Dmitri Stanchevici
Unit 3, Module 4

Ex. 4.1

Printed is **14.**

Ex. 4.2

At the beginning of main

stack record for main		argv
		return address, booking info

After the call to methodA

stack record for methodA	3	parameter i
	4	parameter j
	1	local k
	8	temp
		return address, booking info
stack record for main		argv
		return address, booking info

FIRST call to methodB

stack record for methodB	8	parameter n
	9	temp
	9	m
		return address, booking info
stack record for methodA	3	parameter i
	4	parameter j
	1	local k
	8	temp
		return address, booking info
stack record for main		argv
		return address, booking info

Return to methodA

stack record for methodA	5	parameter i
	4	parameter j
	10	local k
	19	temp
		return address, booking info
stack record for main		argv
		return address, booking info

SECOND Call to methodB

stack record for methodB	19	parameter n
	20	temp
	20	m
		return address, booking info
stack record for methodA	5	parameter i
	4	parameter j
	10	local k
	19	temp
		return address, booking info
		return address, booking info
stack record for main		argv
		return address, booking info

Return to methodA

stack record for methodA	5	parameter i
	4	parameter j
	20	local k
		return address, booking info
stack record for main		argv
		return address, booking info

Return to main

stack record for main		argv
		return address, booking info

Ex. 4.6

Recursive stack diagram for 4^3

			p = 1 b = 0 a = 4			
		p = b = 1 a = 4	p = b = 1 a = 4	p = 4 b = 1 a = 4		
	p = b = 2 a = 4	p = b = 2 a = 4	p = b = 2 a = 4	p = b = 2 a = 4	p = 16 b = 2 a = 4	
p = b = 3 a = 4	p = b = 3 a = 4	p = b = 3 a = 4	p = b = 3 a = 4	p = b = 3 a = 4	p = b = 3 a = 4	p = 64 b = 3 a = 4
Beginning of First Call	Beginning of Second Call	Beginning of Third Call	Fourth Call	End of Third	End of Second	End of First

---

Ex. 4.11

Here is the output:

Intermediate result: 3^0=1
Intermediate result: 3^1=3
Intermediate result: 3^2=9
Intermediate result: 3^3=27
Intermediate result: 3^4=81
3^4 = 81

As we can see, the recursive calls bring b to 0 at which point the recursion starts cascading back, multiplying the returned value by a=3. Thus we end up with the result of 3 to the power of 4.

Making a a global variable does not affect the result because this variable is accessible from anywhere in the program, including from any recursion of power(int b).

Ex. 4.12

STEP 1

<table>
<tr><td>Stack for power()</td><td>Global variables a and b</td></tr>
<tr><td><table><tr><td>Call from main</td><td></td></tr></table></td><td>a = 3
b = 4</td></tr>
</table>

STEP 2

<table>
<tr><td>Stack for power()</td><td>Global variables a and b</td></tr>
<tr><td><table><tr><td>First Recursion</td><td></td></tr><tr><td>Call from main</td><td></td></tr></table></td><td>a = 3
b = 3</td></tr>
</table>

STEP 3

Stack for power()		Global variables a and b
Second Recursion		a = 3 b = 2
First Recursion		
Call from main		

STEP 4

Stack for power()		Global variables a and b
Third Recursion		a = 3 b = 1
Second Recursion		
First Recursion		
Call from main		

STEP 5

Stack for power()		Global variables a and b
Fourth Recursion	p = 1	a = 3 b = 0
Third Recursion		
Second Recursion		
First Recursion		
Call from main		

STEP 6

Stack for power()		Global variables a and b
Third Recursion	p = 3	a = 3 b = 0

Second Recursion		
First Recursion		
Call from main		

STEP 7

Stack for power()		Global variables a and b
Second Recursion	p = 9	a = 3 b = 0
First Recursion		
Call from main		

STEP 8

Stack for power()		Global variables a and b
First Recursion	p = 27	a = 3 b = 0
Call from main		

STEP 9

Stack for power()		Global variables a and b
Call from main	p = 81	a = 3 b = 0

Every recursive call decreases the global b by one (b - 1). When the bottom-out case (b == 0) is reached and the recursions start cascading back, returning p multiplied by a=3, the global b is no longer used and it remains 0.

