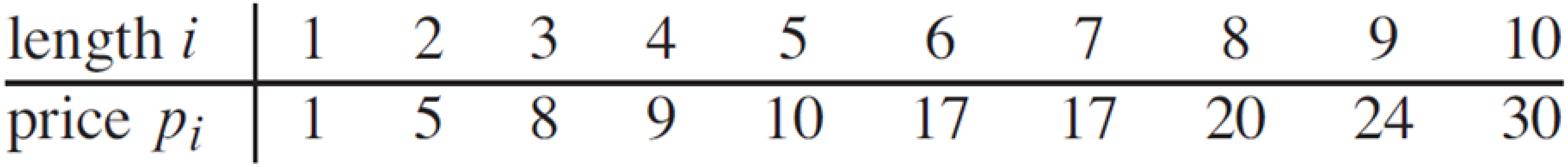
Design & Analysis of Algorithms -I

Lecture 18

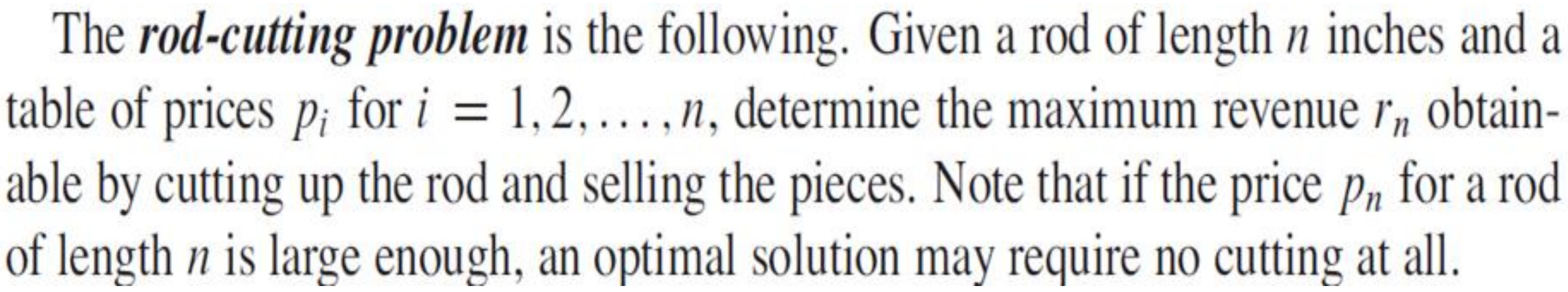
Rod Cutting & Rock Climbing

Rod Cutting

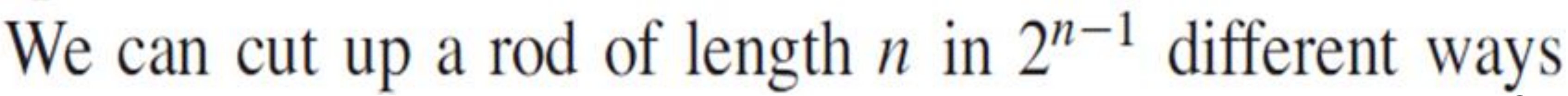
- Sterling Enterprises buys long steel rods and cuts them into shorter rods
  - which it then sells.
- Each cut is free. The management of Serling enterprises wants to know the best way to cut up the rods.



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.



