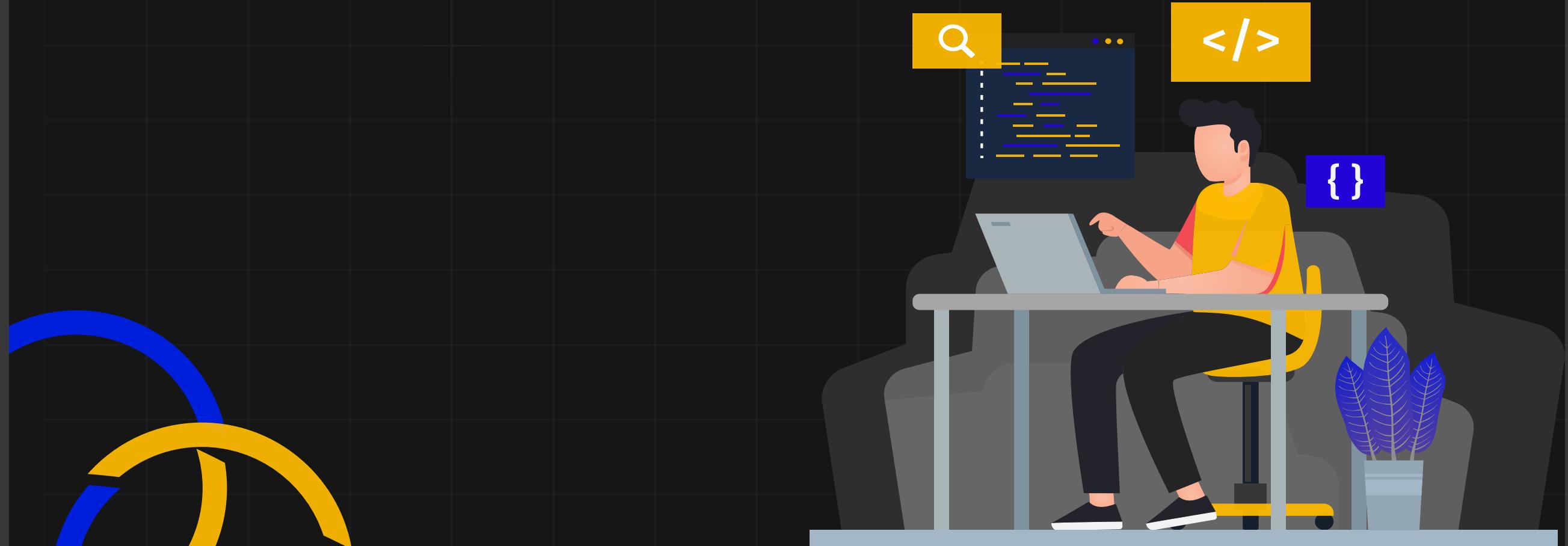


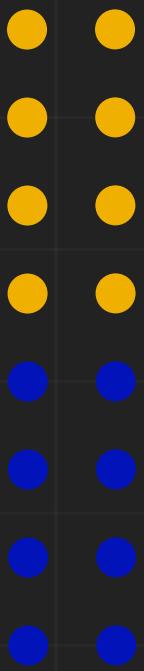


heycoach



DSA Cheat Sheet



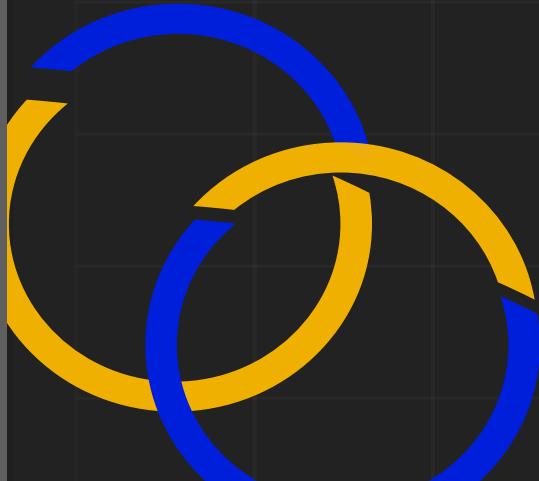


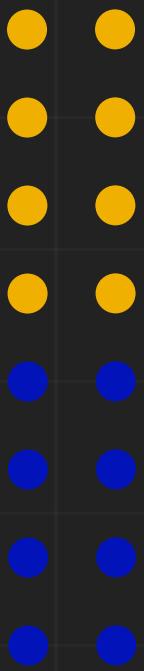
Arrays & Strings →

Stores data elements based on an sequential, most commonly 0 based, index.

Time Complexity

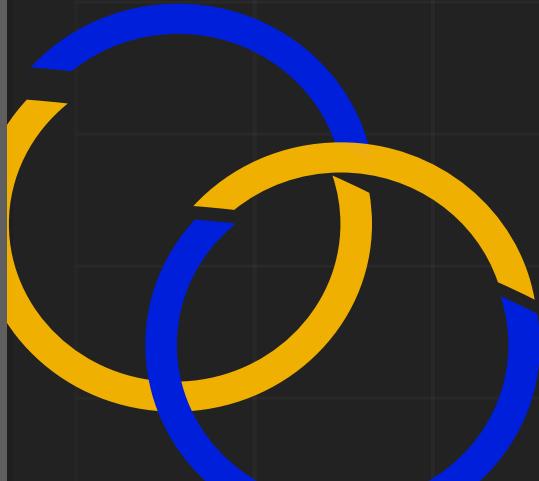
- Indexing: Linear array: $O(1)$, Dynamic array: $O(1)$
- Search: Linear array: $O(n)$, Dynamic array: $O(n)$
- Insertion: Linear array: n/a, Dynamic array: $O(n)$
- Optimized Search: Linear array: $O(\log n)$, Dynamic array: $O(\log n)$

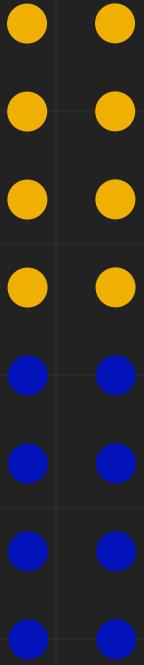




Bonus:

- `type[] name = {val1, val2, ...}`