Important: b does not affect the calculation of p after each recursion; it simply keeps count of how many recursions (and thus multiplications by 3) are needed. This explains the printout of System.out.println ("Intermediate result: " + a + "^" + b + "=" + p);

Intermediate result: 3^0=1
Intermediate result: 3^0=3
Intermediate result: 3^0=9
Intermediate result: 3^0=27
Intermediate result: 3^0=81

So, the logic works here. What is problematic is the presentation of the intermediate results. There should be some other way of keeping track of the exponent to be printed out in the intermediate result.

Ex. 4.14

Below is the output of the print command above the return with the base case marked by **if (b == 0) return 1;**

b=4
b=3
b=2
b=1
3^4 = 81

b decrements by one until it bottoms out at 0. Then the recursions cascade back, but the print command no longer prints b because it appears before the recursive call to power ().

With the base case commented out, **b** continues to decrease until it reaches -5205, at which point **Exception in thread "main" java.lang.StackOverflowError** is thrown multiple times.

Ex. 4.16

Array search with recursion

STEP 1

Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 2

First Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 3

Second Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=2
First Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 4

Third Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=3
Second Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=2
First Recursion	Parameters:

	A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 5

Fourth Recursion: Base Case Reached b/c A[index] == value Return TRUE	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=4
Third Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=3
Second Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=2
First Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 6

Back in Third Recursion Return TRUE	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42

	index=3
Second Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=2
First Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 7

Back in Second Recursion Return TRUE	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=2
First Recursion	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 8

Back in First Recursion Return TRUE	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=1
Call from main	Parameters:

	A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

STEP 9

Back in Call from main Return TRUE to main	Parameters: A ={51, 24, 63, 73, 42, 85, 71, 41, 87, 32} value = 42 index=0

Ex. 4.17

Will the index grow beyond 9, thus exceeding the array bounds?

On implementing with value=1, I get the **Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 10**.

Ex. 4.18

str=”river”;

STEP 1

Call from main	str=”river”

STEP 2

First Recursion Reached the bottom out case str.charAt(0) != str.charAt(str.length()-1) Return “is not a palindrome”	str=”ive”
Call from main	str=”river”

STEP 3

Back in Call from main Return “is not a palindrome” to main	str=”river”

Ex. 4.21

Trace recursions in **binarySearch**.

searchTerm = 32
Unchanged on stack are:

- A = {24, 32, 41, 42, 51, 63, 71, 73, 85, 87}
- value = 32

Stack	start = 0 end = 9 mid = 4	start = 0 end = 4 mid = 2 start = 0 end = 9 mid = 4	start = 0 end = 2 mid = 1 start = 0 end = 4 mid = 2 start = 0 end = 9 mid = 4	start = 0 end = 1 start = 0 end = 2 mid = 1 start = 0 end = 4 mid = 2 start = 0 end = 9 mid = 4	start = 0 end = 2 mid = 1 start = 0 end = 4 mid = 2 start = 0 end = 9 mid = 4	start = 0 end = 4 mid = 2 start = 0 end = 9 mid = 4	start = 0 end = 9 mid = 4
Steps	As called from main: value < A[mid] (51)	First Recursion: value < A[mid] (41)	Second Recursion: value == A[mid] (32)	Third Recursion: Bottom-out case A[end] == value is reached. Start cascading back. Return true.	Back in Second Return true.	Back in First. Return true.	Back in as called from main. Return true.

searchTerm = 55
Unchanged on stack are:

- A = {24, 32, 41, 42, 51, 63, 71, 73, 85, 87}
- value = 32

Stack	start = 0 end = 9 mid = 4	start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 5 end = 7 mid = 6 start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 5 end = 6 mid = 5 start = 5 end = 7 mid = 6 start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 5 end = 5 start = 5 end = 6 mid = 5 start = 5 end = 7 mid = 6 start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 5 end = 6 mid = 5 start = 5 end = 7 mid = 6 start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 5 end = 7 mid = 6 start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 5 end = 9 mid = 7 start = 0 end = 9 mid = 4	start = 0 end = 9 mid = 4
Steps	As called from main: value > A[mid] (51)	First Recursion: value < A[mid] (73)	Second Recursion: value < A[mid] (71)	Third Recursion: value < A[mid] (63)	Fourth Recursion: Bottom-out case **start == end** is reached. Start cascading back. Return false.	Back in Third Return false.	Back in Second Return false.	Back in First. Return false.	Back in as called from main. Return false.

searchTerm = 88
With a value outside the range, such as 88, the code does not break down. It returns the correct **false**. Its bottom-out case is **start == end**.