We can cut up a rod of length n in 2^{n-1} different ways

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

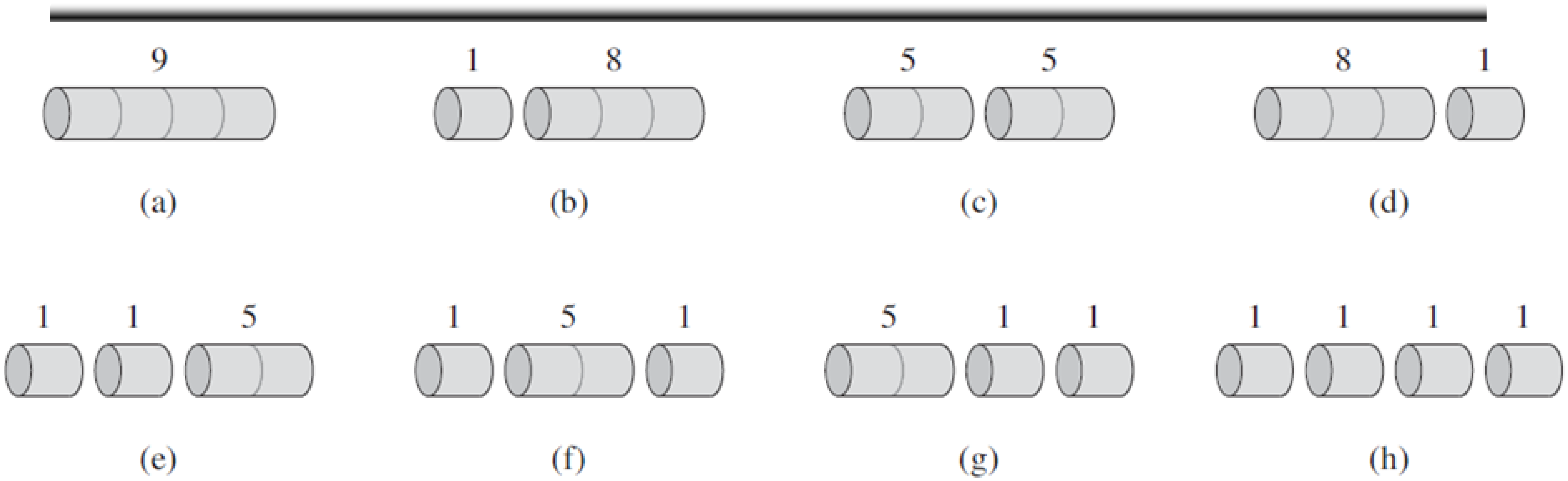


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

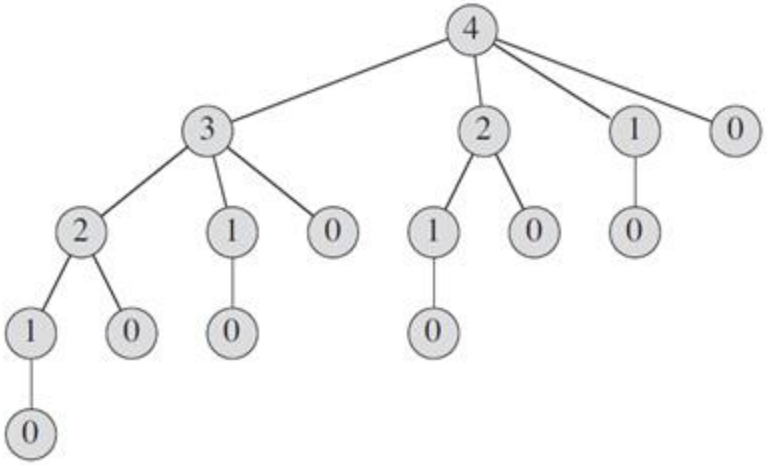
We can cut up a rod of length n in 2^{n-1} different ways.

Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- For n = 40, the program takes at least several minutes, and most likely more than an hour.
- In fact, each time you increase n by 1, your program's running time would approximately double.
- this recursion tree has 2^n nodes and 2^{n-1} leaves.

```
CUT-ROD(p, n)
1  if $n == 0$
2      return 0
3  $q = -\infty$
4  for $i = 1$ to $n$
5      $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
6  return $q$
```



We can cut up a rod of length n in 2^{n-1} different ways.

Dynamic programming

- Step 1: Describe an array of values you want to compute.
- Step 2: Give a recurrence for computing later values from earlier (bottom-up).
- Step 3: Give a high-level program.
- Step 4: Show how to use values in the array to compute an optimal solution.

Rod Cutting

we have to calculate what is the better:
not to make the cut or make the cut

$$S[i][j] = \begin{cases} \text{if } j = 0 \text{ then } 0 \\ \text{if } i = 0 \text{ then } 0 \\ \max\{\, S[i-1][j] \,;\, p[i] + S[i][j-i] \,\} \text{ if } i \le j \\ S[i-1][j] \text{ if } i > j \end{cases}$$

the total value when
*we have the first **i** pieces*
*and the total length is **j***

if the piece is greater than the length
of the rod of course we skip it
(we do not make the cut- so we try to get the
*max revenue with **i-1** cuts and the **j** length is unchanged)*

Rod Cutting

N = 5m

piece #1	l_1 = 1m	p_1 = \$2
piece #2	l_2 = 2m	p_2 = \$5
piece #3	l_3 = 3m	p_3 = \$7
piece #4	l_4 = 4m	p_4 = \$3
Piece #5	l_5 = 5m	p_5 = \$9

		0	1	2	3	4	5	**length [m]**
no pieces	**0**							
[piece #1]	**1**							
[piece #1 and #2]	**2**							
[piece #1, #2, #3]	**3**							
[piece #1, #2, #3, #4]	**4**							
all pieces	**5**							

