CS106 Data Structure & Java Fundamentals

BitTiger丨来自硅谷的终身学习平台

Claire & Bob

BITTIGER

版权声明

Copyright Policy

Feedback

- Refer & Resume/CV
- Requirements for testing positions - https://www.glassdoor.com/Interview/Google-Software-Engineer-In-Test-Interview-Questions-EI_IE9079.0,6_KO7,32.htm

Feedback

- Refer & Resume/CV
- Requirements for testing positions - https://www.glassdoor.com/Interview/Google-Software-Engineer-In-Test-Interview-Questions-EI_IE9079.0,6_KO7,32.htm
- Occupation? (Student - 1, Employee - 2, Others - 3)

Feedback

- Refer & Resume/CV
- Requirements for testing positions - https://www.glassdoor.com/Interview/Google-Software-Engineer-In-Test-Interview-Questions-EI_IE9079.0,6_KO7,32.htm
- Occupation? (Student - 1, Employee - 2, Others - 3)
- Major? (CS/EE - 4, Others -5)

Feedback

- Refer & Resume/CV
- Requirements for testing positions - https://www.glassdoor.com/Interview/Google-Software-Engineer-In-Test-Interview-Questions-EI_IE9079.0,6_KO7,32.htm
- Occupation? (Student - 1, Employee - 2, Others - 3)
- Major? (CS/EE - 4, Others -5)
- Goal? (Job interview - 6, Interest - 7, Others - 8)

Feedback

- Refer & Resume/CV
- Requirements for testing positions - https://www.glassdoor.com/Interview/Google-Software-Engineer-In-Test-Interview-Questions-EI_IE9079.0,6_KO7,32.htm
- Occupation? (Student - 1, Employee - 2, Others - 3)
- Major? (CS/EE - 4, Others -5)
- Goal? (Job interview - 6, Interest - 7, Others - 8)
- Thanks for your thank-you letter ;)

Algorithms

- Binary Search
- Sort (quickSort/mergeSort)
- Dynamic Programming
- DFS & BFS

Algorithms

- Binary Search
- Sort (quickSort/mergeSort)
- Dynamic Programming
- DFS & BFS

Quick Sort

In short:

1)Randomly pick up a PIVOT

2)Make sure all the elements smaller than the pivot come before the elements larger than the pivot (that will naturally create two divisions)

3)Repeated the process on the two portions created

Quick Sort

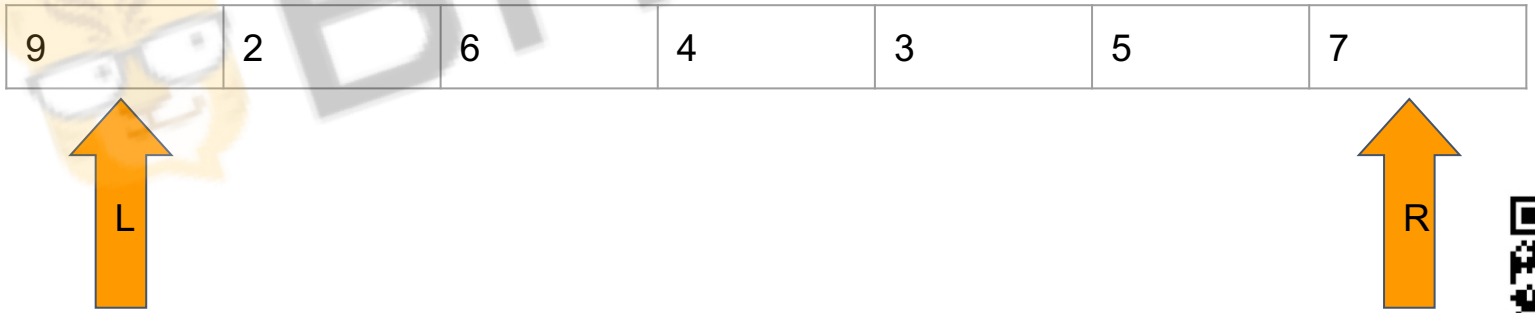
Pivot: A **random** element in the array
In this example, we use middle index for simplicity -
pivot: 4

9	2	6	4	3	5	7

Quick Sort

pivot: 4

Left & right pointer

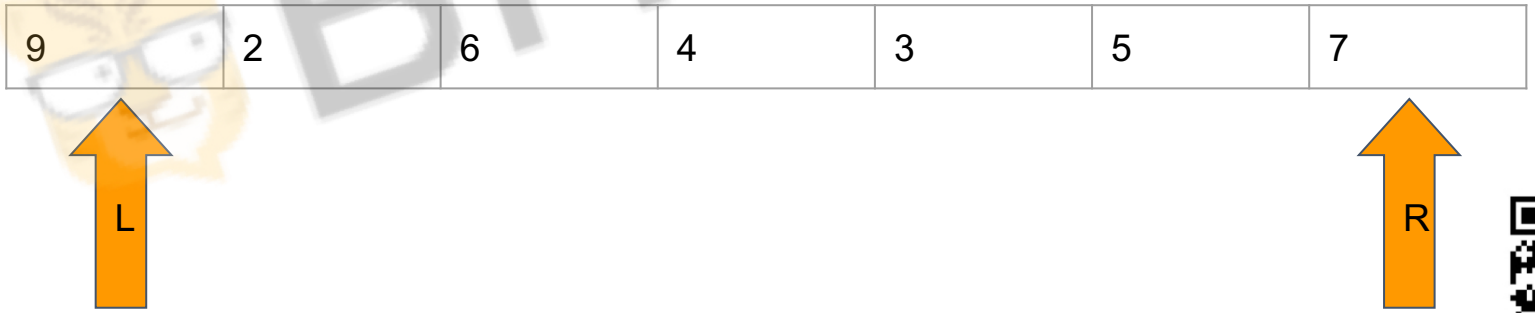


Quick Sort

pivot: 4

Left & right pointer:

- Move left pointer to the right until find one element>= 4

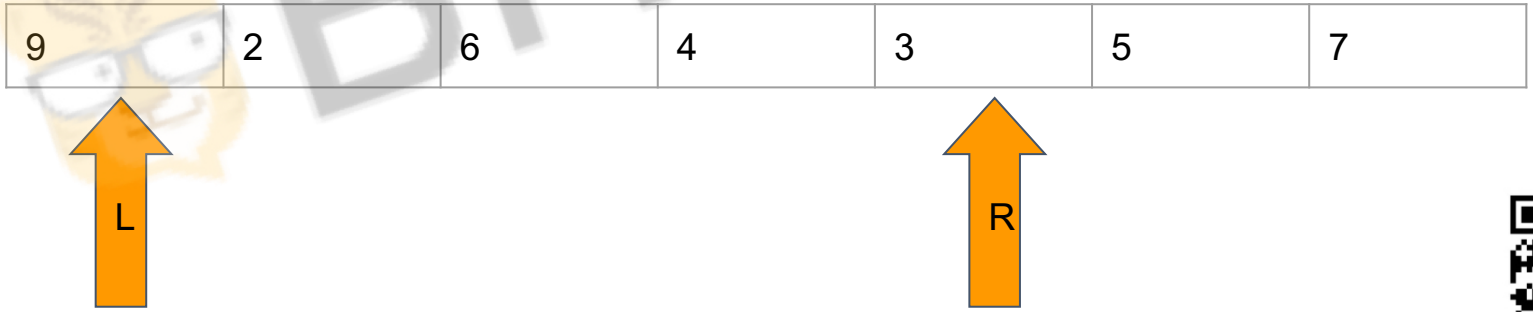


Quick Sort

pivot: 4

Left & right pointer:

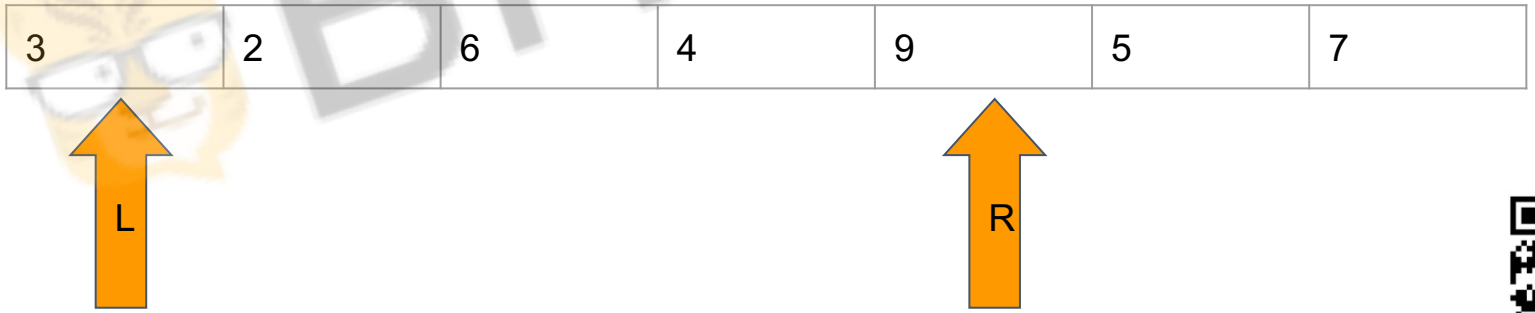
- Move left pointer to the right until finding one element>=4
- Move right pointer to the left until finding one element<=4



Quick Sort

pivot: 4

Out of order - Swap!

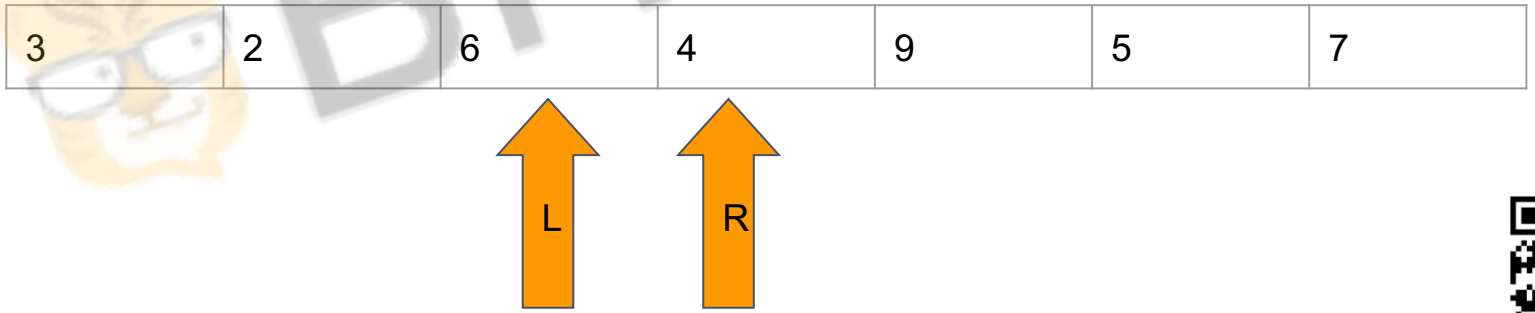


Quick Sort

pivot: 4

After swap, left++ & right--;

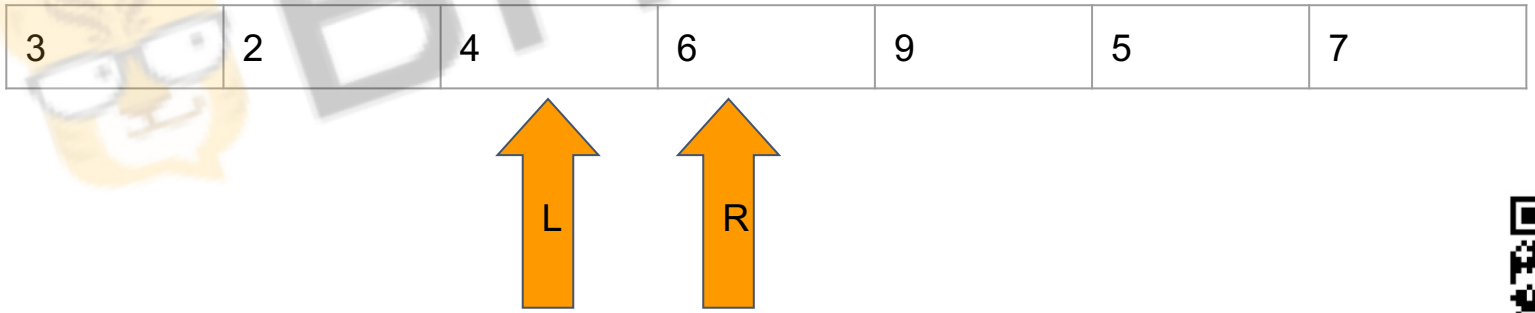
Find the next left and right elements out of order



Quick Sort

pivot: 4

Find the next left and right elements out of order out of order - Swap!