Ex. 4.22

Compare binary search to simple search in SearchComparison.java

Array Size	Number of Comparisons in Simple Search	Number of Comparisons in Binary Search
10	10	7
100	100	15
1,000	215	18
10,000	10,000	29

Ex. 4.24

Trace printPermutations().

ENTER FROM MAIN

As entered from main()	numSpaces = 3 numRemaining = 2 seats = [0, 0, 0] person = 1 ENTER FOR LOOP i = 0 seats[0] = 1

<table>
<tr>
<td></td>
<td>
<table>
<tr>
<td>First Recursion</td>
<td>STACK
numSpaces = 2
numRemaining = 1
seats = [1, 0, 0]
person = 2

ETNER FOR LOOP
i=0 Skip
i=1
seats[1] = 2
<table>
<tr>
<td>Second Rec-n</td>
<td>STACK
numSpaces = 1
numRemaining = 0
seats = [1, 2, 0]
person = 3

Bottom-out
Print: [1, 2, 0]
RETURN TO FIRST</td>
</tr>
</table>
seats[1]=0
i=2
seats[2]=2
<table>
<tr>
<td>Third Rec-n</td>
<td>STACK
numSpaces = 1
numRemaining = 0
seats = [1, 0, 2]
person = 3

Bottom-out
Print: [1, 0, 2]
RETURN TO FIRST</td>
</tr>
</table>
seats[2]=0
RETURN TO AS ENTERED FROM MAIN[]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</table>
seats[0]=0
i = 1
seats[1] = 1
</td>
</tr>
</table>

<table>
<tr><td></td><td>
<table>
<tr><td>Fourth Recursion</td><td>STACK
numSpaces = 2
numRemaining = 1
seats = [0, 1, 0]
person = 2

ETNER FOR LOOP
i=0
seats[0] = 2
<table>
<tr><td>Fifth Rec-n</td><td>STACK
numSpaces = 1
numRemaining = 0
seats = [1, 2, 0]
person = 3

Bottom-out
Print: [2, 1, 0]
RETURN TO FOURTH</td></tr>
</table>
seats[0]=0
i=1 Skip
i=2
seats[2]=2
<table>
<tr><td>Sixth Rec-n</td><td>STACK
numSpaces = 1
numRemaining = 0
seats = [0, 1, 2]
person = 3

Bottom-out
Print: [0, 1, 2]
RETURN TO FOURTH</td></tr>
</table>
seats[2]=0
RETURN TO AS ENTERED FROM MAIN[]</td></tr>
</table>
seats[1]=0
i=2
seats[2]=1</td></tr>
</table>

Seventh Recursion

STACK
numSpaces = 2
numRemaining = 1
seats = [0, 0, 1]
person = 2

ETNER FOR LOOP
i=0
seats[0] = 2

Eighth Rec-n

STACK
numSpaces = 1
numRemaining = 0
seats = [2, 0, 1]
person = 3

Bottom-out
Print: [2, 0, 1]
RETURN TO SEVENTH

seats[0]=0
i=1
seats[1]=2

Ninth Rec-n

STACK
numSpaces = 1
numRemaining = 0
seats = [0, 2, 1]
person = 3

Bottom-out
Print: [0, 2, 1]
RETURN TO SEVENTH

seats[1]=0
i=2
RETURN TO AS ENTERED FROM MAIN[]

seats[2]=0

Ex. 4.25

For the case K=2 and M=5, the number of arrangements is **20**.

Ex. 4.26

Version 2 works because in Recursion 0 (as entered from main), 1 is placed in seats 0 through 2 by going through a for loop each iteration of which creates a fresh copy of seats. The recursions from each of these iterations then place 2 in the other two positions in turn.

