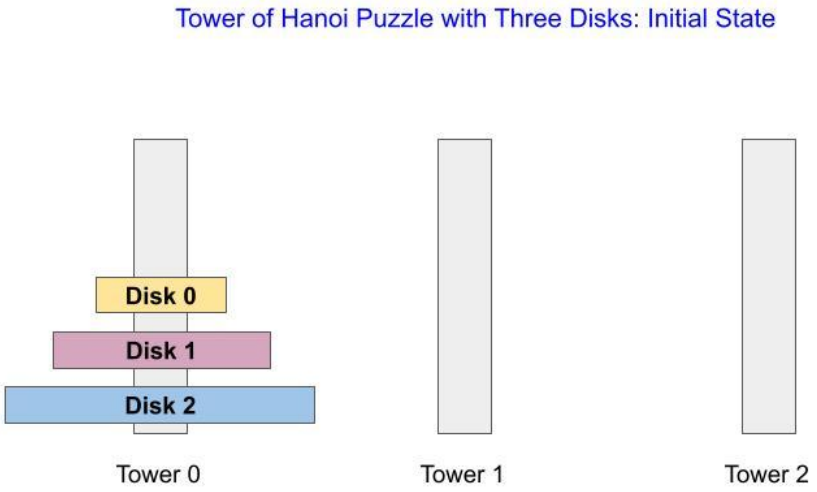
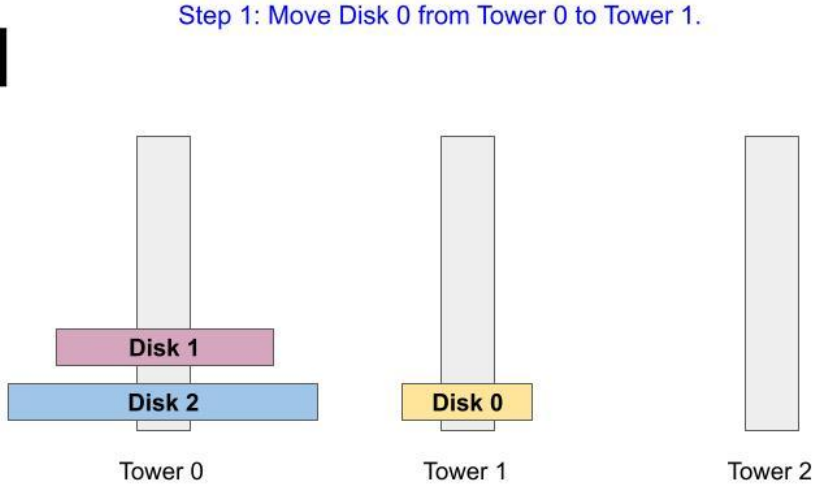
Dmitri Stanchevici
Unit 4
Module 6