- `Arrays.sort(arr) → O(n log(n))`
- `Collections.sort(list) → O(n log(n))`
- `int digit = '4' - '0' → 4`
- `String s = String.valueOf('e') → "e"`
- `(int) 'a' → 97 (ASCII)`
- `new String(char[] arr) ['a','e'] → "ae"`
- `(char) ('a' + 1) → 'b'`
- `Character.isLetterOrDigit(char) → true/false`
- `new ArrayList<>(anotherList); → list w/ items`
- `StringBuilder.append(char||String)`



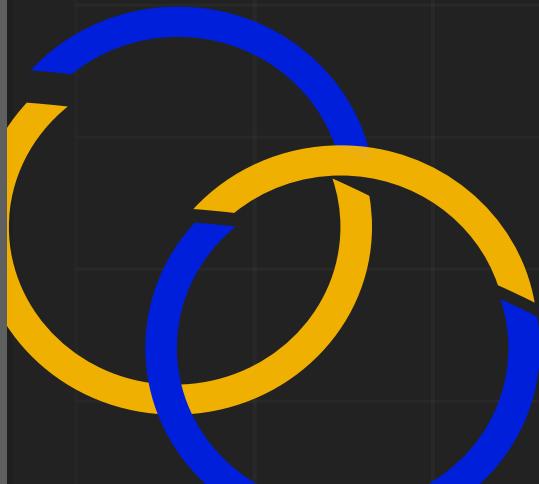


Stack/Queue/Deque →

Stack (Last In First Out, push(val), pop(), peek())
Queue (First In Last Out, offer(val), poll(), peek())
Deque (Provides first/last, offer(val), poll(), peek())
Heap (Ascending Order, offer(val), poll(), peek())