Rod Cutting

N = 5m

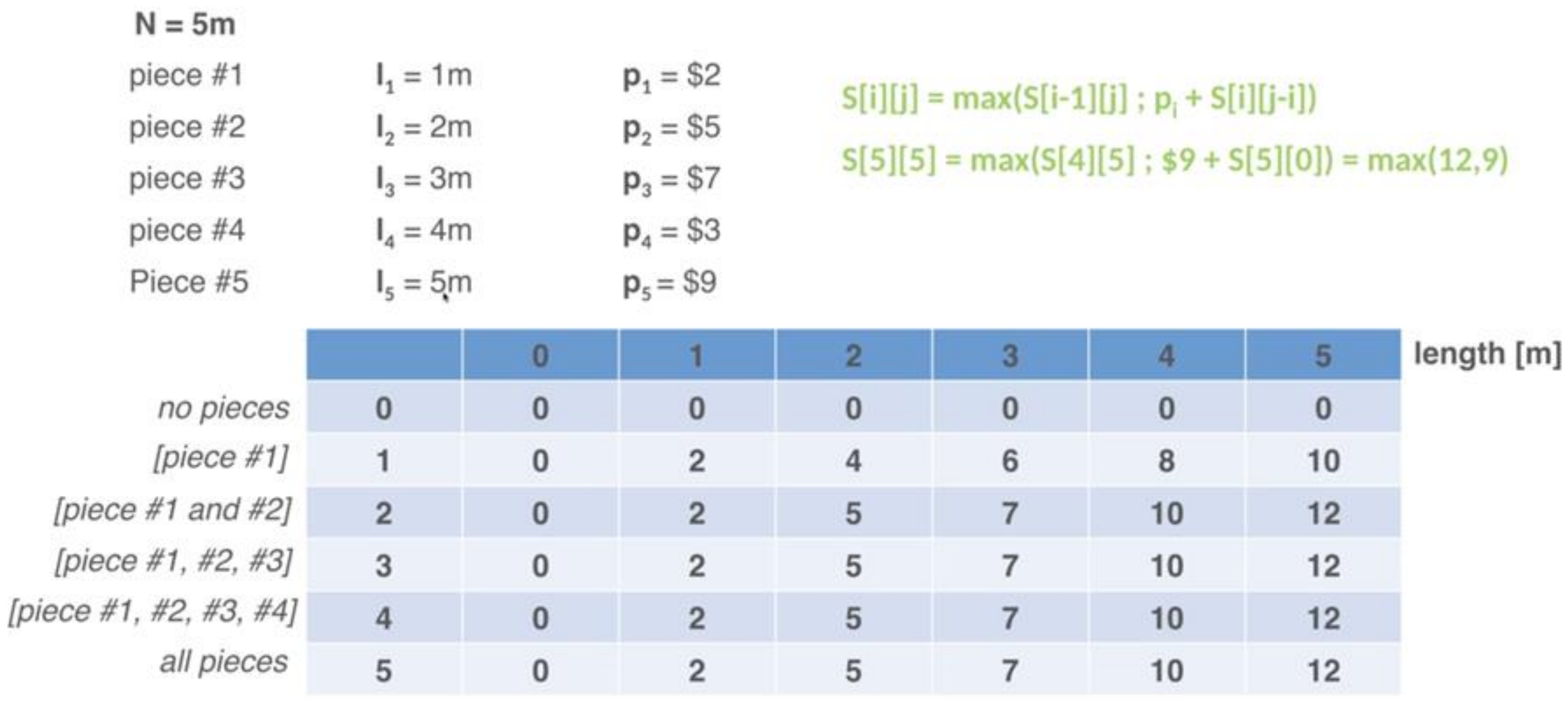
piece #1	l_1 = 1m	p_1 = \$2
piece #2	l_2 = 2m	p_2 = \$5
piece #3	l_3 = 3m	p_3 = \$7
piece #4	l_4 = 4m	p_4 = \$3
Piece #5	l_5 = 5m	p_5 = \$9

S[i][j] = max(S[i-1][j] ; p_i + S[i][j-i])

S[2][1] = max(S[1][1] ; \$5 + S[2][-1]) = max(2,0)

		0	1	2	3	4	5	length [m]
no pieces	0	0	0	0	0	0	0	
[piece #1]	1	0	2	4	6	8	10	
[piece #1 and #2]	2	0	2					
[piece #1, #2, #3]	3	0						
[piece #1, #2, #3, #4]	4	0						
all pieces	5	0						

Rod Cutting - how to cut?