Ex. 6.1

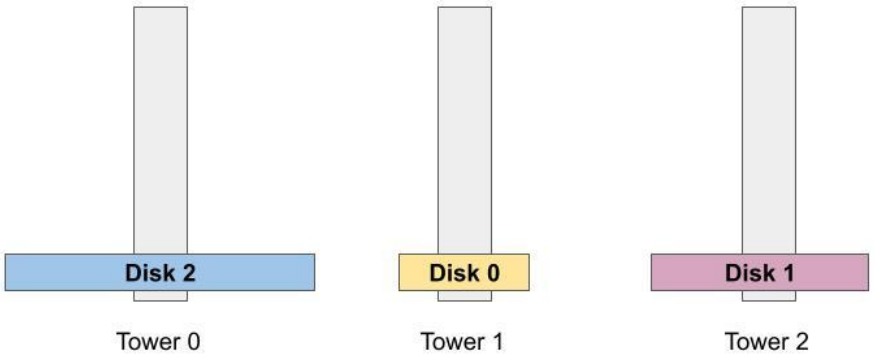


1



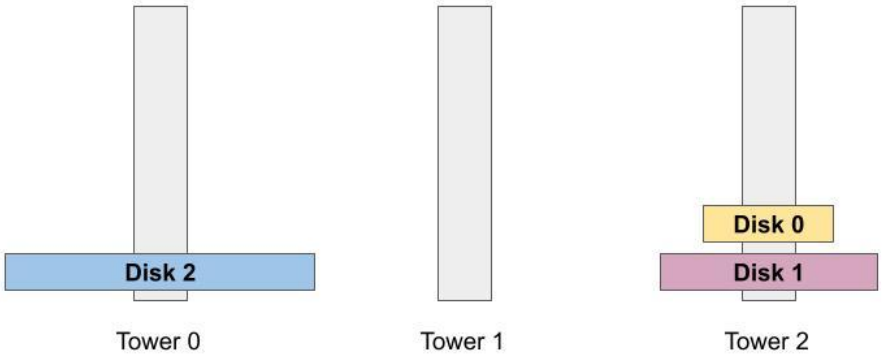
Step 2: Move Disk 1 from Tower 0 to Tower 2.

2



Step 3: Move Disk 0 from Tower 1 to Tower 2.

3

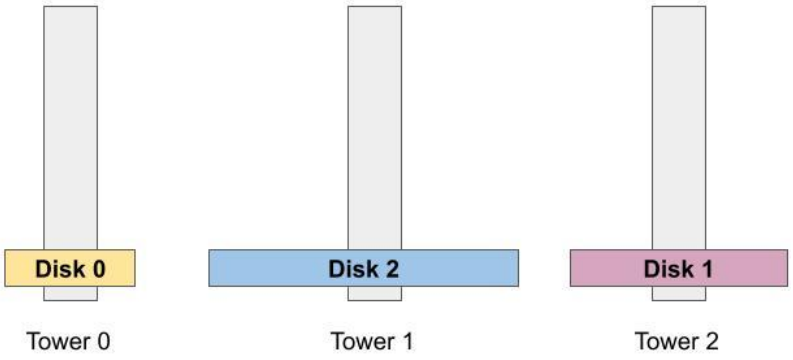


4

Step 4: Move Disk 2 from Tower 0 to Tower 1.

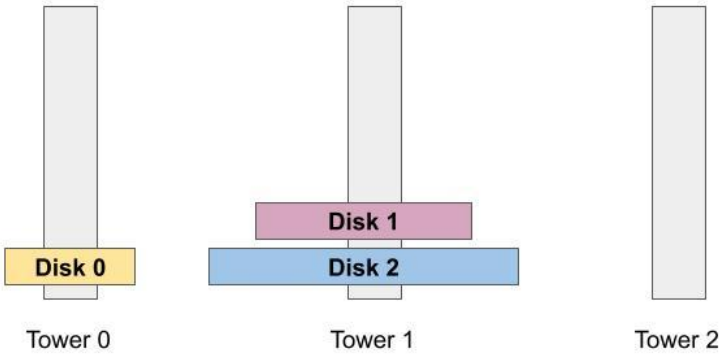
5

Step 5: Move Disk 0 from Tower 2 to Tower 0.



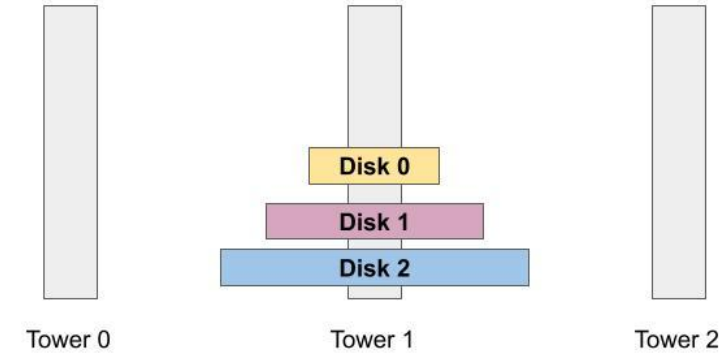
6

Step 6: Move Disk 1 from Tower 2 to Tower 1.



7

Step 7: Move Disk 0 from Tower 0 to Tower 1.