Implementation:

```
Stack <E> stack = new Stack();
Queue <E> queue = new Linked List();
Deque <E> deque = new Linked List();
PriorityQueue <E> pq = new PriorityQueue();
```





DFS & BFS Big O Notation →

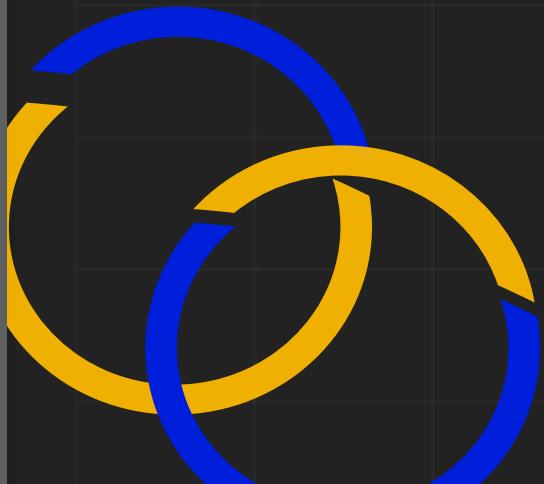
DFS (Time = $O(E+V)$ Space = $O(\text{Height})$)

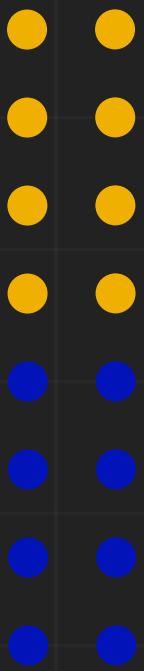
BFS (Time = $O(E+V)$ Space = $O(\text{Length})$)

V & E → where V is the number of vertices and E is the number of edges.

Height → where h is the maximum height of the tree.

Length → where l is the maximum number of nodes in a single level.





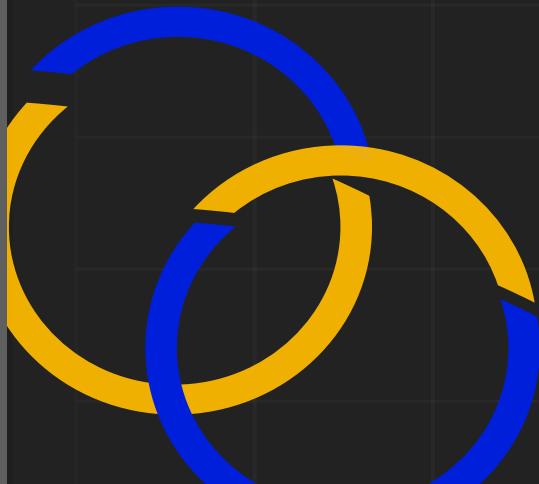
DFS vs BFS

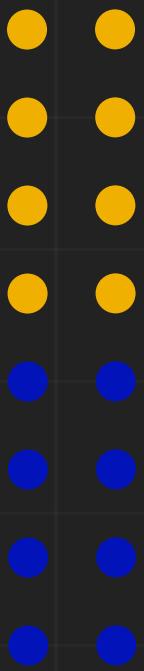
DFS:

- Better when target is closer to Source.
- Stack → LIFO
- Preorder, Inorder, Postorder
- Search
- Goes deep
- Recursive
- Fast

BFS:

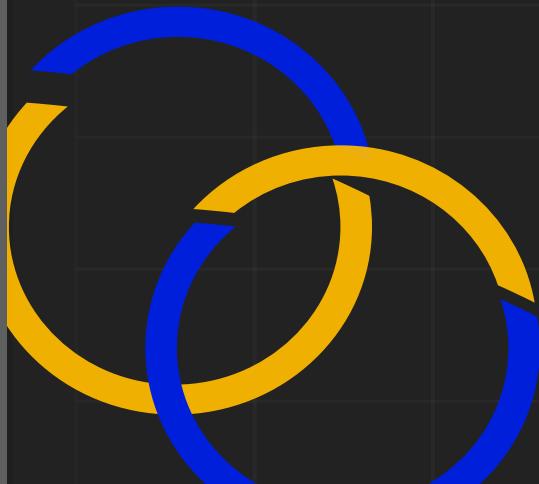
- Better when target is far from Source.
- Queue → FIFO
- Level Order Search
- Goes wide
- Iterative
- Slow

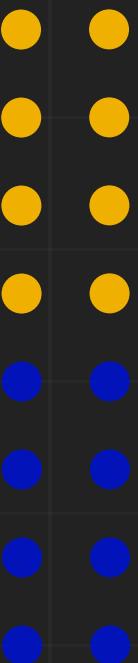




Bonus	{1, -1, 0, 2, -2} into map
HashMap	{-1, 0, 2, 1, -2} → any order
Linked HashMap	{1, -1, 0, 2, -2} → insertion order
TreeMap	{-2, -1, 0, 1, 2} → sorted

Set doesn't allow duplicates.
map.get or Default Value (key, default value)



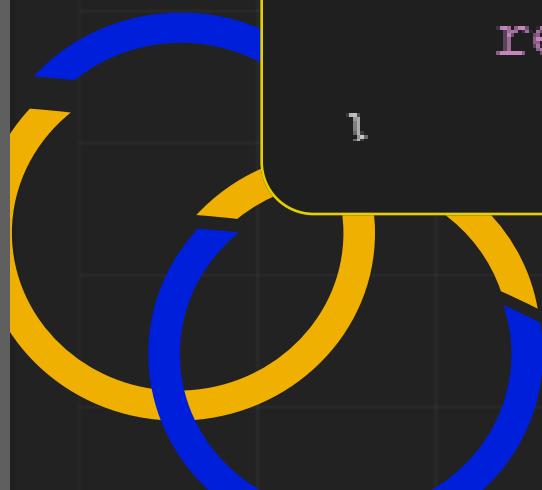


BFS Implementation For Graph:



```
public boolean connected(int[][] graph, int start, int end) {  
    Set<Integer> visited = new HashSet<>();  
    Queue<Integer> toVisit = new Linked  
List<>();  
    toVisit.enqueue(start);  
    while(!toVisit.isEmpty()) {  
        int curr = toVisit.dequeue();  
        if (visited.contains(curr)) continue;  
        if (curr == end) return true;  
        for (int i : graph[start]) {  
            toVisit.enqueue(i);  
        }  
        visited.add(curr);  
    }  
    return false;  
}
```

1





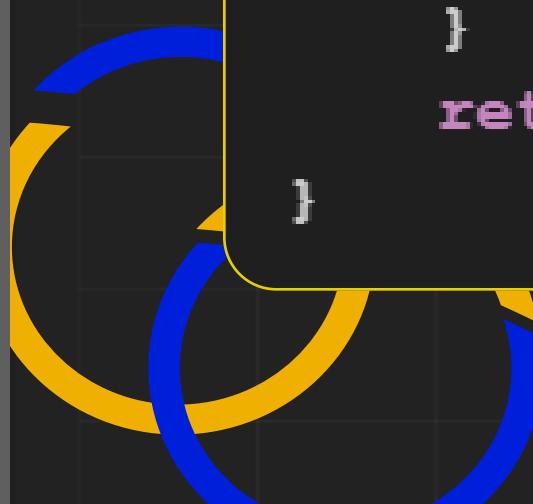
BFS Implementation For Level- Order Tree Traversal:

```
private void printLevelOrder(TreeNode root) {  
    Queue<TreeNode> queue = new Linked  
List<>();  
    queue.offer(root);  
    while (!queue.isEmpty()) {  
        TreeNode tempNode = queue.poll();  
        print( tempNode.data + " ");  
        // add left child  
        if (tempNode.left != null) {  
            queue.offer(tempNode.left);  
        }  
        // add right right child  
        if (tempNode.right != null) {  
            queue.offer(tempNode.right);  
        }  
    }  
}
```