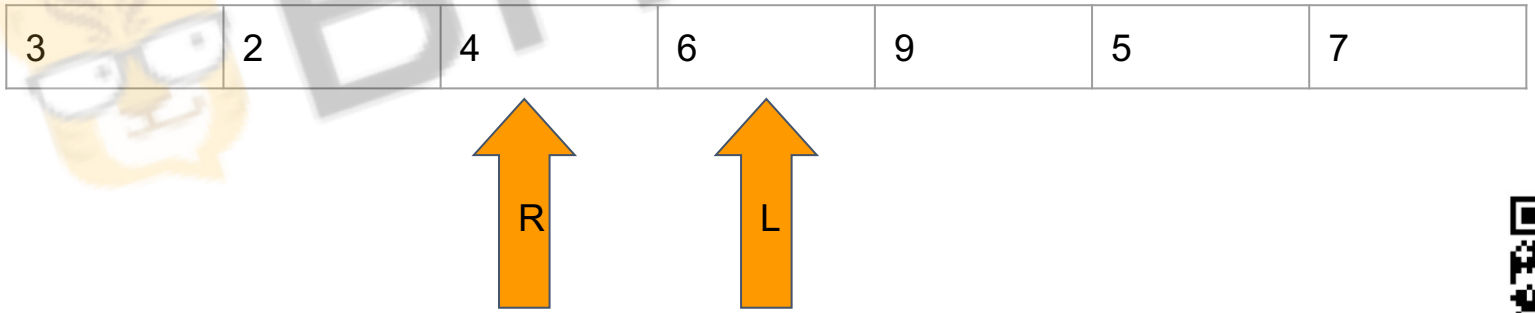


Quick Sort

pivot: 4

After swap, left++ & right--;

Note that right < left → stop the loop!

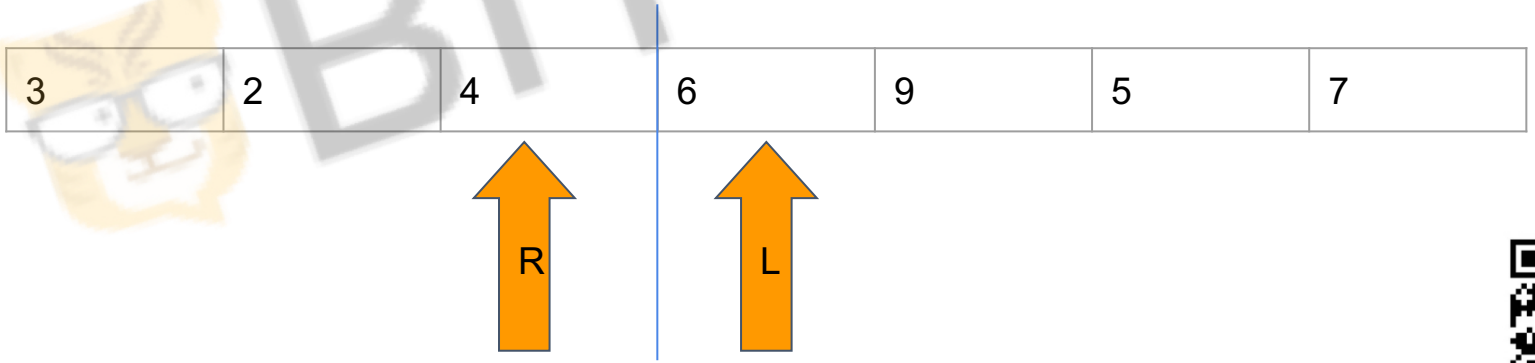


Quick Sort

pivot: 4

Note that right < left → stop the loop!

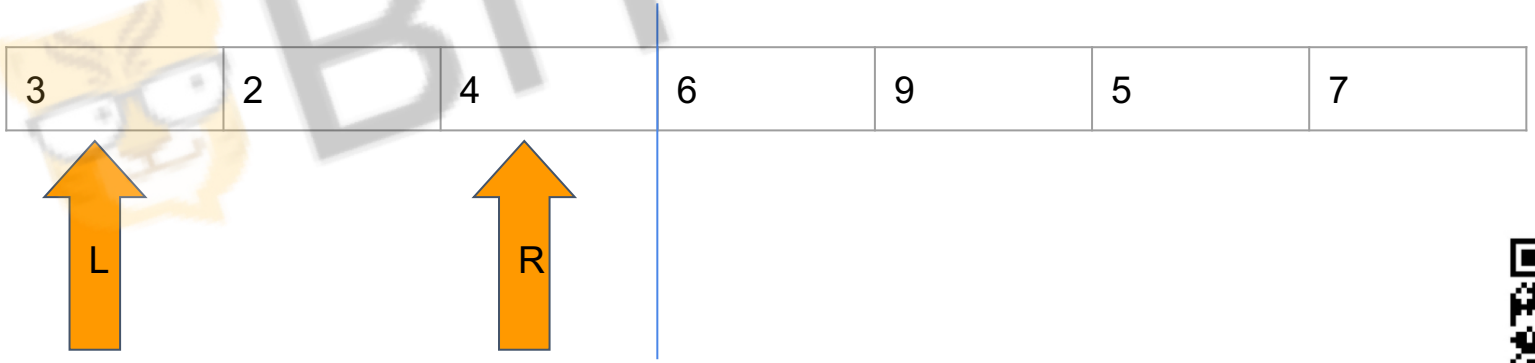
The array is divided naturally by [0, L-1] and [L, len-1]



Quick Sort

Repeat the process on two portions

Pivot: 2

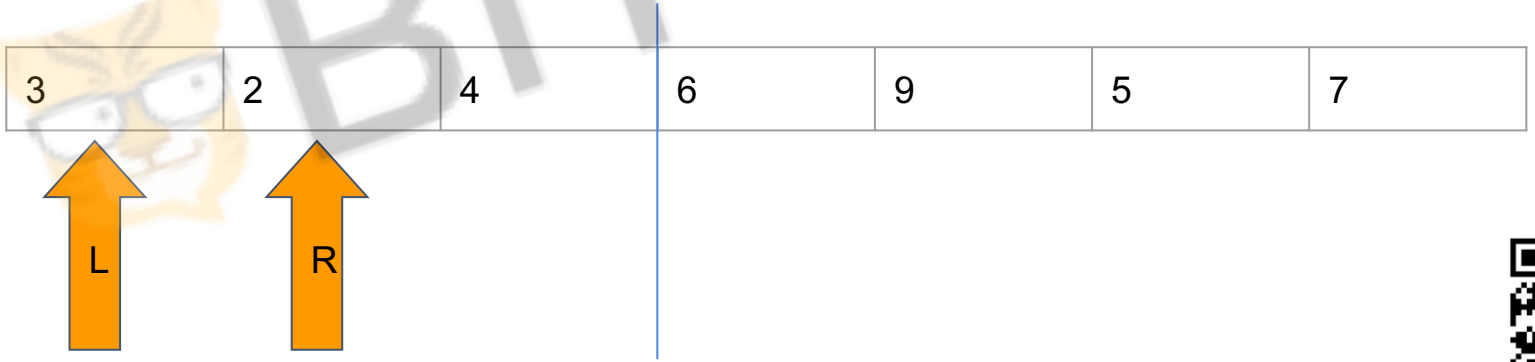


Quick Sort

Pivot: 2

Move left pointer to the right until finding an element >= 2

Move right pointer to the left until finding an element <= 2



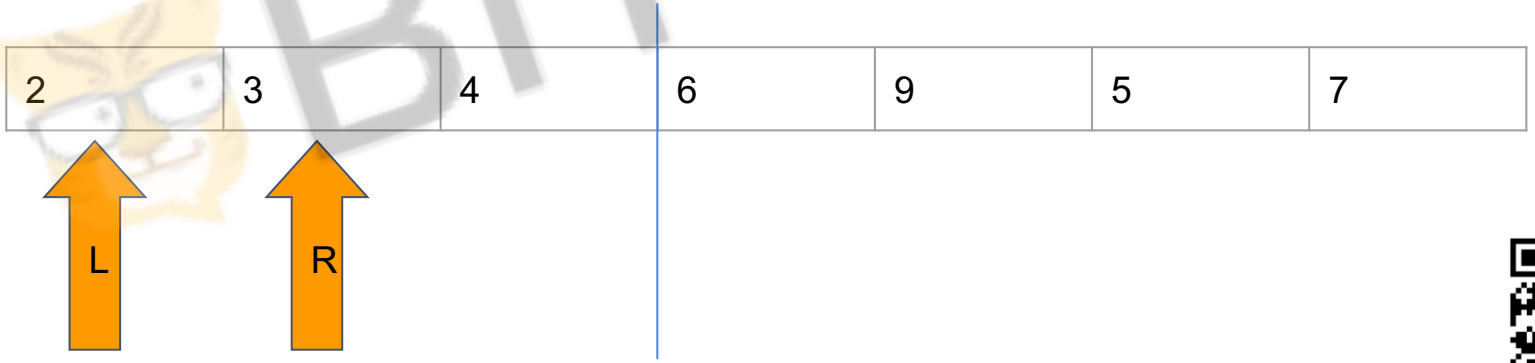
Quick Sort

Pivot: 2

Move left pointer to the right until finding an element >= 2

Move right pointer to the left until finding an element <= 2

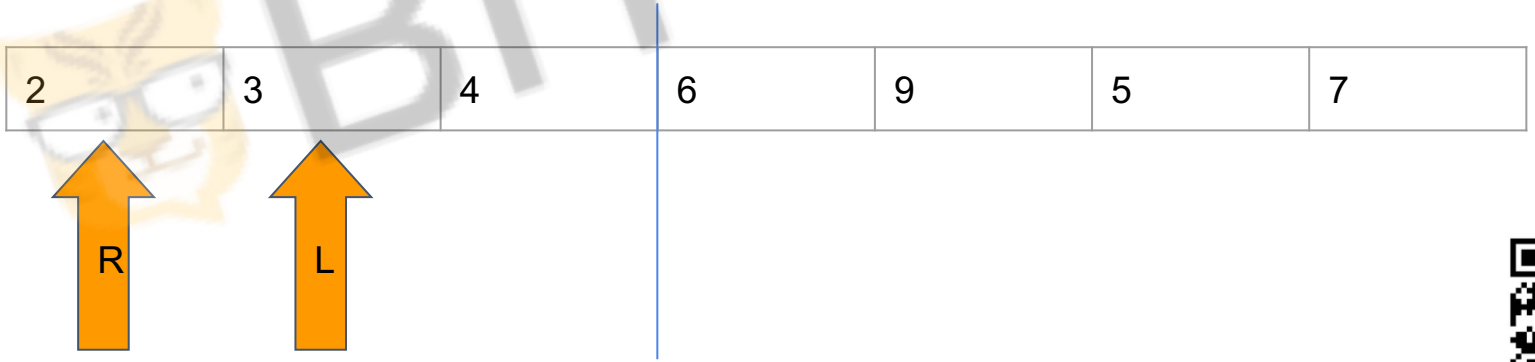
Swap!