Ex. 6.2

The output corresponds to the steps I identified in Ex. 6.1.

```
=> Move disk# 0 from tower 0 to tower 1
=> Move disk# 1 from tower 0 to tower 2
=> Move disk# 0 from tower 1 to tower 2
=> Move disk# 2 from tower 0 to tower 1
=> Move disk# 0 from tower 2 to tower 0
=> Move disk# 1 from tower 2 to tower 1
=> Move disk# 0 from tower 0 to tower 1
```

Ex. 6.4

Out of moves for n from 1 to 20:

```
n=1 moves=1
n=2 moves=3
n=3 moves=7
n=4 moves=15
n=5 moves=31
n=6 moves=63
n=7 moves=127
n=8 moves=255
n=9 moves=511
n=10 moves=1023
n=11 moves=2047
n=12 moves=4095
n=13 moves=8191
n=14 moves=16383
n=15 moves=32767
n=16 moves=65535
n=17 moves=131071
n=18 moves=262143
n=19 moves=524287
n=20 moves=1048575
```

Because $moves = 2^n - 1$, this is an **exponential** algorithm.

Ex. 6.7

Recursion of **solveHanoi(intn, int i, int j)** for a ToH 3-disk puzzle from TowerOfHanoi.java in Ex. 6.2.

Step	Stack record of **solveHanoi()**			Number of stack snapshots at step
1	Call from main (n,i,j) n = 2 i = 0 j = 1 k = other(i,j) = 2 Call Step ONE (n-1=1, i=0, k=2)			1
2	Call from main (n,i,j) n = 2 i = 0 j = 1 k = other(i,j) = 2 Call Step ONE (n-1=1, i=0, k=2)	Recursion 1 (n, i, j) n = 1 i = 0 j = 2 k = other(i,j) = 1 Call Step ONE (n-1=0, i=0, k=1)		2
3	Call from main (n,i,j) n = 2 i = 0 j = 1 k = other(i,j) = 2 Call Step ONE (n-1=1, i=0, k=2)	Recursion 1 (n, i, j) n = 1 i = 0 j = 2 k = other(i,j) = 1 Call Step ONE (n-1=0, i=0, k=1)	Recursion 2 (n, i, j) n = 0 i = 0 j = 1 bottom-out (n==0) **move** (0,i=0,j=1): move Disk 0 from 0 to 1. 1 2 | 0 |	3

<table>
<tr><td></td><td></td><td></td><td><table><tr><td>0</td><td>1</td><td>2</td></tr></table>
Return</td><td></td></tr>
<tr><td>4</td><td>Call from main (n,i,j)
n = 2
i = 0
j = 1

k = other(i,j) = 2

Call Step ONE (n-1=1, i=0, k=2)</td><td>Back in Recursion 1
n = 1
i = 0
j = 2
k = other(i,j) = 1

move (1,i=0,j=2): move Disk 1 from 0 to 2.

<table><tr><td>2</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
call to Step THREE (n-1=0, k=1,j=2)</td><td></td><td>2</td></tr>
<tr><td>5</td><td>Call from main (n,i,j)
n = 2
i = 0
j = 1

k = other(i,j) = 2

Call Step ONE (n-1=1, i=0, k=2)</td><td>Recursion 1
n = 1
i = 0
j = 2
k = other(i,j) = 1

move (1,i=0,j=2): move Disk 1 from 0 to 2.

<table><tr><td>2</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
call to Step THREE (n-1=0, k=1,j=2)</td><td>Recursion 3 (n, i, j)
n=0
i=1
j=2

bottom-out (n==0)

move (0, 1,2)

<table><tr><td>2</td><td></td><td>0
1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
Return</td><td>3</td></tr>
<tr><td>6</td><td>Call from main (n,i,j)</td><td>Back in Recursion 1</td><td></td><td>2</td></tr>
</table>

<table>
<tr><td></td><td>n = 2
i = 0
j = 1

k = other(i,j) = 2

Call Step ONE (n-1=1, i=0, k=2)</td><td>Return</td><td></td><td></td></tr>
<tr><td>7</td><td>Back in call from main
n = 2
i = 0
j = 1

k = other(i,j) = 2

move (n=2, i=0, j=1): move Disk 1 from 0 to 2.