N = 5m

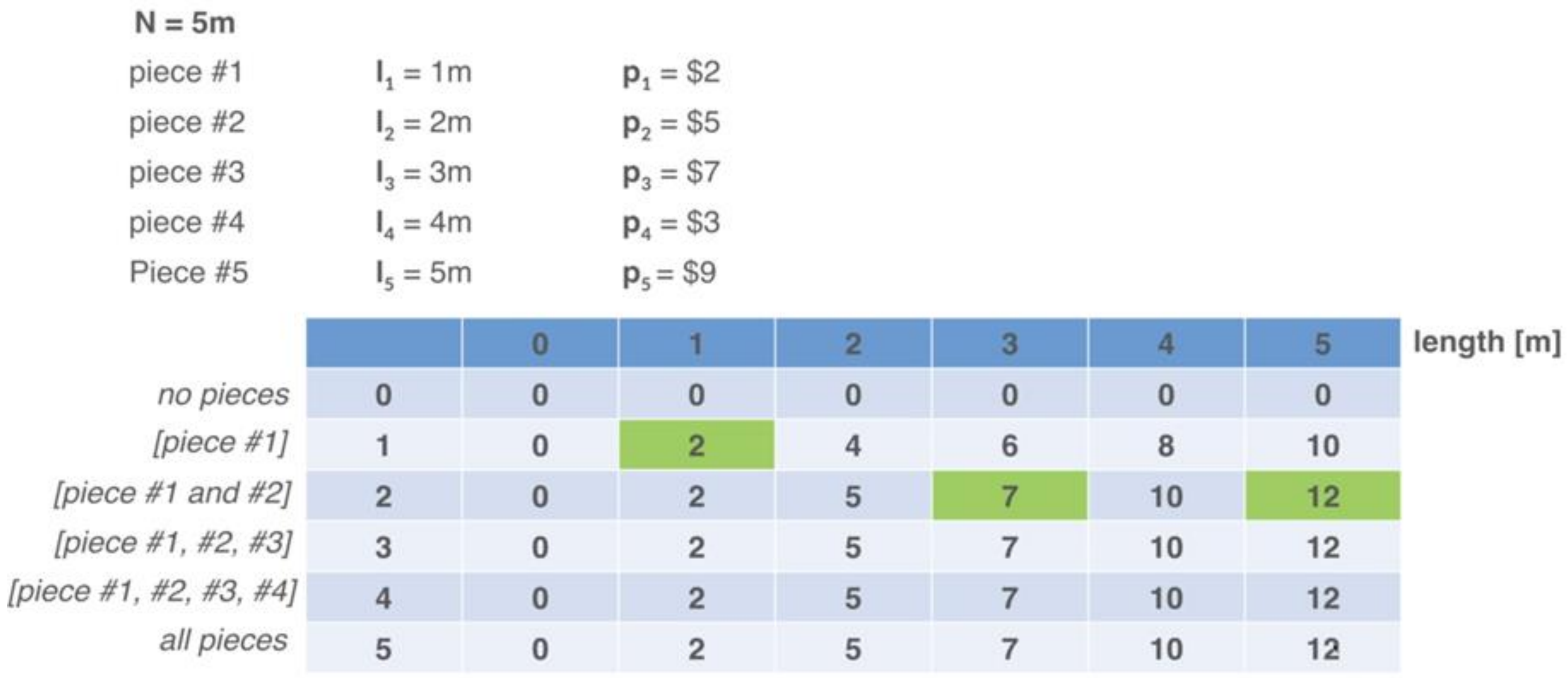
piece #1	l_1 = 1m	p_1 = \$2
piece #2	l_2 = 2m	p_2 = \$5
piece #3	l_3 = 3m	p_3 = \$7
piece #4	l_4 = 4m	p_4 = \$3
Piece #5	l_5 = 5m	p_5 = \$9

S[i][j] = max(S[i-1][j] ; p_i + S[i][j-i])

S[5][5] = max(S[4][5] ; \$9 + S[5][0]) = max(12,9)

		0	1	2	3	4	5	length [m]
no pieces	0	0	0	0	0	0	0	
[piece #1]	1	0	2	4	6	8	10	
[piece #1 and #2]	2	0	2	5	7	10	12	
[piece #1, #2, #3]	3	0	2	5	7	10	12	
[piece #1, #2, #3, #4]	4	0	2	5	7	10	12	
all pieces	5	0	2	5	7	10	12	

Rod Cutting - how to cut?