Quick Sort

Pivot: 2

Left++ & right--

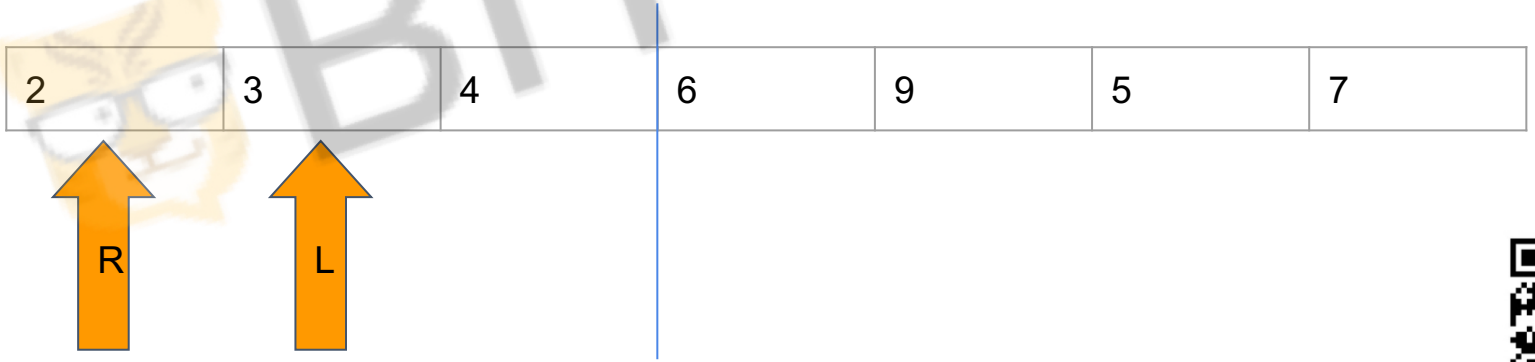


Quick Sort

Pivot: 2

Left++ & right--;

Note that right < left → Stop the loop

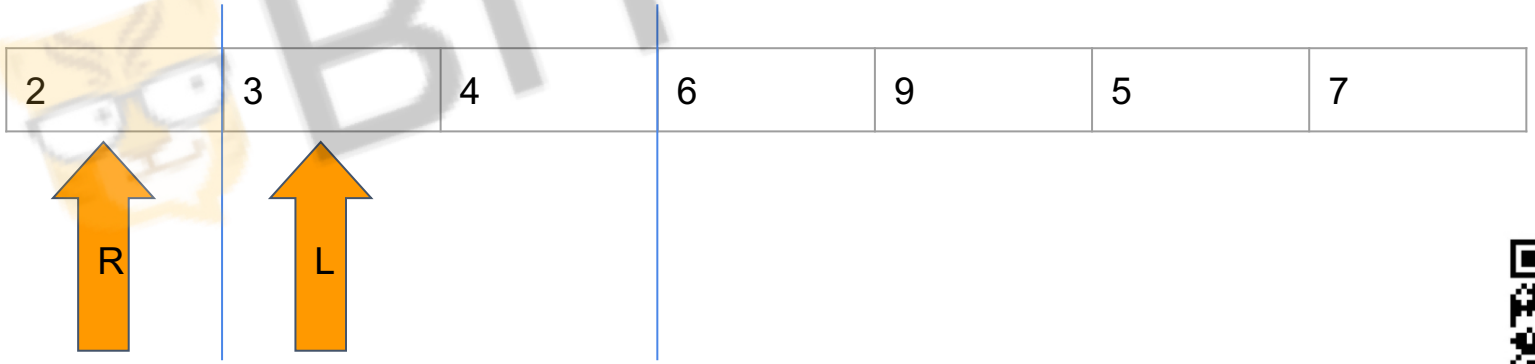


Quick Sort

Pivot: 2

Note that right < left → Stop the loop

Further divide into two parts by L-1

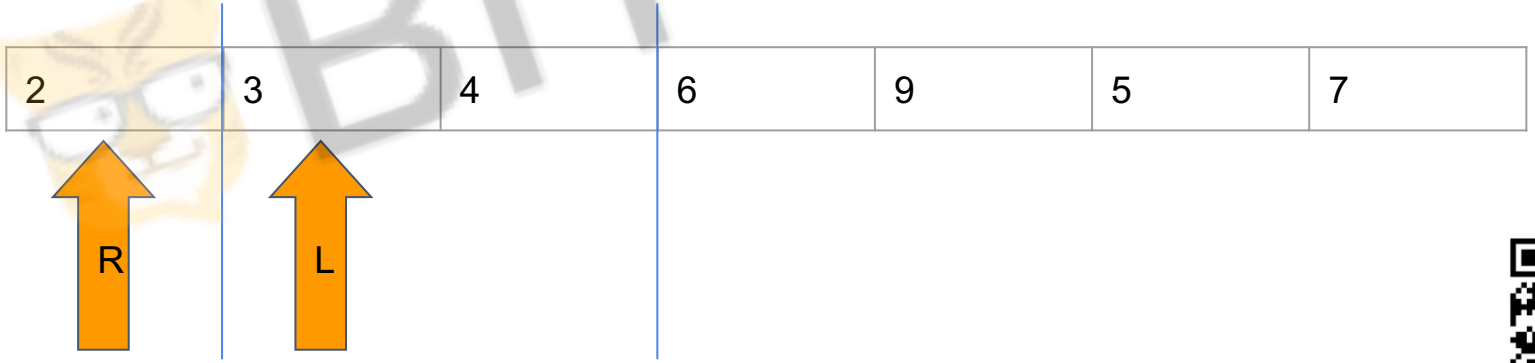


Quick Sort

Pivot: 2

Further divide into two parts by L-1

Apply the process to every parts...



Time complexicity

Average: O(nLogN)
Worst: O(N^2)

Learning by Doing

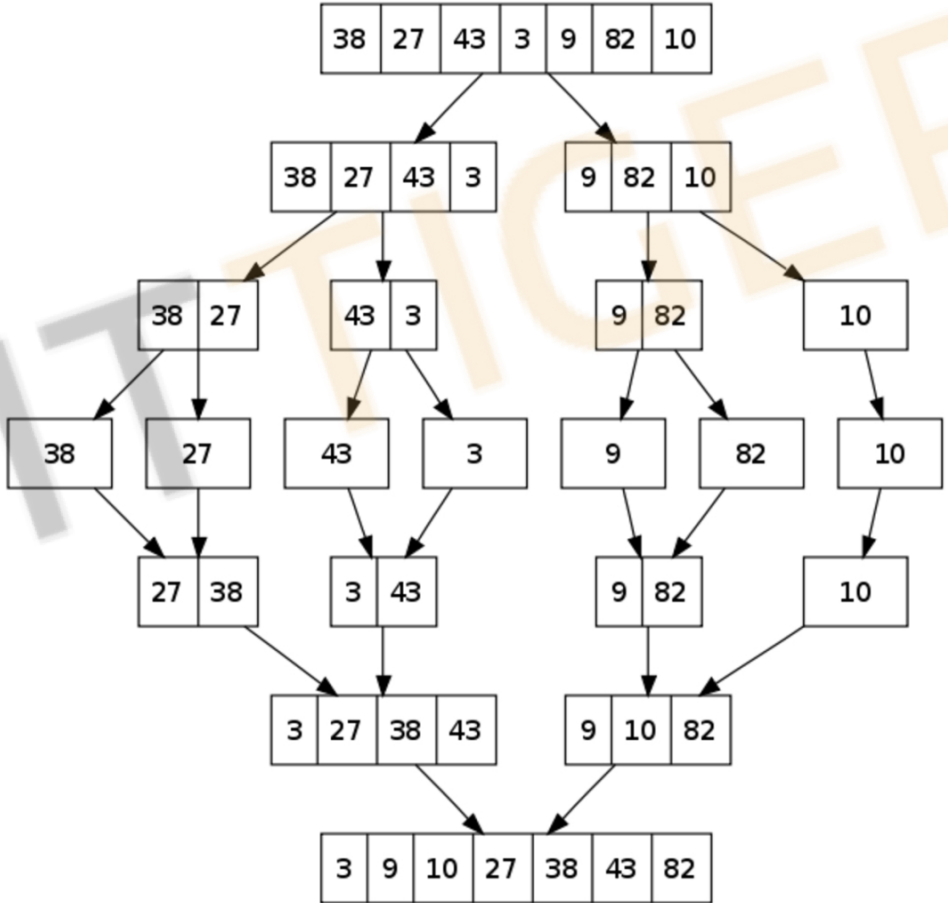
Let’s implement quickSort algorithm

Merge Sort

In short:
MergeSort is a Divide and Conquer Algorithm.
It divides input array in two halves,
Calls itself for the two halves,
And then merges the two sorted halves.

Merge Sort

(1)Divide
(2)Merge(Sort)



Merge Sort

Pros: Always O(nLogn)
Cons:

- Extra space when doing merge.
- Need to copy all the elements into a new array, merge tham and copy them back

Learning by Doing

Let’s implement mergeSort algorithm

Algorithms

- Binary Search
- Sort (quickSort/mergeSort)
- Dynamic Programming
- DFS & BFS

Fibonacci Problem

Let’s think about Fibonacci problem…
fib(n) = fib(n-1) + fib(n-2)
fib(0) = 0
fib(1) = 1

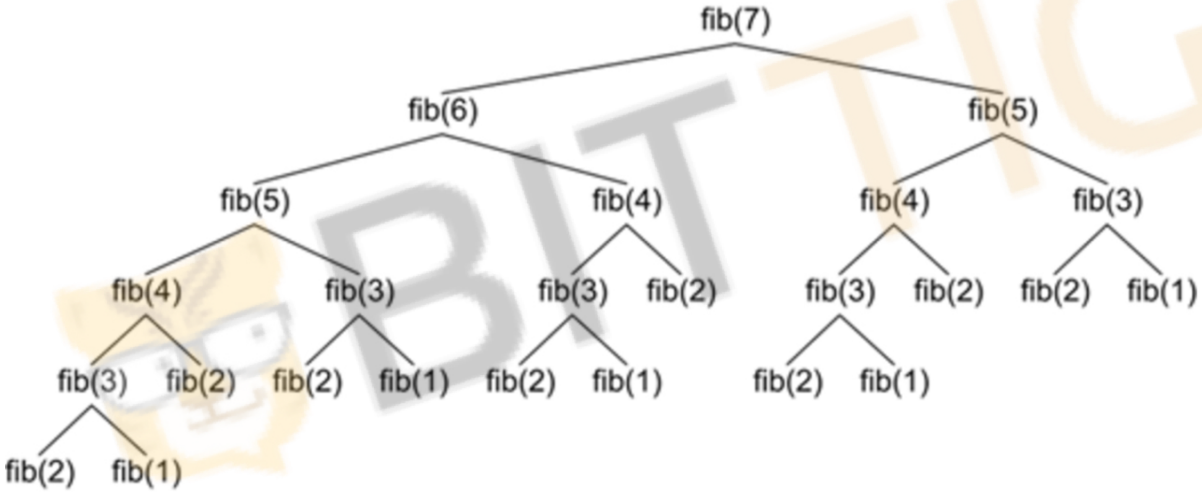
Fibonacci Problem - Recursive

Solve Fib in recursive approach:

```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

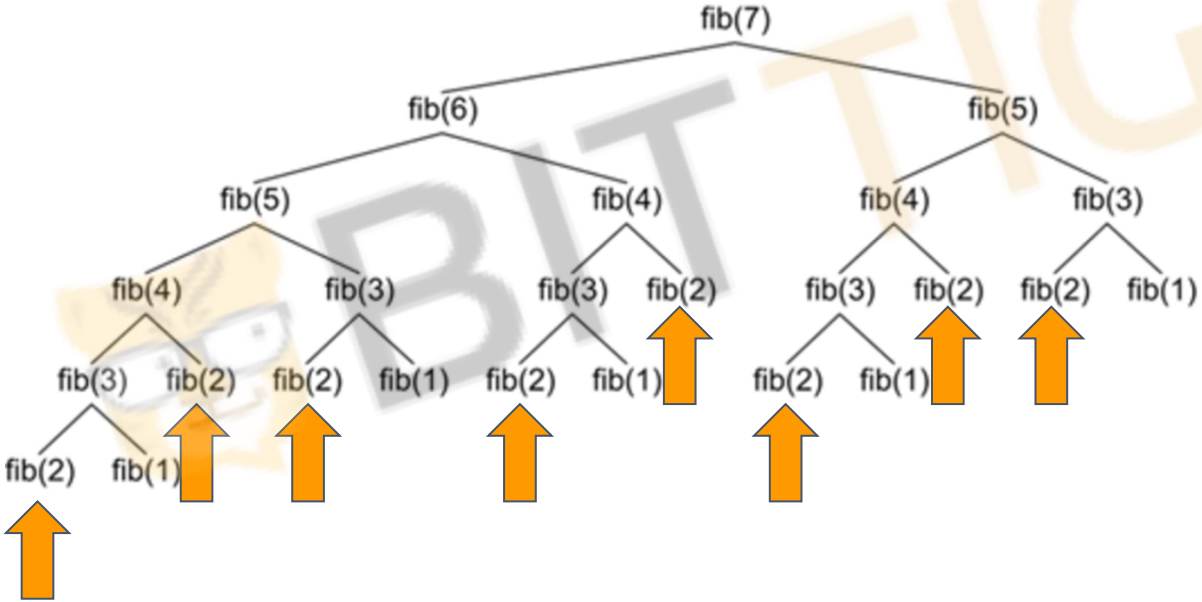
Fibonacci Problem

