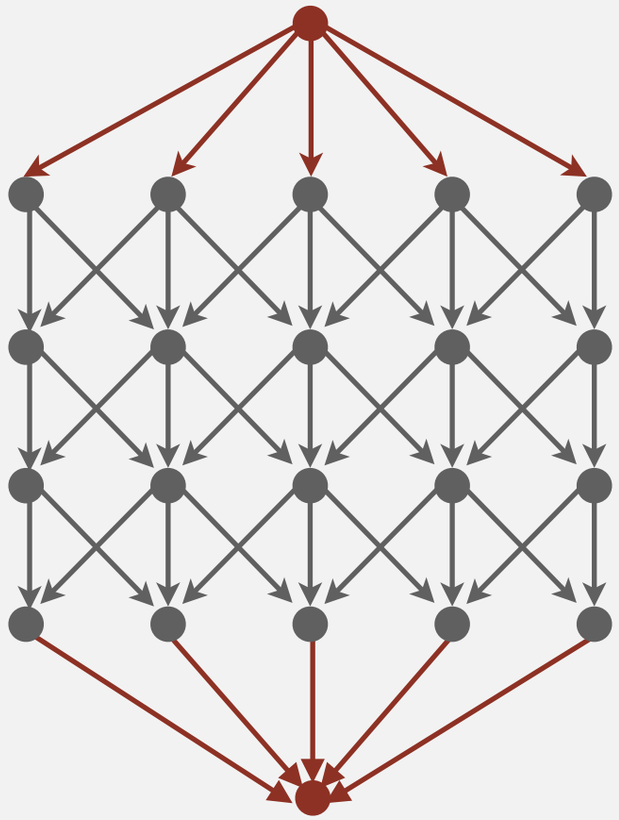
- Delete pixels on seam (one in each row).



Shortest path variants

Q2. How to model multiple sources and sinks?





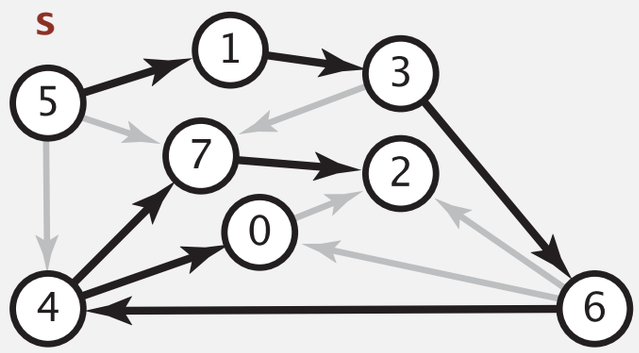
Longest paths in edge-weighted DAGs

Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
- Find shortest paths.
- Negate weights in result.

equivalent: reverse sense of equality in `relax()`

longest paths input		shortest paths input	
5->4	0.35	5->4	-0.35
4->7	0.37	4->7	-0.37
5->7	0.28	5->7	-0.28
5->1	0.32	5->1	-0.32
4->0	0.38	4->0	-0.38
0->2	0.26	0->2	-0.26
3->7	0.39	3->7	-0.39
1->3	0.29	1->3	-0.29
7->2	0.34	7->2	-0.34
6->2	0.40	6->2	-0.40
3->6	0.52	3->6	-0.52
6->0	0.58	6->0	-0.58
6->4	0.93	6->4	-0.93