<table><tr><td></td><td>2</td><td>0
1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
call to Step THREE (n-1=1, k=2,j=1)</td><td></td><td></td><td>1</td></tr>
<tr><td>8</td><td>call from main
n = 2
i = 0
j = 1

k = other(i,j) = 2

move (n=2, i=0, j=1): move Disk 1 from 0 to 2.

<table><tr><td></td><td>2</td><td>0
1</td></tr></table></td><td>Recursion 4
n=1,
i=2
j=1

k = 0

call to Step ONE (n-1=0, i=2, k=0)</td><td></td><td>2</td></tr>
</table>

	<table><tr><td>0</td><td>1</td><td>2</td></tr></table> call to Step THREE (n-1=1, k=2,j=1)			
9	call from main n = 2 i = 0 j = 1 k = other(i,j) = 2 **move** (n=2, i=0, j=1): move Disk 1 from 0 to 2. <table><tr><td></td><td>2</td><td>0 1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table> call to Step THREE (n-1=1, k=2,j=1)	Recursion 4 n=1, i=2 j=1 k = 0 call to Step ONE (n-1=0, i=2, k=0)	Recursion 5 n=0 i=2 j=0 base (n==0) **move** (n=0, i=2, j=0): move Disk 1 from 0 to 2. <table><tr><td>0</td><td>2</td><td>1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table> return	3
10	call from main n = 2 i = 0 j = 1 k = other(i,j) = 2 **move** (n=2, i=0, j=1): move Disk 1 from 0 to 2. <table><tr><td></td><td>2</td><td>0 1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>	Back in Recursion 4 n=1, i=2 j=1 k = 0 **move** (n=1, i=2, j=1): move Disk 1 from 0 to 2. <table><tr><td>0</td><td>1 2</td><td></td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table> call to Step THREE (n-1=0,		2

<table>
<tr><td></td><td>call to Step THREE
(n-1=1, k=2,j=1)</td><td>k=0,j=1)</td><td></td><td></td></tr>
<tr><td>11</td><td>call from main
n = 2
i = 0
j = 1

k = other(i,j) = 2

move (n=2, i=0, j=1): move Disk 1 from 0 to 2.

<table><tr><td></td><td>2</td><td>0
1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
call to Step THREE
(n-1=1, k=2,j=1)</td><td>Recursion 4
n=1,
i=2
j=1

k = 0

move (n=1, i=2, j=1): move Disk 1 from 0 to 2.

<table><tr><td>0</td><td>1
2</td><td></td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
call to Step THREE (n-1=0, k=0,j=1)</td><td>Recursion 6
n=0
i=0
j=1

base (n==0)

move (0, 0, 1)

<table><tr><td></td><td>0
1
2</td><td></td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
Return</td><td>3</td></tr>
<tr><td>12</td><td>call from main
n = 2
i = 0
j = 1

k = other(i,j) = 2

move (n=2, i=0, j=1): move Disk 1 from 0 to 2.

<table><tr><td></td><td>2</td><td>0
1</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table>
call to Step THREE
(n-1=1, k=2,j=1)</td><td>Back in Recursion 4

Return</td><td></td><td>2</td></tr>
</table>

13	Back in call from main Return			1

Ex. 6.8

TowerOfHanoi.java can be either a tail recursion or a recursion with backtracking.

It can be a **tail recursion** because the placements of the disks on the three poles are carried over from recursion to recursion. In this sense, changes are not undone, a behavior of a tail recursion.

At the same time, TowerOfHanoi.java can be viewed as a **recursion with backtracking** because the recursive calls continually change the order of **i** and **j** in **solveHanoi (n, i, j)** by substituting **k** (with a continually changing value) for one of them. This corresponds to module 6's definition of a recursion with backtracking as "Recursion where you need to un-do changes so that you can properly explore all possibilities."