```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```



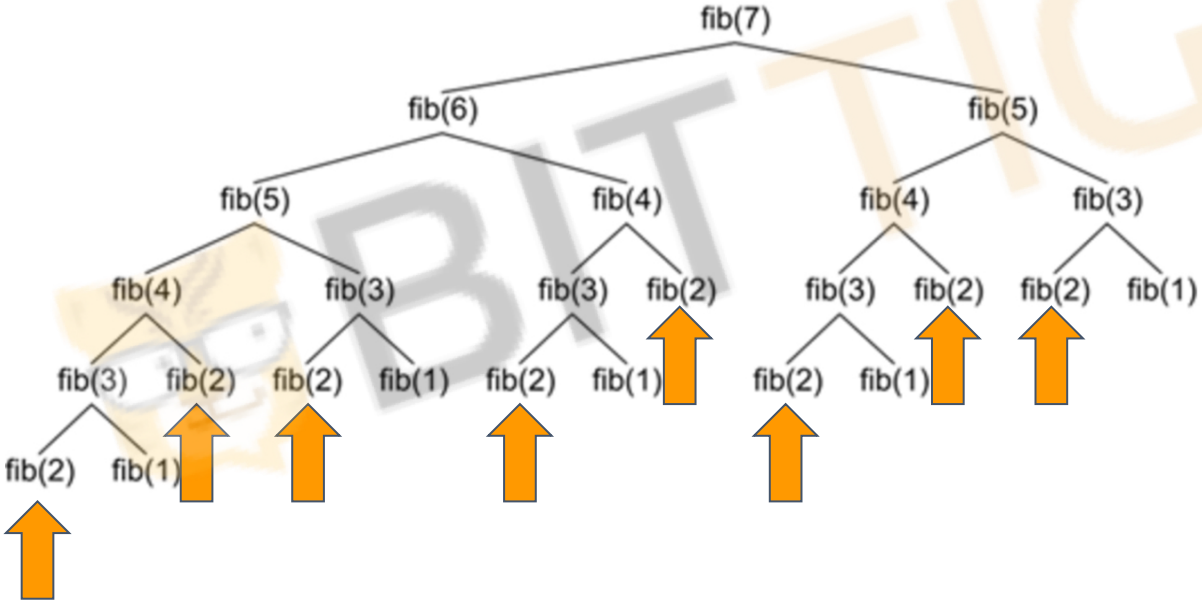
Fibonacci Problem

```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```



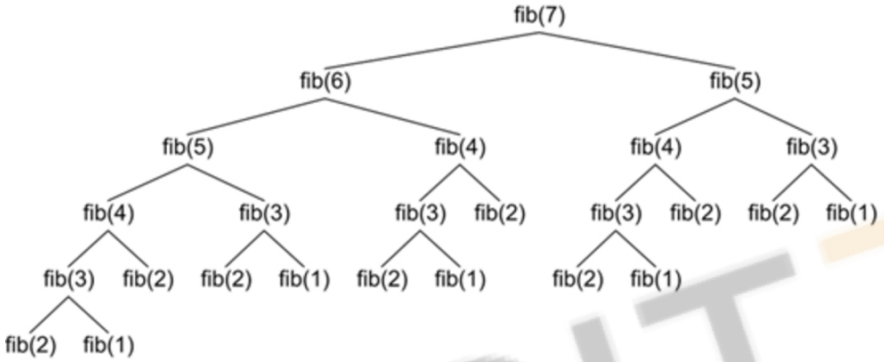
Fibonacci Problem

```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```



Time:O(2^N)!

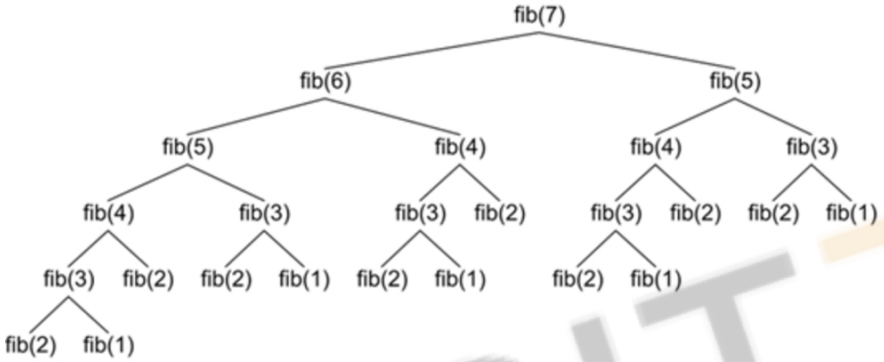
Memoization




```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

```
int fib(int n, int[] mem) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if (exist(mem, n)){
        return mem[n];
    }
    return fib(n-1, mem) + fib(n-2, mem);
}
```

Memoization

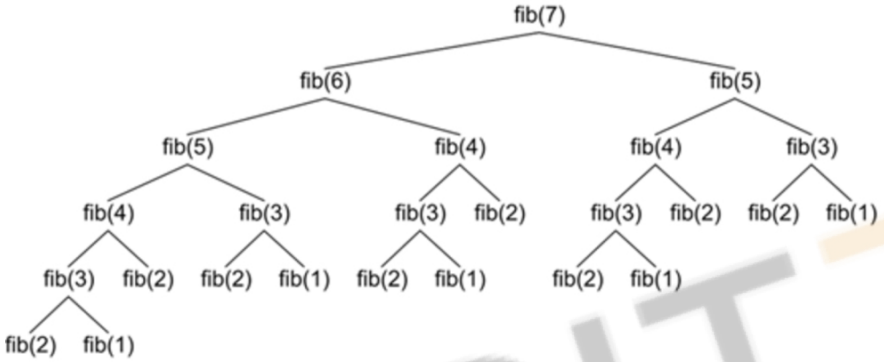



```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Memoization!
No duplicate effort!

```
int fib(int n, int[] mem) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if (exist(mem, n)){
        return mem[n];
    }
    return fib(n-1, mem) + fib(n-2, mem);
}
```

Memoization




```
int fib(int n) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

Time: O(n)