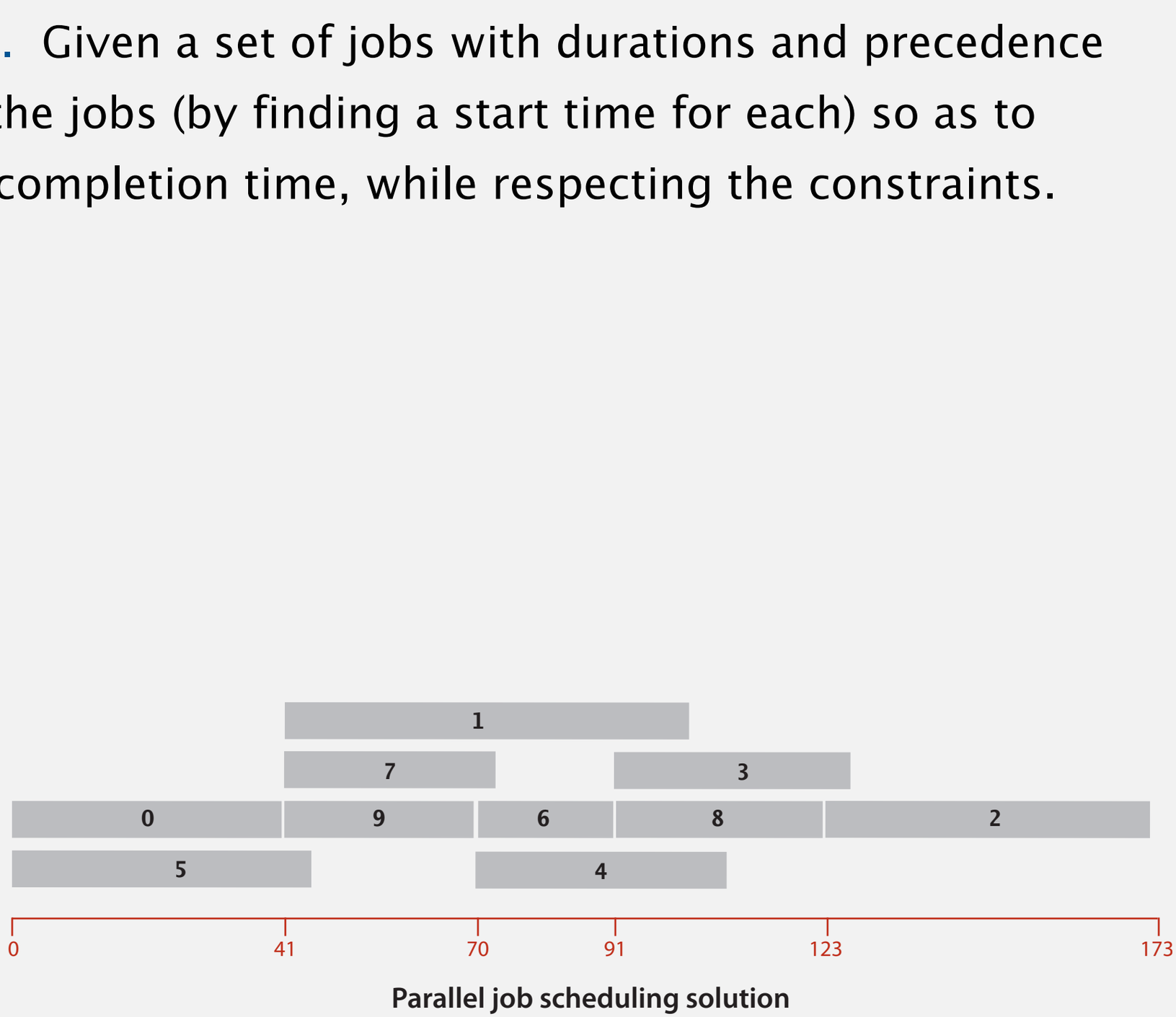


Key point. Topological sort algorithm works even with negative weights.

Longest paths in edge-weighted DAGs: application

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

job	*duration*	*must complete before*		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	