<table>
<tr><td>RECURSION 0
numSpaces=3
numRemaining=2
seats=[0, 0, 0]
person=1

Enter loop
i=0
seatsCopy=[0, 0, 0]
seatsCopy[0]=1</td><td></td><td></td></tr>
<tr><td></td><td>RECURSION 0.0

numSpaces=2
numRemaining=1
seats=[1, 0, 0]
person=2

Enter loop
i=0 SKIP
i=1
seatsCopy=[1, 0, 0]
seatsCopy[1]=2</td><td></td></tr>
<tr><td></td><td></td><td>RECURSION 0.0.0

numSpaces=1
numRemaining=0
seats=[1, 2, 0]
person=3

Bottom-out
Print: 1, 2, 0

RETURN TO 0.0</td></tr>
<tr><td></td><td>RECURSION 0.0

i=2
seatsCopy=[1, 0, 0]
seatsCopy[2]=2</td><td></td></tr>
<tr><td></td><td></td><td>RECURSION 0.0.1

numSpaces=1
numRemaining=0
seats=[1, 0, 2]</td></tr>
</table>

		person=3 Bottom-out Print: 1, 0, 2 RETURN TO 0.0
	RECURSION 0.0 RETURN TO 0	
RECURSION 0 i=1 seatsCopy=[0, 0, 0] seatsCopy[1]=1		
	RECURSION 0.1 numSpaces=2 numRemaining=1 seats=[0, 1, 0] person=2 Enter loop i=0 seatsCopy=[0, 1, 0] seatsCopy[0]=2	
		RECURSION 0.1.0 numSpaces=1 numRemaining=0 seats=[2, 1, 0] person=3 Bottom-out Print: 2, 1, 0 RETURN TO 0.1
	RECURSION 0.1 i=1 SKIP i=2 seatsCopy=[0, 1, 0] seatsCopy[2] = 2	
		RECURSION 0.1.1 numSpaces=1 numRemaining=0 seats=[0, 1, 2] person=3 Bottom-out Print: 0, 1, 2 RETURN TO 0.1
	RECURSION 0.1 RETURN TO 0	

RECURSION 0 i=2 seatsCopy=[0, 0, 0] seatsCopy[2]=1		
	RECURSION 0.2 numSpaces=2 numRemaining=1 seats=[0, 0, 1] person=2 Enter loop i=0 seatsCopy=[0, 0, 1] seatsCopy[0]=2	
		RECURSION 0.2.0 numSpaces=1 numRemaining=0 seats=[2, 0, 1] person=3 Bottom-out Print: 2, 0, 1 RETURN TO 0.2
	RECURSION 0.2 i=1 seatsCopy=[0, 0, 1] seatsCopy[1]=2	
		RECURSION 0.2.1 numSpaces=1 numRemaining=0 seats=[0, 2, 1] person=3 Bottom-out Print: 0, 2, 1 RETURN TO 0.2
	RECURSION 0.2 i=2 SKIP RETURN TO 0TH	
RECURSION 0 DONE		

Using System.currentTimeMillis (), I measured the time of each version. Below are the results:

Version 1 (no array copy): TIME = 21
Version 2 (with array copy): TIME=16

Thus, Version 2 (with array copying) is more efficient.

Ex. 4.29

- Logically, once the edge of the grid is reached (either horizontal OR vertical), there is only one way to the destination. This is why we use || in the base-case if statement. If we used AND (&&), we would get off the grid and never reach the destination, unless the starting position had either numRows=0 or numCols=0.
- I ran the program, and with r=5 and c=3, it resulted in **56** paths.

Counting the ways from 0,0 to 2,2:

0 numRows=2 numCols=2 downCount=			
	0.0 numRows=1 numCols=2 downCount=		
		0.0.0 numRows=0 numCols=2 Base case Return 1	
	0.0 numRows=1 numCols=2 downCount=1 rightCount=		
		0.0.1 numRows=1 numCols=1 downCount=1 rightCount=	

			0.0.1.0 numRows=1 numCols=0 Base case Return 1
		0.0.1 numRows=1 numCols=1 downCount=1 rightCount=1 Return 1+1=2	
	0.0 numRows=1 numCols=2 downCount=1 rightCount=2 Return 1+2=3		
0 numRows=2 numCols=2 downCount=3 rightCount=			
	0.1 numRows=1 numCols=2 downCount=		
		0.1.0 numRows=0 numCols=2 Base case Return 1	
	0.1 numRows=1 numCols=2 downCount=1 rightCount=		
		0.1.1 numRows=1 numCols=1 downCount=	
			0.1.1.0

			numRows=0 numCols=1 Base Return 1
		0.1.1 numRows=1 numCols=1 downCount=1 rightCount=	
			0.1.1.1 numRows=1 numCols=0 Base Return 1
		0.1.1 numRows=1 numCols=1 downCount=1 rightCount=1 Return 1+1=2	
	0.1 numRows=1 numCols=2 downCount=1 rightCount=2 Return 1+2 = 3		
0 numRows=2 numCols=2 downCount=3 rightCount=3 return 3+3=6			