```
int fib(int n, int[] mem) {
    if (n <= 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if (exist(mem, n)){
        return mem[n];
    }
    return fib(n-1, mem) + fib(n-2, mem);
}
```

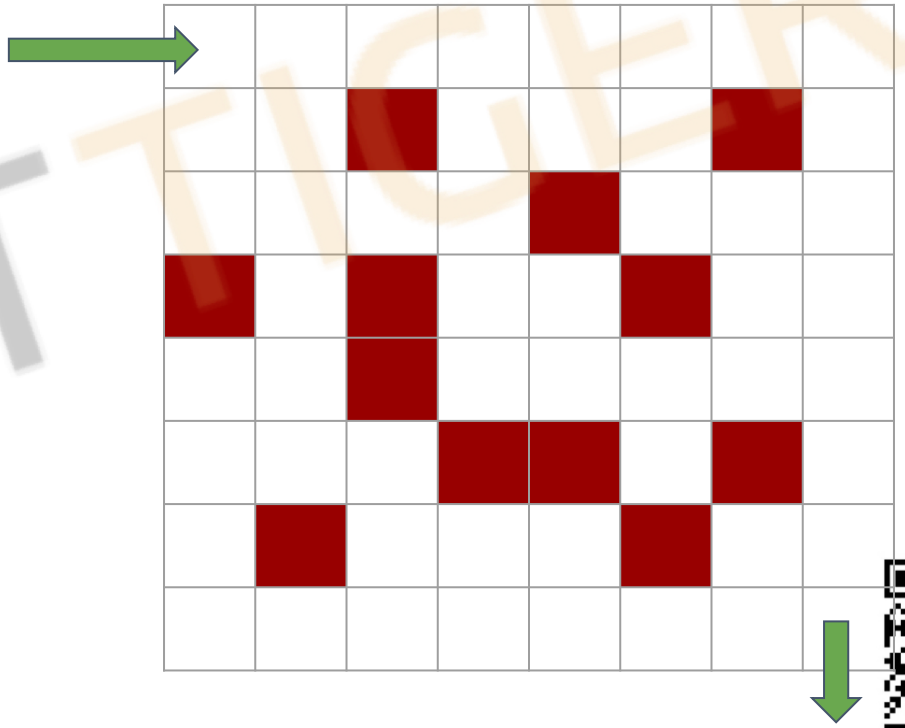
Difference between Memoization and DP

- Memoization is a term describing an optimization technique where you cache previously computed results (hashmap, array etc), and return the cached result when the same computation is needed again.
- DP is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap.
- DP is typically implemented using tabulation or memoization.

Tabulation v.s. Memoization

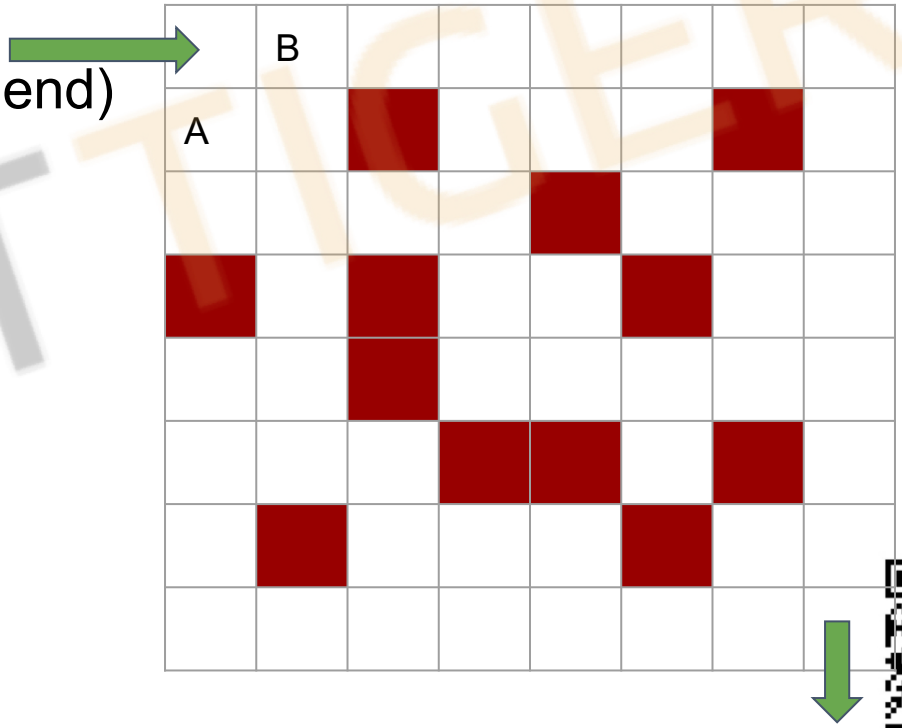
Memoization: Top down - solve the problem by maintaining a map of already solved sub problems.
Tabulation: Bottom up - solve all related sub-problems first, typically by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Evil Islands Problem



Evil Islands Problem

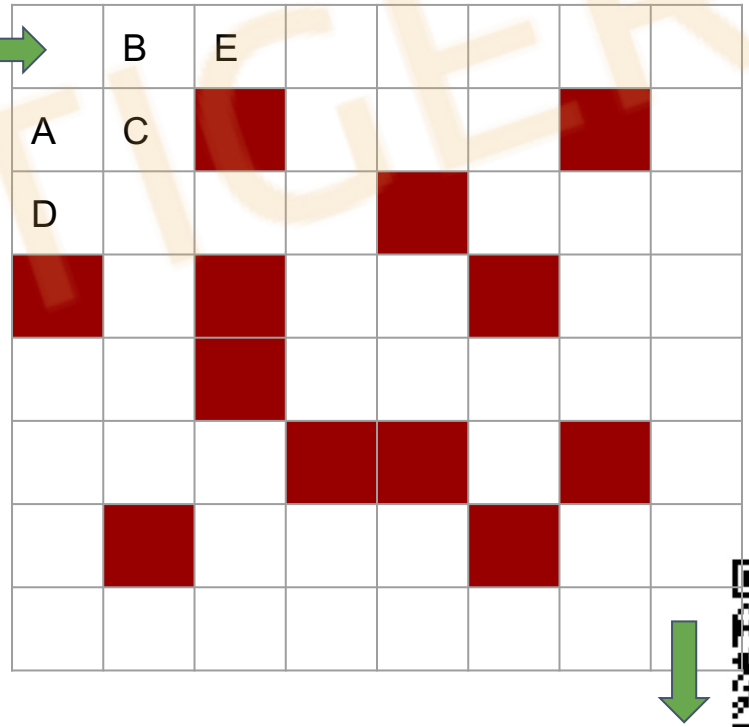
paths(start, end) =
paths(A, end) + paths(B, end)



Evil Islands Problem

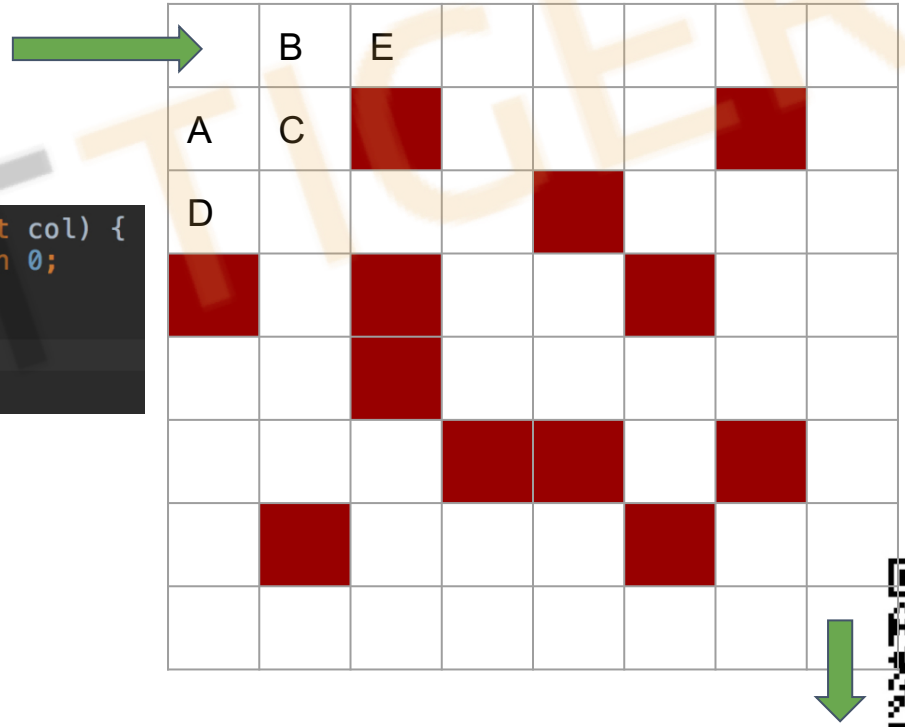
paths(A, end) =
paths(D, end) + paths(C, end)

paths(B, end) =
paths(E, end) + paths(C, end)



Evil Islands Problem

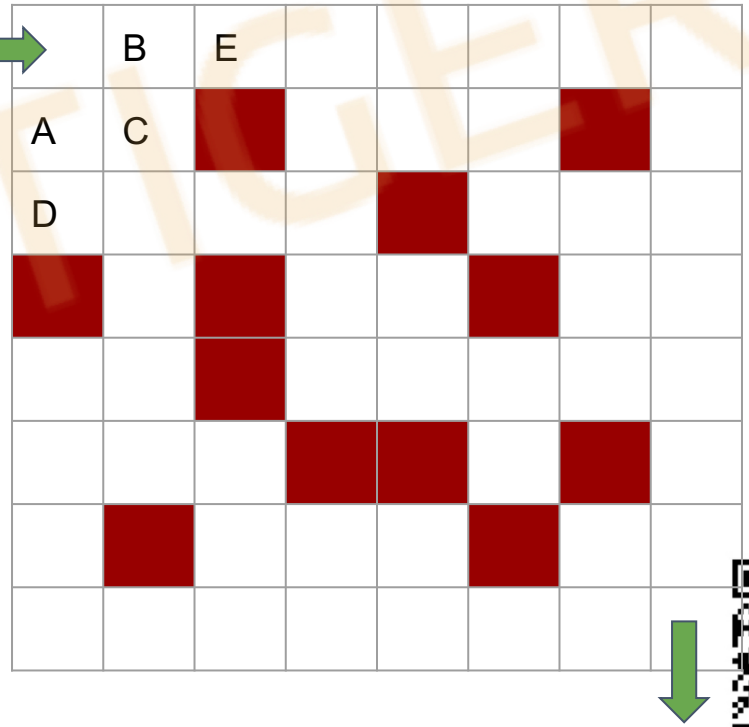
```
int countPaths(boolean[][] grid, int row, int col) {
    if (!validSquiare(grid, row, col)) return 0;
    if(isAtEnd(grid, row, col)) return 1;
    return countPaths(grid, row+1, col) +
           countPaths(grid, row, col+1);
}
```



Evil Islands Problem - Memoization

paths(A, end) =
paths(D, end) + paths(C, end)

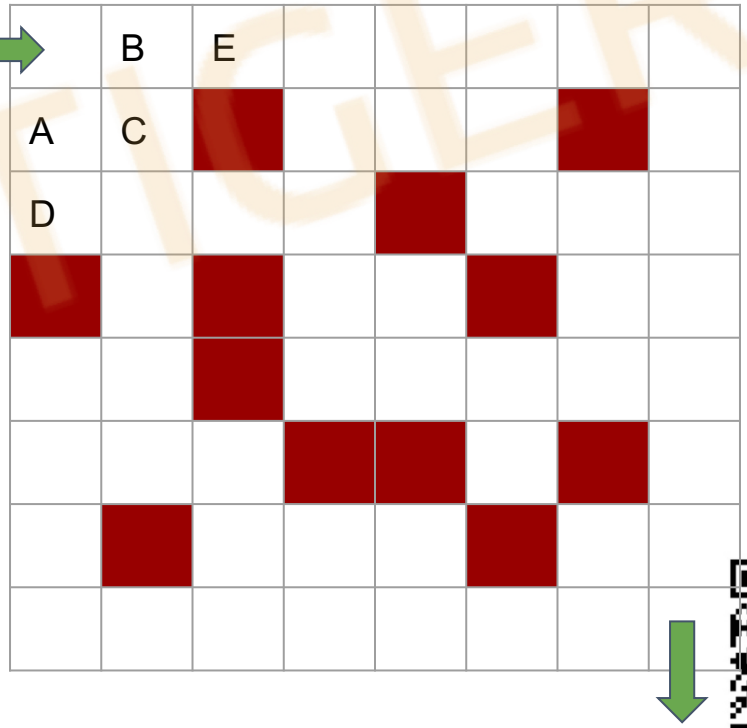
paths(B, end) =
paths(E, end) + paths(C, end)



Evil Islands Problem - Memoization

```
int countPaths(boolean[][] grid, int row, int col) {
    if (!validSquiare(grid, row, col)) return 0;
    if(isAtEnd(grid, row, col)) return 1;
    return countPaths(grid, row+1, col) +
           countPaths(grid, row, col+1);
}
```

```
int countPaths(boolean[][] grid, int row, int col, int[][] paths) {
    if (!validSquiare(grid, row, col)) return 0;
    if(isAtEnd(grid, row, col)) return 1;
    if (paths[row][col] == 0) {
        paths[row][col] = countPaths(grid, row+1, col, paths) +
                countPaths(grid, row, col+1, paths);
    }
    return paths[row][col];
}
```



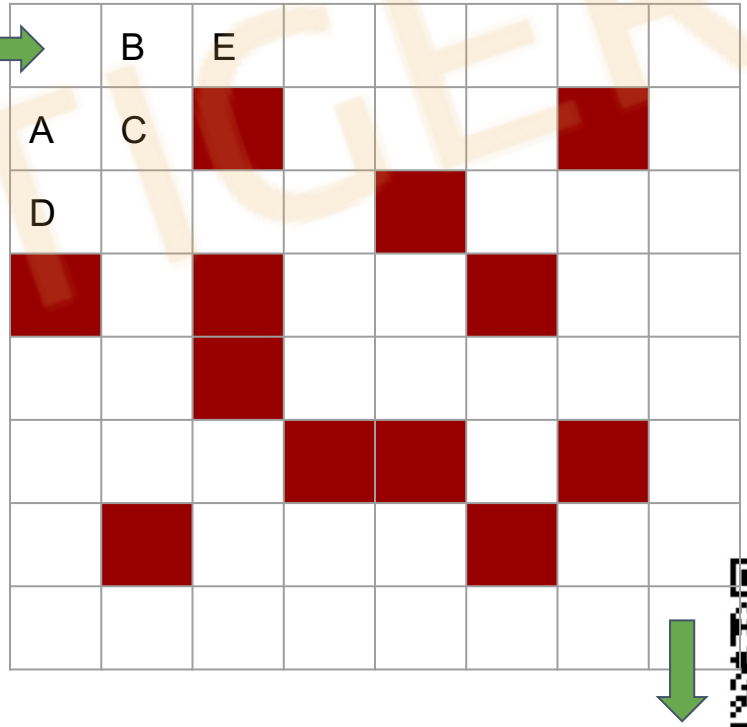
Evil Islands Problem - Memoization

Recursive: O(2^(N^2))!!

```
int countPaths(boolean[][] grid, int row, int col) {
    if (!validSquiare(grid, row, col)) return 0;
    if(isAtEnd(grid, row, col)) return 1;
    return countPaths(grid, row+1, col) +
           countPaths(grid, row, col+1);
}
```

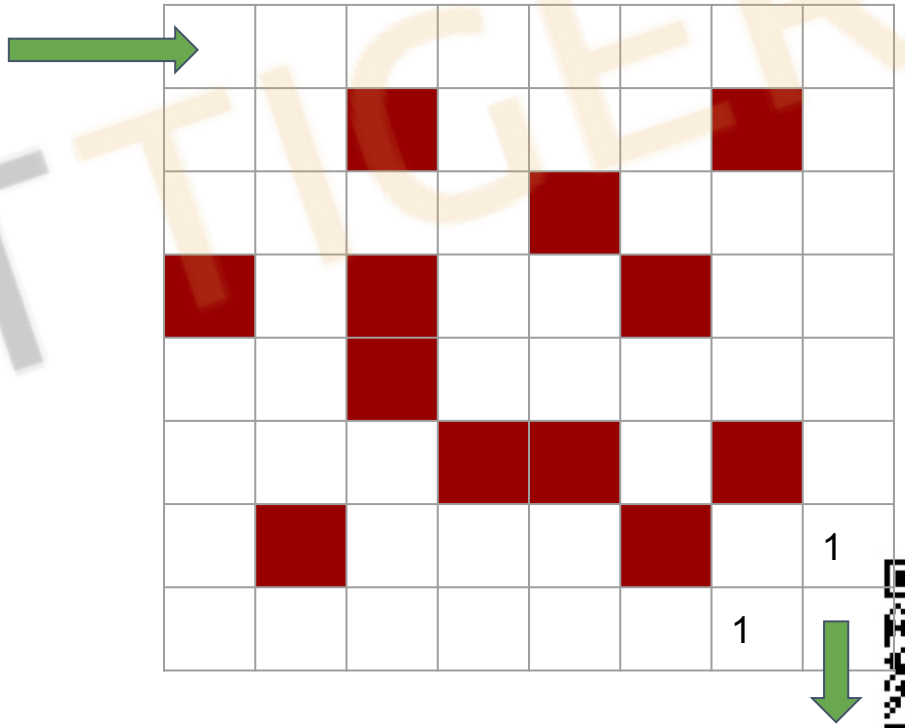
Memoization: O(N^2)