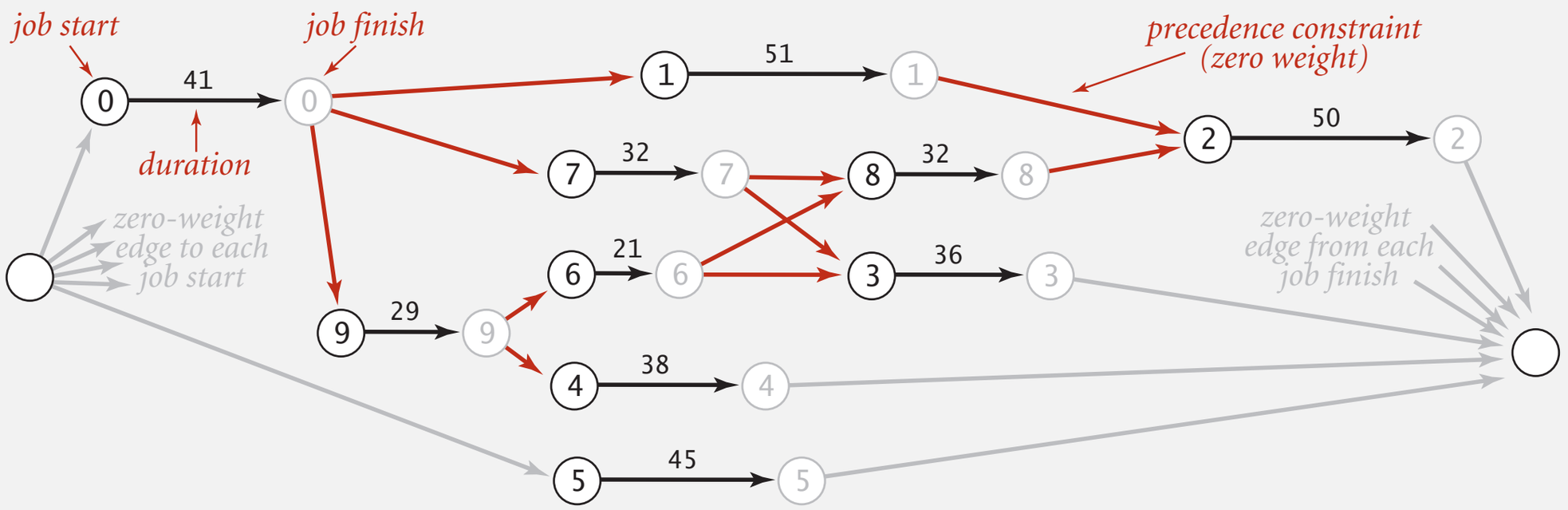
Parallel job scheduling solution

Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

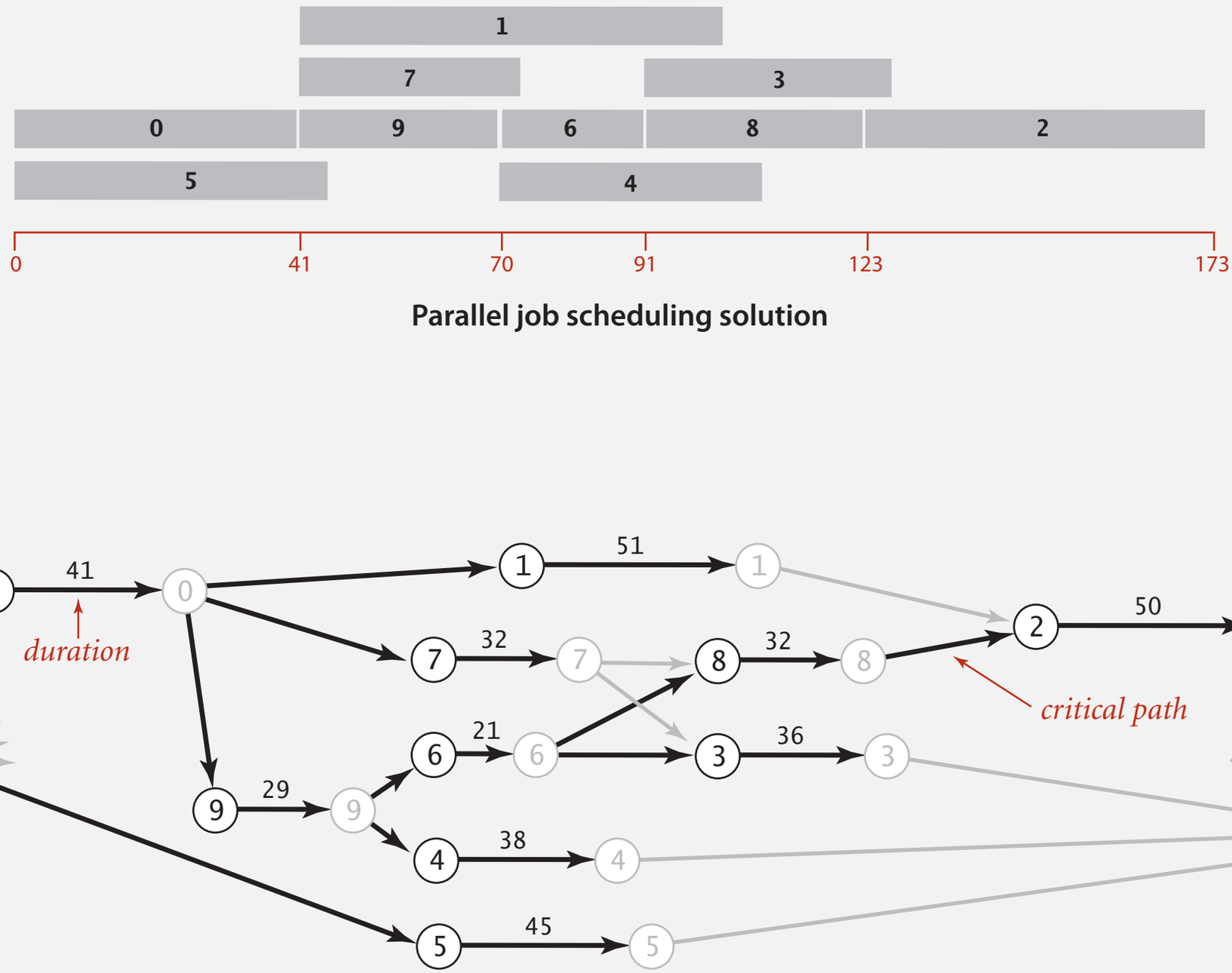
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
  - begin to end (weighted by duration)
  - source to begin (0 weight)
  - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