N = 5m

piece #1	l_1 = 1m	p_1 = \$2
piece #2	l_2 = 2m	p_2 = \$5
piece #3	l_3 = 3m	p_3 = \$7
piece #4	l_4 = 4m	p_4 = \$3
Piece #5	l_5 = 5m	p_5 = \$9

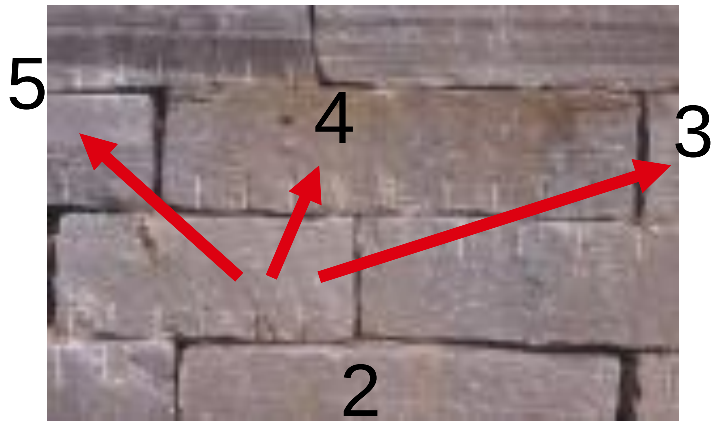
		0	1	2	3	4	5	length [m]
no pieces	0	0	0	0	0	0	0	
[piece #1]	1	0	2	4	6	8	10	
[piece #1 and #2]	2	0	2	5	7	10	12	
[piece #1, #2, #3]	3	0	2	5	7	10	12	
[piece #1, #2, #3, #4]	4	0	2	5	7	10	12	
all pieces	5	0	2	5	7	10	12	

Rock Climbing Problem

- A rock climber wants to get from the bottom of a rock to the top by the safest possible path.
- At every step, he reaches for handholds above him; some holds are safer than other.
- From every place, he can only reach a few nearest handholds.

Rock climbing (cont)

❖Suppose we have a wall instead of the rock.



At every step our climber can reach exactly three handholds: above, above and to the right and above and to the left.

There is a table of “danger ratings” provided. The “Danger” of a path is the sum of danger ratings of all handholds on the path.

Rock Climbing (cont)

- We represent the wall as a table.

- Every cell of the table contains the danger rating of the corresponding block.

2	8	9	5	8
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

The obvious greedy algorithm does not give an optimal solution.The rating of this path is 13.

The rating of an optimal path is 12.

However, we can solve this problem by a dynamic programming strategy in polynomial time.

Idea: once we know the rating of a path to every handhold on a layer, we can easily compute the ratings of the paths to the holds on the next layer.

For the top layer, that gives us an answer to the problem itself.

For every handhold, there is only one "path" rating. Once we have reached a hold, we don't need to know how we got there to move to the next level.

This is called an "optimal substructure" property. Once we know optimal solutions to subproblems, we can compute an optimal solution to the problem itself.

Recursive solution:

To find the best way to get to stone j in row i, check the cost of getting to the stones

- (i-1,j-1),
- (i-1,j) and
- (i-1,j+1), and take the cheapest.

Problem: each recursion level makes three calls for itself, making a total of 3^n calls – too much!

Solution - memorization

We query the value of A(i,j) over and over again.

Instead of computing it each time, we can compute it once, and remember the value.

A simple recurrence allows us to compute A(i,j) from values below.

Dynamic programming

- Step 1: Describe an array of values you want to compute.
- Step 2: Give a recurrence for computing later values from earlier (bottom-up).
- Step 3: Give a high-level program.
- Step 4: Show how to use values in the array to compute an optimal solution.

Rock climbing: step 1.

- *Step 1: Describe an array of values you want to compute.*
- For $1 \le i \le n$ and $1 \le j \le m$, define $A(i,j)$ to be the cumulative rating of the least dangerous path from the bottom to the hold (i,j).
- The rating of the best path to the top will be the minimal value in the last row of the array.

Rock climbing: step 2.

- *Step 2: Give a recurrence for computing later values from earlier (bottom-up).*
- Let C(i,j) be the rating of the hold *(i,j)*. There are three cases for *A(i,j)*:
- Left *(j=1)*: $C(i,j)+\min\{A(i-1,j),A(i-1,j+1)\}$
- Right *(j=m)*: $C(i,j)+\min\{A(i-1,j-1),A(i-1,j)\}$
- Middle: $C(i,j)+\min\{A(i-1,j-1),A(i-1,j),A(i-1,j+1)\}$
- For the first row *(i=1)*, $A(i,j)=C(i,j)$.