```
int countPaths(boolean[][] grid, int row, int col, int[][] paths) {
    if (!validSquiare(grid, row, col)) return 0;
    if(isAtEnd(grid, row, col)) return 1;
    if (paths[row][col] == 0) {
        paths[row][col] = countPaths(grid, row+1, col, paths) +
                countPaths(grid, row, col+1, paths);
    }
    return paths[row][col];
}
```



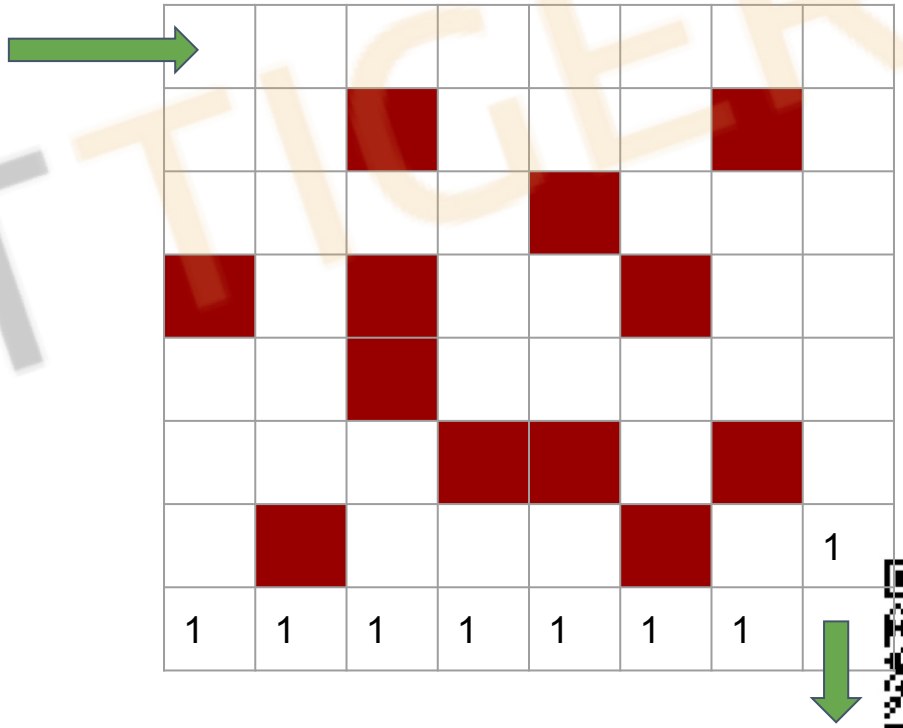
Evil Islands Problem - Tabulation

Bottom Up!!



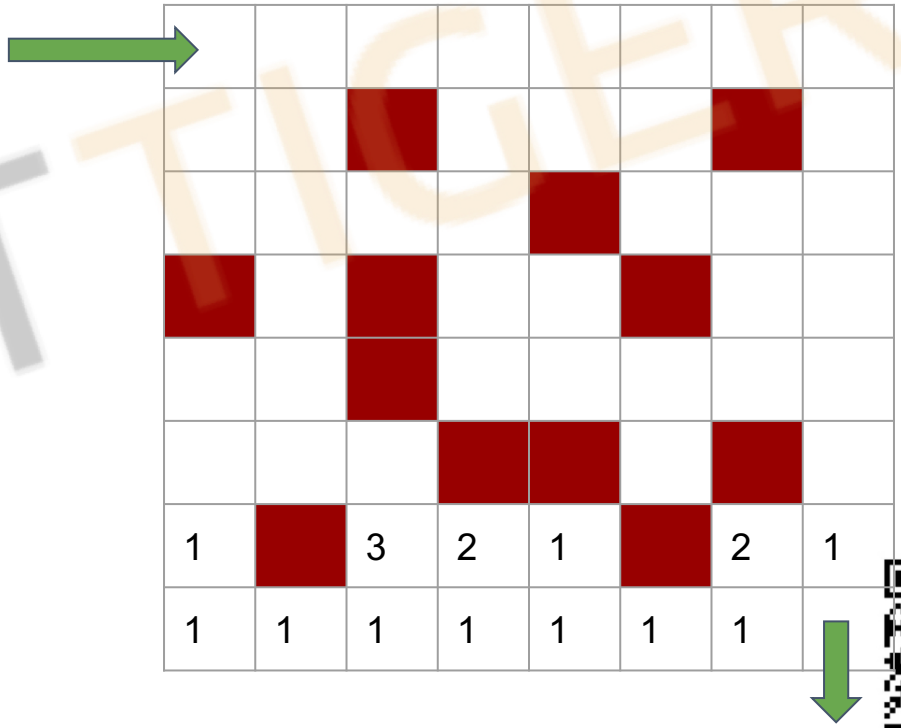
Evil Islands Problem - Tabulation

Bottom Up!!



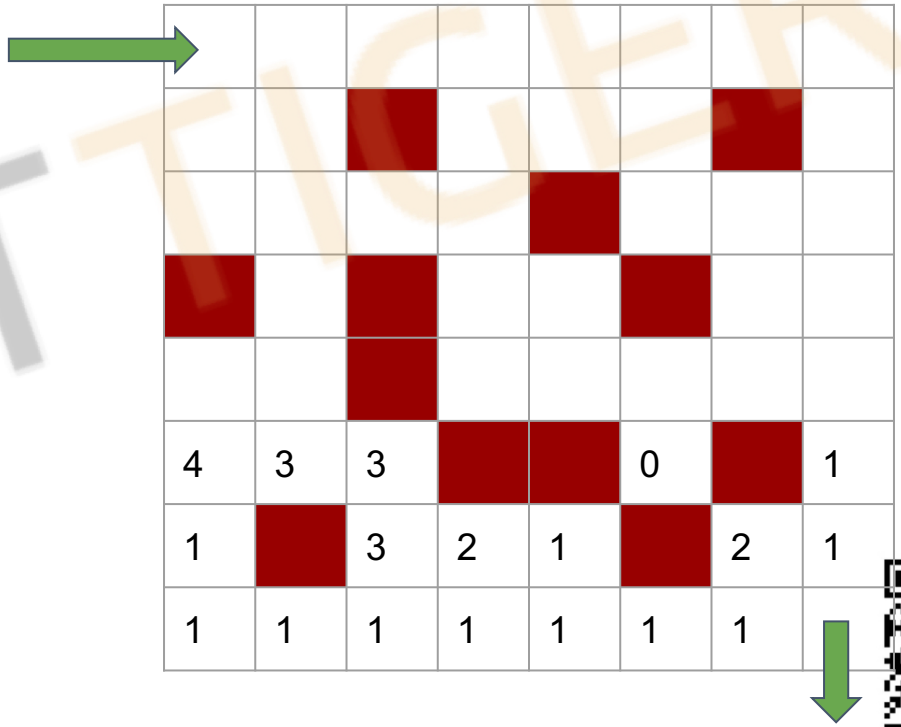
Evil Islands Problem - Tabulation

Bottom Up!!



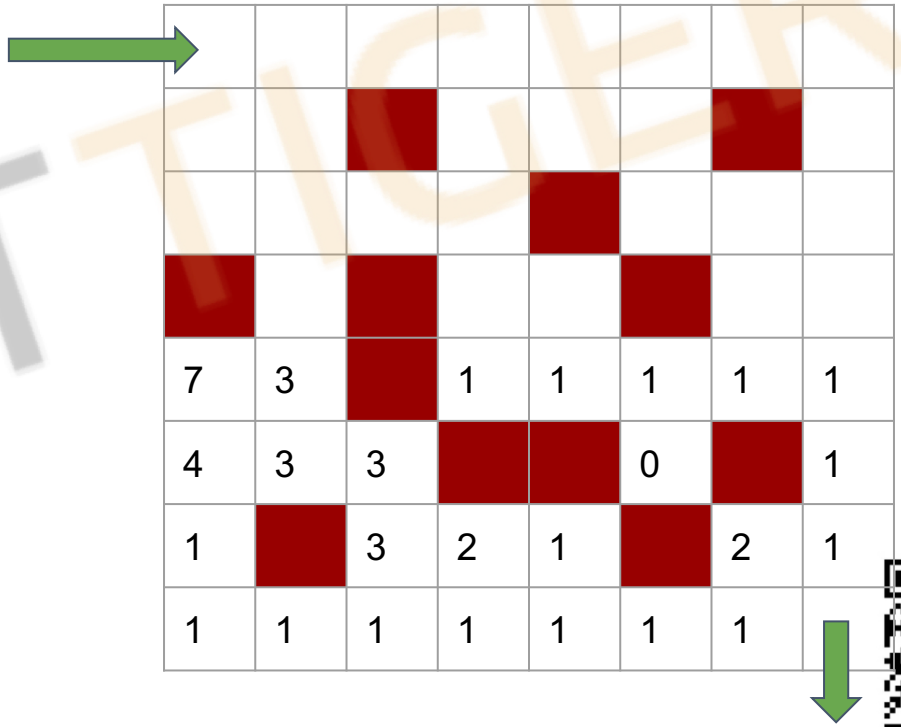
Evil Islands Problem - Tabulation

Bottom Up!!



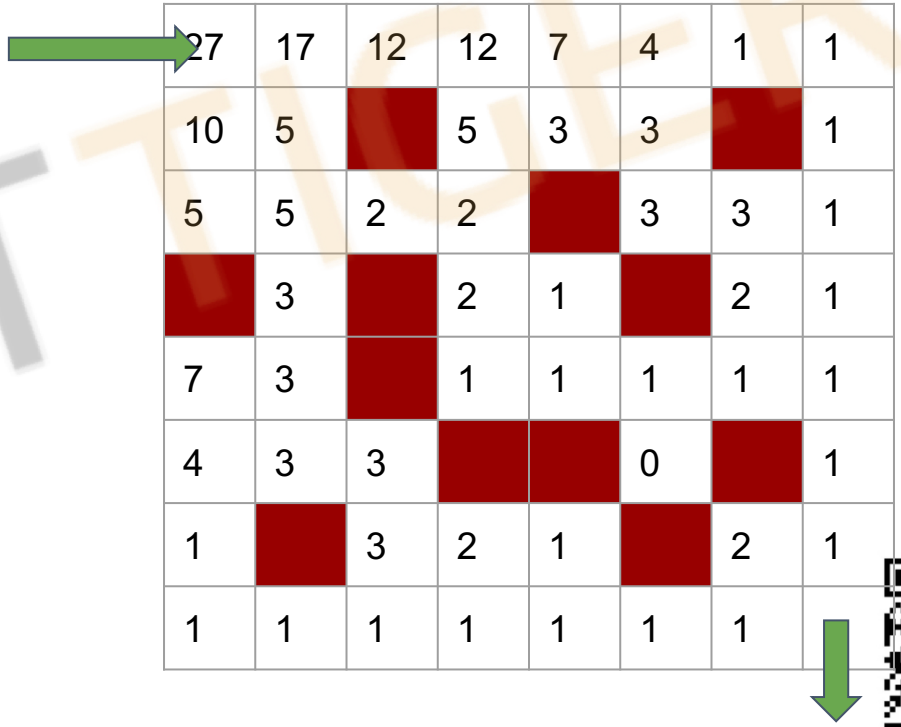
Evil Islands Problem - Tabulation

Bottom Up!!



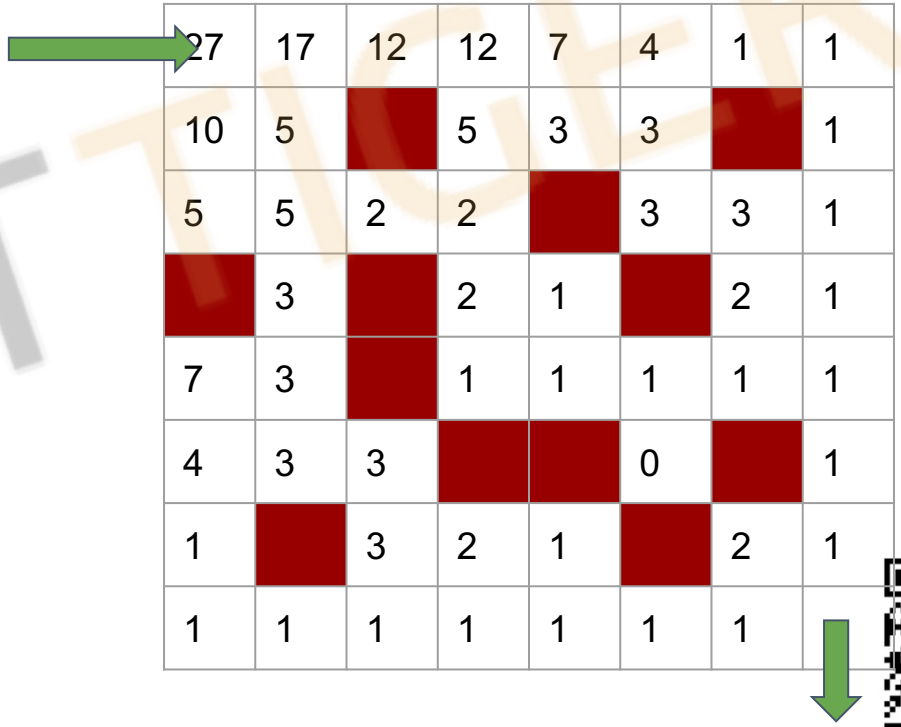
Evil Islands Problem - Tabulation

Bottom Up!!



Evil Islands Problem - Tabulation

Bottom Up!!
Time: O(N^2)



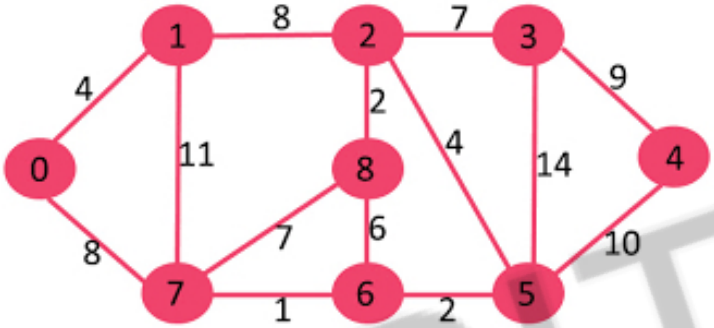
Tabulation v.s. Memoization

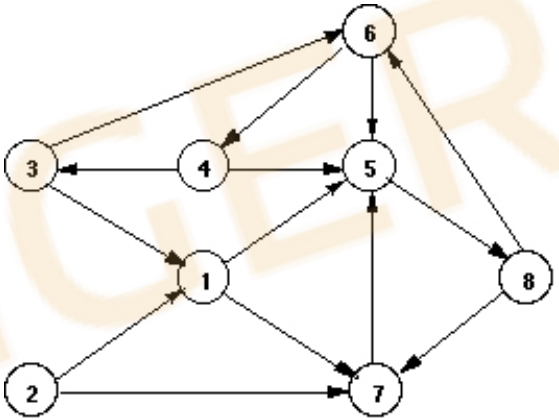
Memoization: Top down - solve the problem by maintaining a map of already solved sub problems.
Tabulation: Bottom up - solve all related sub-problems first, typically by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

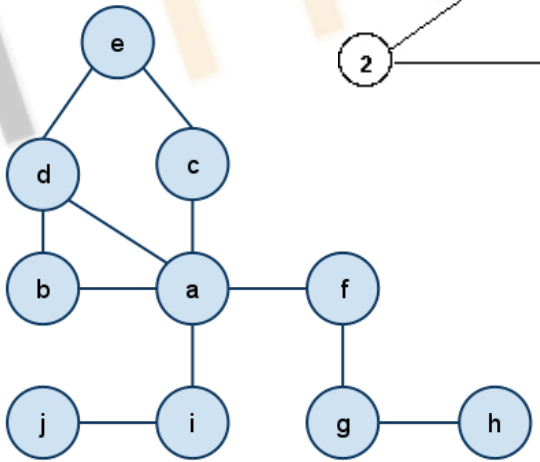
Algorithms

- Binary Search
- Sort (quickSort/mergeSort)
- Dynamic Programming
- DFS & BFS