job	*duration*	*must complete before*		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	



Critical path method

CPM. Use longest path from the source to schedule each job.



Longest repeated substring

Given a string of N characters, find the longest repeated substring.

```
a a c a a g t t t a c a a g c a t g a t g c t g t a c t a
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a
c c t a a t c c t t g t g t g t a c a c a c a c t a c t a
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a
a c c g g a a g g c c g g a c a a g g c g g g g g g t a t
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a
t a g a t c t a g c t a g c t a g c t c a t c g a t a c a
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a
g a c a g a a a a a a a a a c t c t a t a t c t a t a a a a
```

Applications. Bioinformatics, cryptanalysis, data compression, ...

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

Brute-force algorithm.

- Try all indices i and j for start of possible match.
- Compute longest common prefix (LCP) for each pair.

Analysis. Running time $\leq D N^2$, where D is length of longest match.

Longest repeated substring: a sorting solution

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

form suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

sort suffixes to bring repeated substrings together

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
11	a	a	g	c											
3	a	a	g	t	t	t	a	c	a	a	g	c			
9	a	c	a	a	g	c									
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
12	a	g	c												
4	a	g	t	t	t	a	c	a	a	g	c				
14	c														
10	c	a	a	g	c										
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
13	g	c													
5	g	t	t	t	a	c	a	a	g	c					
8	t	a	c	a	a	g	c								
7	t	t	a	c	a	a	g	c							
6	t	t	t	a	c	a	a	g	c						

compute longest prefix between adjacent suffixes

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Longest repeated substring: Java 6 implementation

```
public String lrs(String s)
{
  int N = s.length();

  String[] suffixes = new String[N];
  for (int i = 0; i < N; i++)
     suffixes[i] = s.substring(i, N);

  Arrays.sort(suffixes);

  String lrs = "";
  for (int i = 0; i < N-1; i++)
  {
     int len = lcp(suffixes[i], suffixes[i+1]);
     if (len > lrs.length())
        lrs = suffixes[i].substring(0, len);
  }
  return lrs;
}
```

create suffixes (linear time and space)

sort suffixes

find LCP between adjacent suffixes in sorted order

```
% java LRS < mobydick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```