Ex. 4.32

Trace fibonacci(5)

Recursion 0 n=5 f_n_minus_one=			

	Recursion 1 n=4 f_n_minus_one=		
		Recursion 2 n=3 f_n_minus_one=	
			Recursion 3 n=2 Return 1
		Recursion 2 n=3 f_n_minus_one=1 f_n_minus_two=	
			Recursion 4 n=1 Return 0
		Recursion 2 n=3 f_n_minus_one=1 f_n_minus_two=0 Return 1+0=1	
	Recursion 1 n=4 f_n_minus_one=1 f_n_minus_two=		
		Recursion 5 n=2 Return 1	
	Recursion 1 n=4 f_n_minus_one=1 f_n_minus_two=1 Return 1+1=2		
Recursion 0 n=5 f_n_minus_one=2 f_n_minus_two=			
	Recursion 6 n=3 f_n_minus_one=		
		Recursion 7 n=2 Return 1	
	Recursion 6 n=3 f_n_minus_one=1 f_n_minus_two=		
		Recursion 8 n=1 Return 0	
	Recursion 6 n=3		

	f_n_minus_one=1 f_n_minus_two=0 Return 1+0 = 1		
Recursion 0 n=5 f_n_minus_one=2 f_n_minus_two=1 Return 3			

Ex. 4.33
fibonacci(20) will require many (13,529) recursive calls because decrementing 20 by 1 and then by 2 will generate multiple branches. Each of these branches will in turn also require further multiple double branches, and so on. They will return the required values repeatedly.

Ex. 4.34

Count calls to recursions in ManhattanWithCallCount.java.

Without Stored Values:
r=2 c=2 => n=6; numCalls=11
r=5 c=7 => n=792; numCalls=1583

With Stored Values:
r=2 c=2 => n=6; numCalls=9
r=5 c=7 => n=792; numCalls=71

Tracing fibonacci(n) with Stored Values:

<table>
<tr><td>Global</td><td>Stack</td><td></td><td></td><td></td></tr>
<tr><td>fValues<table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td><td>0
n=5
fV[4]=</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>1
n=4
fV[3]=</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>2
n=3
fV[2]=</td><td></td></tr>
</table>

<table>
<tr>
<td>fValues<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td>
<td></td>
<td></td>
<td></td>
<td>3
n==2
base
fV[2]=1
Return 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2
n=3
fV[2]=1
fV[1]=</td>
<td></td>
</tr>
<tr>
<td>fValues<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td>
<td></td>
<td></td>
<td></td>
<td>4
n=1
base
fV[1]=0
Return 0</td>
</tr>
<tr>
<td>fValues<table><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td>
<td></td>
<td></td>
<td>2
n=3
fV[2]=1
fV[1]=0
fV[3] = 1+0=1
Return fV[3]=1</td>
<td></td>
</tr>
<tr>
<td>fValues<table><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>2</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td>
<td></td>
<td>1
n=4
fV[3]=1
fV[n-2] != 0 SKIP
fV[4]=fV[3]+fV[2]=2
Return fV[4]=2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fValues<table><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>2</td><td>0</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table></td>
<td>0
n=5
fV[4]=2
fV[5-2] != 0 SKIP
fV[5]=fV[4]+fV[3]=
3
Return 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</table>

Trace ManhattanWithCallCount with Stored Values
r=2
c=2

Global	Stack			

{
{0, 0, 0},
{0, 0, 0},
{0, 0, 0}
}
	0
	r=2
	c=2
	down=			

		1 r=1 c=2 down=		
			2 r=0 c=2 base return 1	
		1 r=1 c=2 down=1 right=		
			3 r=1 c=1 down=	
				4 r=0 c=1 base return 1
			3 r=1 c=1 down=1 right=	
				5 r=1 c=0 base return 1

{
{0, 0, 0},
{0, 2, 0},
{0, 0, 0}
}
			3
			r=1
			c=1
			down=1
			right=1
			return 2	