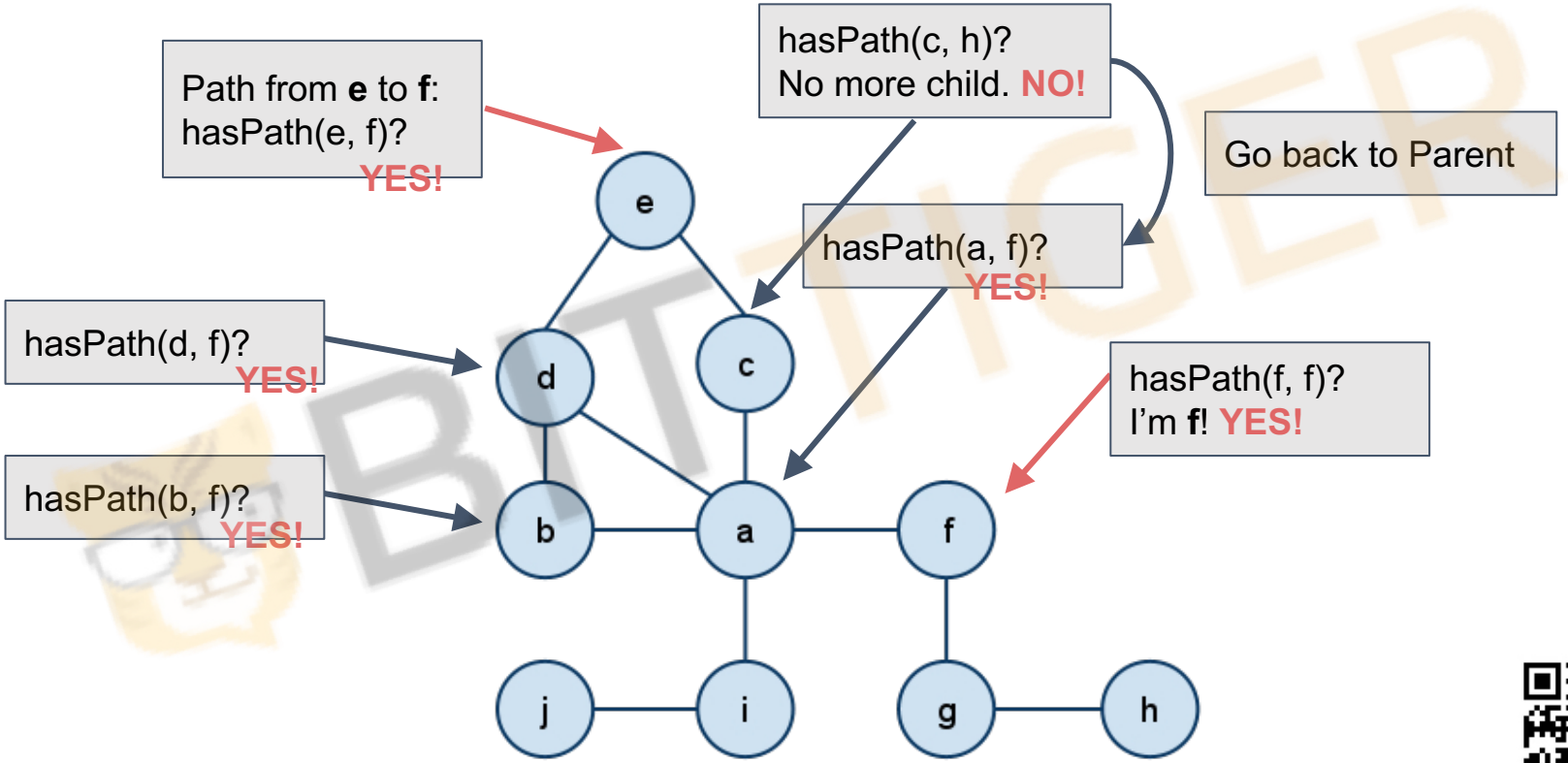
Graph





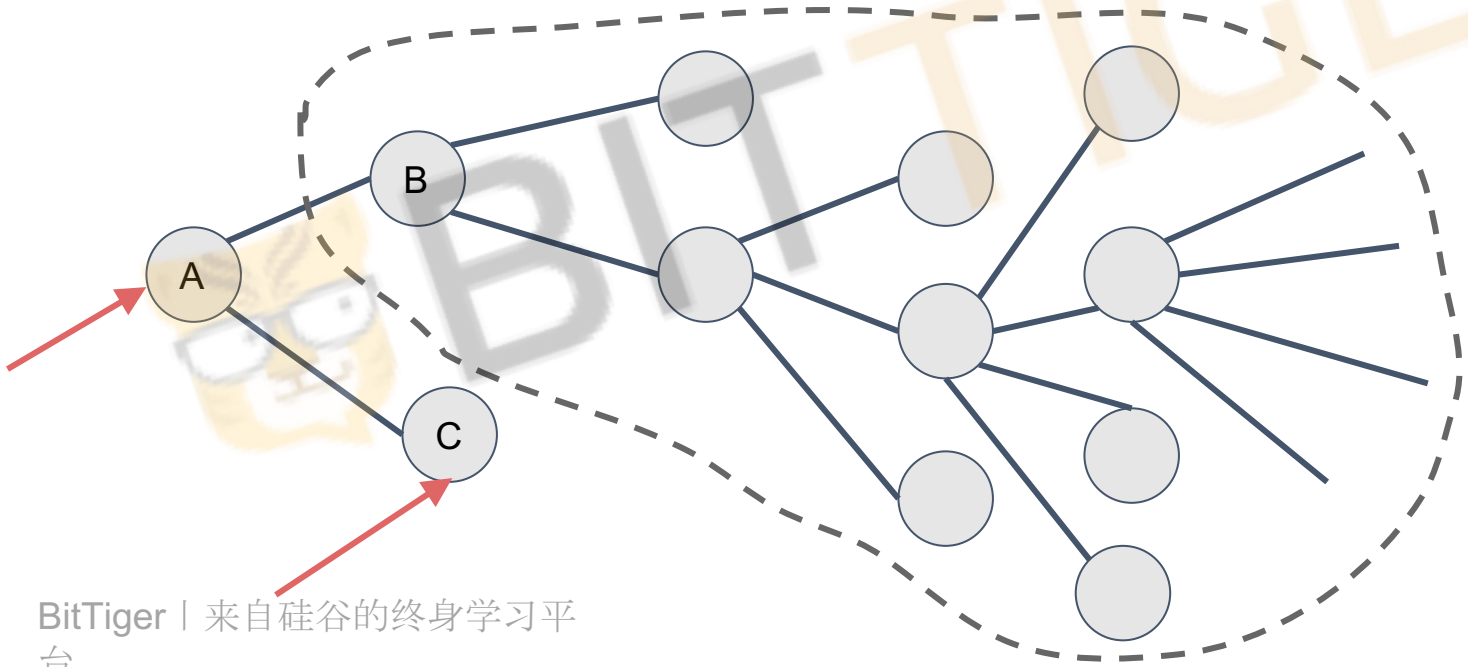


Depth First Search



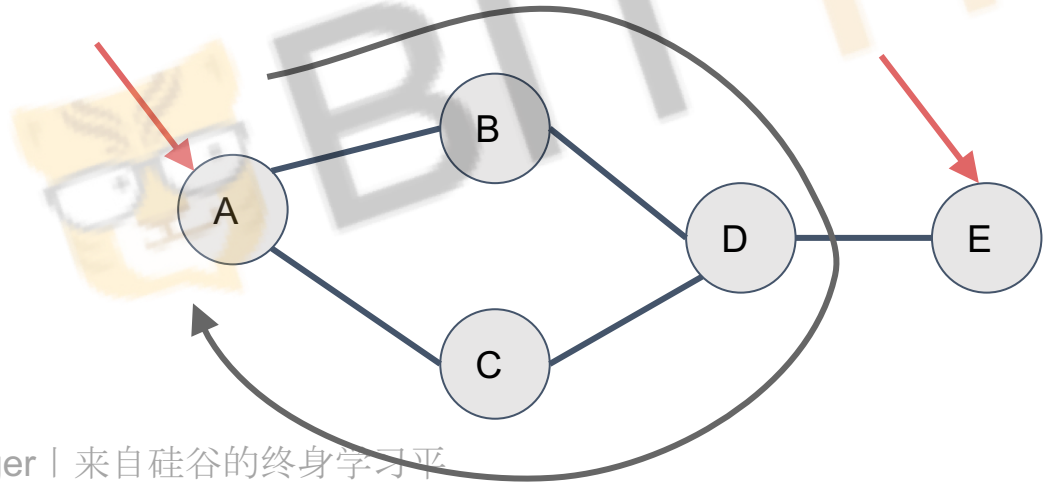
Depth First Search

Go **DEEP** to children before go broad to neighbors



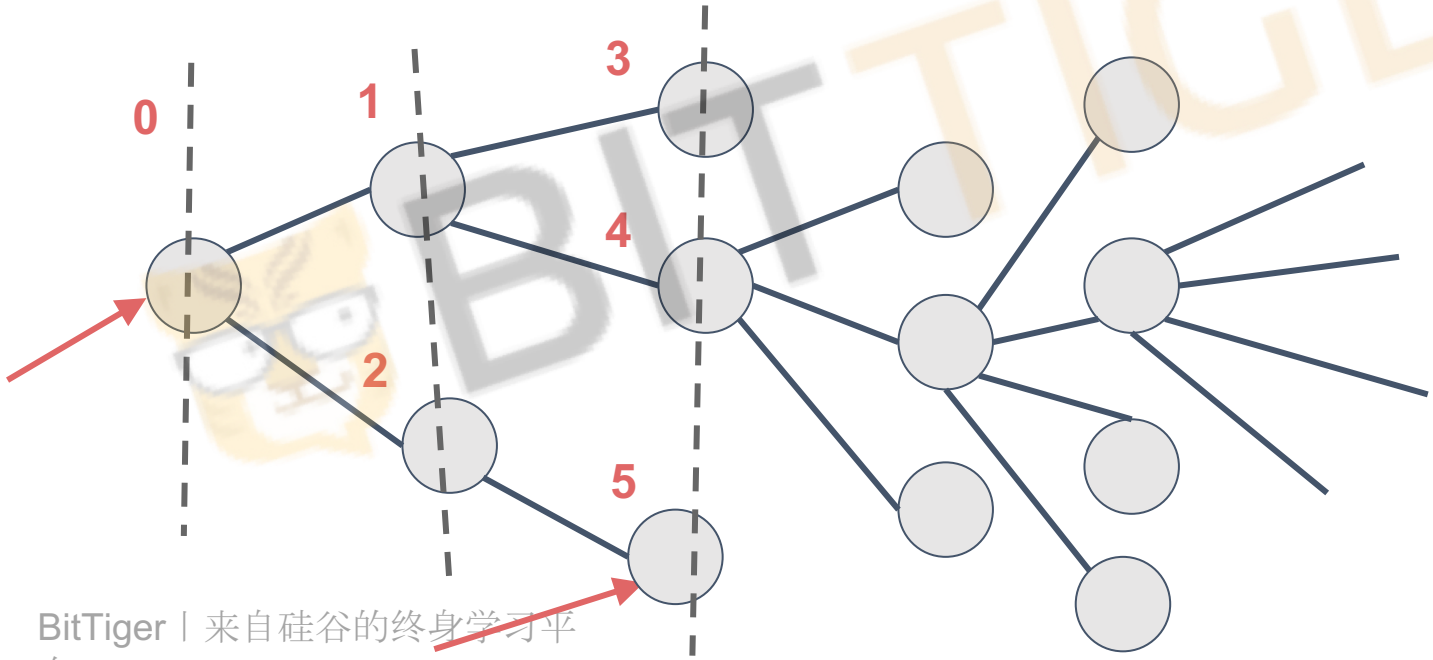
Depth First Search

Go **DEEP** to children before go broad to neighbors
Need **flag** or someway to prevent infinite loops



Breadth First Search

Go **BROAD** to neighbors before go deep to children



Learning by Doing

Number of island https://leetcode.com/problems/number-of-islands/

DFS vs BFS

DFS: usually implemented with **recursion**

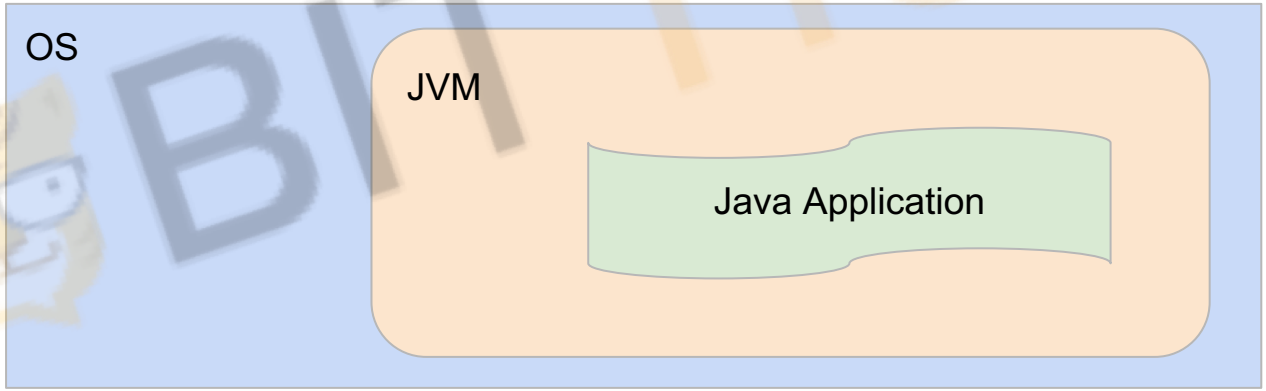
- Easy to program
- Stack overflow

BFS: usually implemented with **queue**

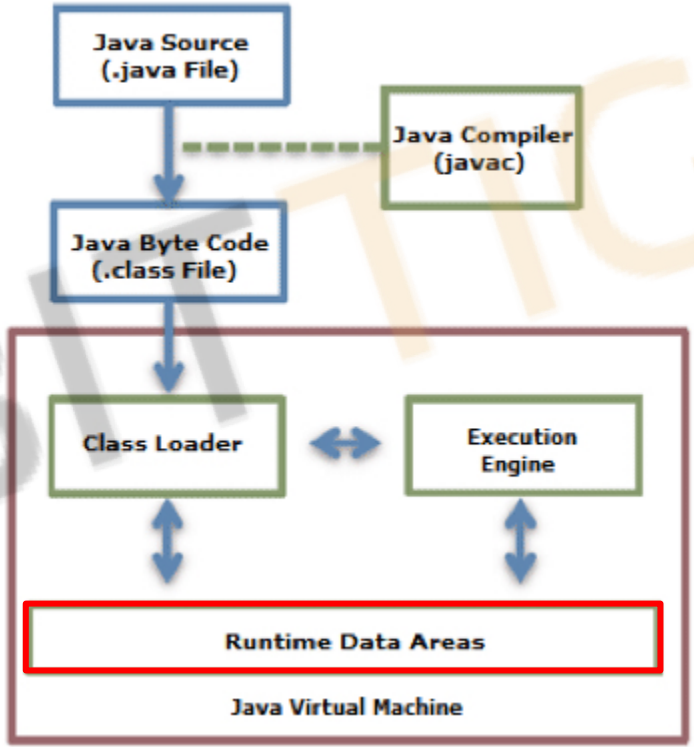
- Need to keep states of all nodes of a level (more memory)

JVM

- Java Virtual Machine
- Goal: Write Once Run Anywhere



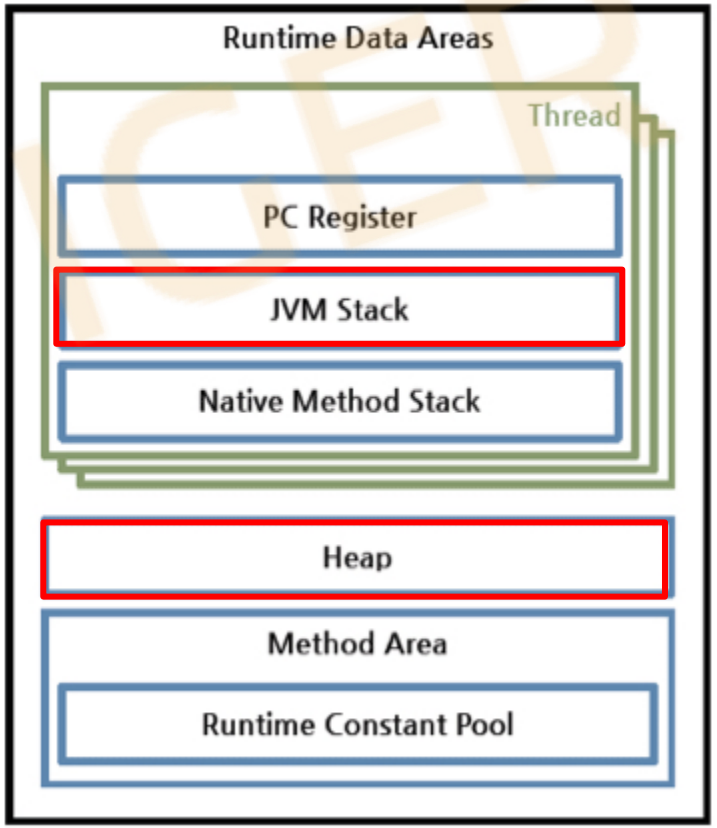
JVM Structure



Source: http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/

Runtime Data Area

- Heap
  - Allocate memory space at runtime
  - Objects
  - GC target
- JVM Stack
  - Used for execution of thread
  - Each thread has its own stack space
  - **Push** block to stack whenever method is invoked
  - **Pop** Block when method returns
  - Much less memory compared to Heap



Source: http://www.cubrid.org/blog/dev-platform/understanding-jvm-internals/

Stack & Heap

Stack & Heap