Ex. 6.10

MazeMaker.java does not produce a path of the desired length (5x5=25) because in most cases before this length is reached, the path gets to a cell that does not have neighbors that (1) are closed off from this cell and that (2) have not been visited yet. Both conditions have to be satisfied for cells to be added to the **Coord[] validNeighbors** array. If not cells are added (because none of them satisfy the two conditions), the program ends up with **(validNeighbors == null) || (validNeighbors.length == 0),** a condition that breaks out of the recursion and returns false to **generateMaze()**. I see two patterns that may satisfy these two conditions for a desired path of 25. However, a random selection for either of these two paths, while not impossible, seems highly improbable.

This is an example of a **tail recursion** because the changes made to maze through maze.breakWall() and maze.markVisited() are kept from recursion to recursion. This behavior corresponds to module 6's definition of a tail recursion as "Recursion where you don't un-do changes."

Ex. 6.11

We rearrange the array of neighbors randomly in order to ensure a variety of possible openings from each cell. Thus, due to this randomization, the cell may open to its neighbor to the north, or to the south, or east, or west.

This program produces a path covering all 25 cells.

Ex. 6.14

I have conducted five (5) experiments, executing the program 1,000 times in each experiment. Below are the numbers of found paths in each experiment:

1. 569
2. 546
3. 563
4. 521
5. 538

Thus, on average, the program finds the path from (0,0) to (4,4) in a little more than 50% of cases.

The code does not always find the solution because it follows a predefined path as specified by **validNeighbors[0]**. Having this as the next coordinate/cell may or may not lead to the end position (4,4).

We mark the cell as visited to keep track of the path. Not marking cells as visited would enable the path to double back. I don't know why we don't allow it here; it would seem that allowing the path to go back may solve the problem of finding a solution leading to the end.

Ex. 6.15

Yes, the code in MazeMaker5.java always finds the solution. I repeated the experiment reported in the last exercises, and the results confirmed that the solution was found in 100% of cases.

1. Why is the start cell added at the very end? -- The population of the solutionPath begins with the last cell (4,4). After this cell has been added, the cell that led into it is added in the front of the list. Next is added the element that led into the last-but-one element. Thus elements are **added in reverse order,** as the recursions cascade back. Because elements are added to the front of the list, the start has to be added at the very end, when all the other elements have already been added.
2. How many dead-end's are reached in trying to find a path? Add code to count such dead-end's. -- In 1000 experiments, there were 6 dead ends on average.
3. What does a trace look like? Add copious println's to see how the recursion works. -- See below

```
dmitristanchevici@Dmitris-iMac module6 % java MazeMaker5
------------------------------
```

```
Recursion #=0
c=[0,0]
validNeighbors: [1,0]
Entered for-loop. i=0
------------------------------
Recursion #=1
c=[1,0]
validNeighbors: [2,0] [1,1]
Entered for-loop. i=0
------------------------------
Recursion #=2
c=[2,0]
validNeighbors: [2,1]
Entered for-loop. i=0
------------------------------
Recursion #=3
c=[2,1]
validNeighbors: [3,1] [2,2]
Entered for-loop. i=0
------------------------------
Recursion #=4
c=[3,1]
validNeighbors: [4,1] [3,2] [3,0]
Entered for-loop. i=0
------------------------------
Recursion #=5
c=[4,1]
validNeighbors: [4,2] [4,0]
Entered for-loop. i=0
------------------------------
Recursion #=6
c=[4,2]
validNeighbors: [3,2] [4,3]
Entered for-loop. i=0
------------------------------
Recursion #=7
c=[3,2]
validNeighbors: [3,3]
Entered for-loop. i=0
------------------------------
Recursion #=8
c=[3,3]
validNeighbors: [2,3] [4,3] [3,4]
Entered for-loop. i=0
```

```
------------------------------
Recursion #=9
c=[2,3]
validNeighbors: [2,4] [2,2]
Entered for-loop. i=0
------------------------------
Recursion #=10
c=[2,4]
validNeighbors: [1,4] [3,4]
Entered for-loop. i=0
------------------------------
Recursion #=11
c=[1,4]
validNeighbors: [0,4] [1,3]
Entered for-loop. i=0
------------------------------
Recursion #=12
c=[0,4]
validNeighbors == nul: true
[0,4] did not work. Make it Unvisited and move to next
Entered for-loop. i=1
------------------------------
Recursion #=13
c=[1,3]
validNeighbors: [0,3] [1,2]
Entered for-loop. i=0
------------------------------
Recursion #=14
c=[0,3]
validNeighbors: [0,2]
Entered for-loop. i=0
------------------------------
Recursion #=15
c=[0,2]
validNeighbors: [0,1]
Entered for-loop. i=0
------------------------------
Recursion #=16
c=[0,1]
validNeighbors: [1,1]
Entered for-loop. i=0
------------------------------
Recursion #=17
c=[1,1]
```