Rock climbing: simpler step 2

- Add initialization row: $A(0,j)=0$. No danger to stand on the ground.
- Add two initialization columns: $A(i,0)=A(i,m+1)=\infty$. It is infinitely dangerous to try to hold on to the air where the wall ends.
- Now the recurrence becomes, for every i,j:

$$A(i,j) = C(i,j)+min\{A(i-1,j-1),A(i-1,j),A(i-1,j+1)\}$$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞						∞
2	∞						∞
3	∞						∞
4	∞						∞

Initialization: $A(i,0)=A(i,m+1)=\infty$,
$A(0,j)=0$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞						∞
3	∞						∞
4	∞						∞

The values in the first row are the same as C(i,j).

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7					∞
3	∞						∞
4	∞						∞

A(2,1)=5+min{∞,3,2}=7

.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9				∞
3	∞						∞
4	∞						∞

$A(2,1)=5+\min\{\infty,3,2\}=7.$

$A(2,2)=7+\min\{3,2,5\}=9$

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7			∞
3	∞						∞
4	∞						∞

A(2,1)=5+min{∞,3,2}=7.
A(2,2)=7+min{3,2,5}=9
A(2,3)=5+min{2,5,4}=7.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞						∞
4	∞						∞

The best cumulative rating on the second row is 5.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞						∞

The best cumulative rating on the third row is 7.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The best cumulative rating on the last row is 12.

Rock climbing: example

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The best cumulative rating on the last row is 12.

So the rating of the best path to the top is 12.

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	5	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	2	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The last hold was (4,4).

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	1
4	4	6	**2**	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

The hold before the last was (3,4), since min{13,7,8} was 7.

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	4	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

The hold before that was (2,5), since min{7,10,5} was 5.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	**4**	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

Finally, the first hold was (1,4), since min{5,4,8} was 4.

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

To find the actual path we need to retrace backwards the decisions made during the calculation of A(i,j).

Rock climbing example: step 4

C(i,j):

3	2	5	**4**	8
5	7	5	6	**1**
4	4	6	**2**	3
2	8	9	**5**	8

A(i,j):

i\j	0	1	2	3	4	5	6
0	∞	0	0	0	0	0	∞
1	∞	3	2	5	4	8	∞
2	∞	7	9	7	10	5	∞
3	∞	11	11	13	7	8	∞
4	∞	13	19	16	12	15	∞

We are done!

Printing out the solution recursively

```
PrintBest(A,i,j) // Printing the best path ending at (i,j)
  if (i==0) OR (j=0) OR (j=m+1)
      return;
  if (A[i-1,j-1]<=A[i-1,j]) AND (A[i-1,j-1]<=A[i-1,j+1])
      PrintBest(A,i-1,j-1);
  elseif (A[i-1,j]<=A[i-1,j-1]) AND (A[i-1,j]<=A[i-1,j+1])
      PrintBest(A,i-1,j);
  elseif (A[i-1,j+1]<=A[i-1,j-1]) AND (A[i-1,j+1]<=A[i-1,j])
      PrintBest(A,i-1,j+1);
  printf(i,j)
```

Sum of Subset Problem

- Problem:
  - Suppose you are given N positive integer numbers A[1…N] and it is required to produce another number K using a subset of A[1..N] numbers. How can it be done using Dynamic programming approach?
- Example:

  N = 6, A[1..N] = {2, 5, 8, 12, 6, 14}, K = 19

  Result: 2 + 5 + 12 = 19

Coin Change Problem

- Suppose you are given ***n*** types of coin - C_1, C_2, … , C_n coin, and another number ***K.***
- Is it possible to make K using above types of coin?
  - Number of each coin is infinite
  - Number of each coin is finite
- Find minimum number of coin that is required to make ***K***?
  - Number of each coin is infinite
  - Number of each coin is finite

Maximum-sum interval

- Given a sequence of real numbers $a_1a_2\dots a_n$, find a consecutive subsequence with the maximum sum.

```
9 −3 1 7 −15 2 3 −4 2 −7 6 −2 8 4 -9
```

For each position, we can compute the maximum-sum interval starting at that position in $O(n)$ time. Therefore, a naive algorithm runs in $O(n^2)$ time.

Try Yourself