{
{0, 0, 0},
{0, 2, 3},
{0, 0, 0}
}
		1
		r=1
		c=2
		down=1
		right=2
		return 3		

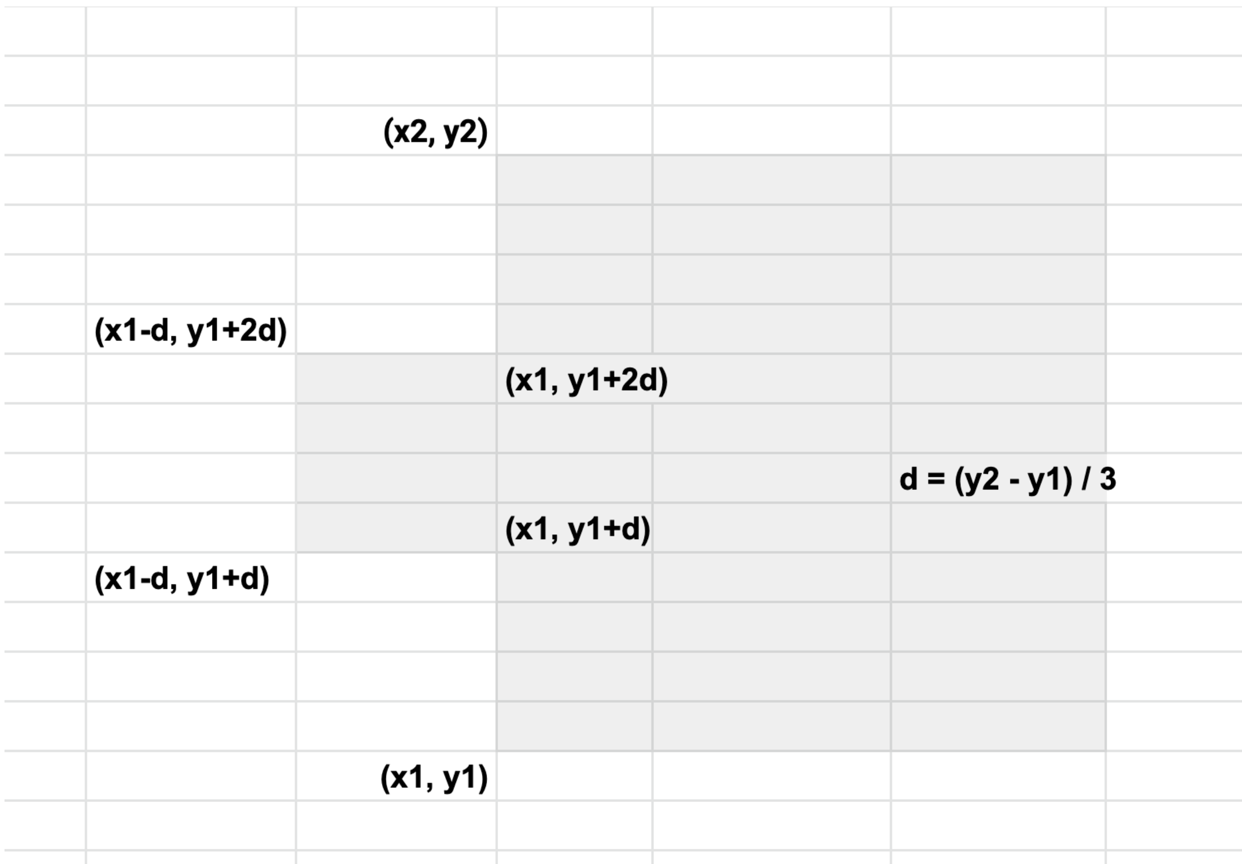
	0 r=2 c=2 down=3 right=			
		6 r=2		

		c=1 down=		
			7 r=1 c=1 base return gridValues [1][1]=2	
		6 r=2 c=1 down=2 right=		
			8 r=2 c=0 base return 1	

{
{0, 0, 0},
{0, 2, 3},
{0, 3, 0}
}
		6
		r=2
		c=1
		down=2
		right=1
		return 3		

{
{0, 0, 0},
{0, 2, 3},
{0, 3, 3}
}
	0
	r=2
	c=2
	down=3
	right=3
	return 3			

Ex. 4.36



Ex. 4.37

depth = 5