validNeighbors: [1,2]
Entered for-loop. i=0

Recursion #=18
c=[1,2]
validNeighbors: [2,2]
Entered for-loop. i=0

Recursion #=19
c=[2,2]
validNeighbors == nul: true
[2,2] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[1,2] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[1,1] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[0,1] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[0,2] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[0,3] did not work. Make it Unvisited and move to next
Entered for-loop. i=1

Recursion #=20
c=[1,2]
validNeighbors: [2,2] [1,1]
Entered for-loop. i=0

Recursion #=21
c=[2,2]
validNeighbors == nul: true
[2,2] did not work. Make it Unvisited and move to next
Entered for-loop. i=1

Recursion #=22
c=[1,1]
validNeighbors: [0,1]
Entered for-loop. i=0

Recursion #=23
c=[0,1]
validNeighbors: [0,2]
Entered for-loop. i=0

```
------------------------------
Recursion #=24
c=[0,2]
validNeighbors: [0,3]
Entered for-loop. i=0
------------------------------
Recursion #=25
c=[0,3]
validNeighbors == nul: true
[0,3] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[0,2] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[0,1] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[1,1] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[1,2] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[1,3] did not work. Make it Unvisited and move to next
After the for-loop, none of the neighbors worked. Return false.
[1,4] did not work. Make it Unvisited and move to next
Entered for-loop. i=1
------------------------------
Recursion #=26
c=[3,4]
validNeighbors: [4,4]
Entered for-loop. i=0
------------------------------
Recursion #=27
c=[4,4]
(c.row == end.row): true(c.col == end.col): true
After the recursive call(s) inside iteration 0 [4,4] is ready to be added. Return true.
After the recursive call(s) inside iteration 1 [3,4] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [2,4] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [2,3] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [3,3] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [3,2] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [4,2] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [4,1] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [3,1] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [2,1] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [2,0] is ready to be added. Return true.
After the recursive call(s) inside iteration 0 [1,0] is ready to be added. Return true.
```

1: Move from [0,0] to [1,0]
2: Move from [1,0] to [2,0]
3: Move from [2,0] to [2,1]
4: Move from [2,1] to [3,1]
5: Move from [3,1] to [4,1]
6: Move from [4,1] to [4,2]
7: Move from [4,2] to [3,2]
8: Move from [3,2] to [3,3]
9: Move from [3,3] to [2,3]
10: Move from [2,3] to [2,4]
11: Move from [2,4] to [3,4]
12: Move from [3,4] to [4,4]

Ex. 6.16

1. Can 3 queens be placed on a 3X3 board? -- No solution for 3 on a 3 x 3 board
2. Can 3 queens be placed on a 4X4 board? -- Yes
   Solution found for 3 on a 4 x 4 board

Chessboard:
X O O O
O O X O
O O O O
O X O O

Ex. 6.17

Partitioning with 5 as the partitioning element.

data

5	7	8	6	2	1	9	3	4
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

left=0
right=8
partitioningElement=5
LC = left + 1 = 1
RC = right = 8

In **while(true)**

Iteration 1

5	4	8	6	2	1	9	3	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

LC = 2
RC = 7

Iteration 2

5	4	3	6	2	1	9	8	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

LC = 3
RC = 6

Iteration 3

RC = 5

5	4	3	1	2	6	9	8	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

LC = 4
RC = 4

Iteration 4

break out of loop

Outside **while (true)** loop:

2	4	3	1	5	6	9	8	7
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*