```
public void m1 () {

}
```

Stack & Heap

```
public void m1 () {

}
```

Stack & Heap

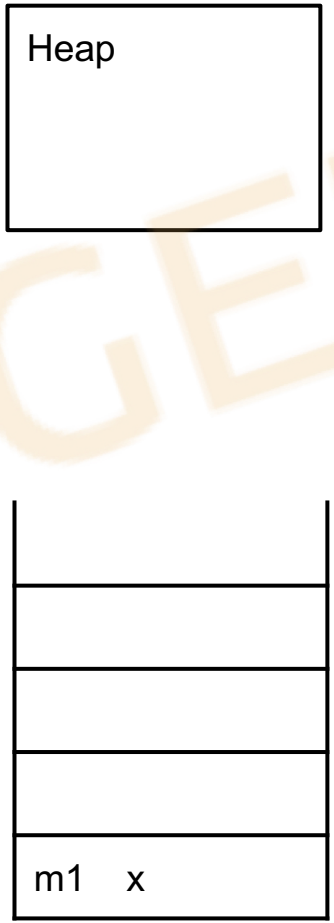
```
public void m1 () {
  int x = 1;
}
```

Stack & Heap

```
public void m1 () {
  int x = 1;
}
```

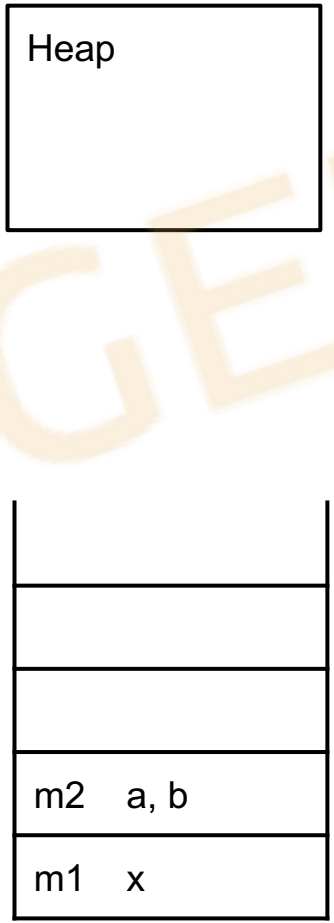
Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}

public void m2(int a) {
  int b = 2;
}
```



Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}

public void m2(int a) {
  int b = 2;
}
```

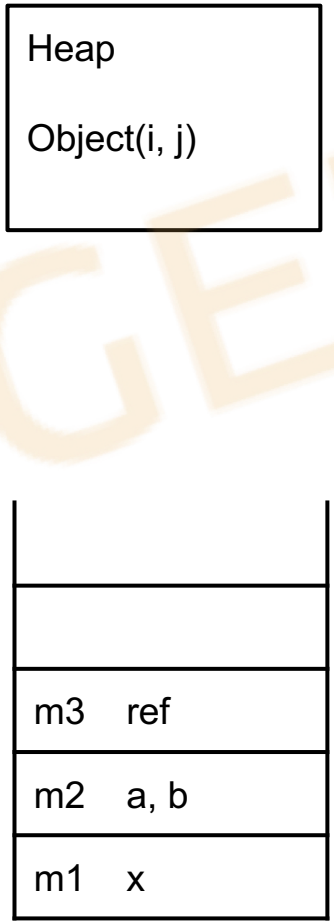


Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}

public void m2(int a) {
  int b = 2;
  m3();
}

public void m3() {
   MyObject ref = new MyObject()
}

class Object { int i; int j; }
```

Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}

public void m2(int a) {
  int b = 2;
  m3();
}

public void m3() {
   MyObject ref = new MyObject()
}

class Object { int i; int j; }
```



Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}

public void m2(int a) {
  int b = 2;
  m3();
}
```

Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}

public void m2(int a) {
  int b = 2;
  m3();
}
```

Stack & Heap

```
public void m1 () {
  int x = 1;
  m2();
}
```

Stack & Heap

Algorithms

- Binary Search
- Sort (quickSort/mergeSort)
- Dynamic Programming
- DFS & BFS

Q & A

Email: claireAtBittiger@gmail.com
Email: bobzhang2333@gmail.com