DFS Implementation For Graph:

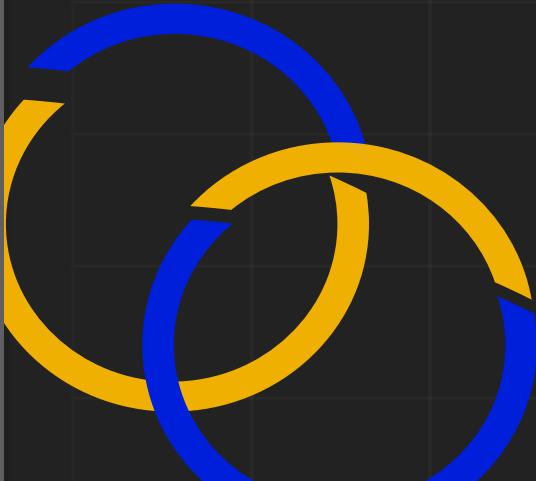
```
public boolean connected(int[][] graph, int start, int end) {  
    Set<Integer> visited = new HashSet<>();  
    return connected(graph, start, end, visited);  
}  
  
private boolean connected (int[][] graph, int start, int end, Set<Integer> visited) {  
    if (start == end) return true;  
    if (visited.contains(start)) return false;  
    visited.add(start);  
    for(int i : graph[start]) {  
        if (connected(graph, i, end, visited))  
        {  
            return true;  
        }  
    }  
    return false;  
}
```

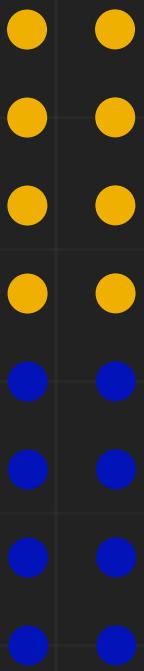




DFS Implementation For In-Order Tree Traversal:

```
private void inorder(TreeNode TreeNode) {  
    if(TreeNode == null)  
        return;  
    // Traverse left  
    inorder(TreeNode.left);  
    // Traverse root  
    print(TreeNode.data + " ");  
    // Traverse right  
    inorder(TreeNode.right);  
}
```

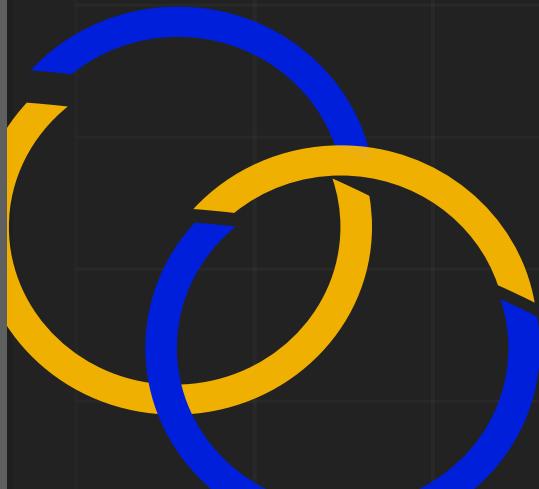


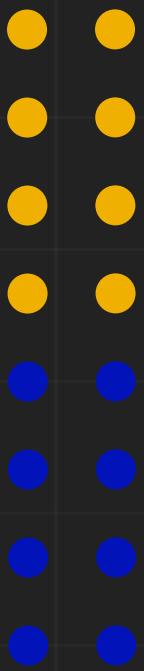


Dynamic Programming →

Dynamic programming is the technique of storing repeated computations in memory, rather than recomputing them every time you need them.

The ultimate goal of this process is to improve runtime. Dynamic programming allows you to use more space to take less time.





Dynamic Programming Patterns:

Minimum (Maximum) Path To Reach A Target

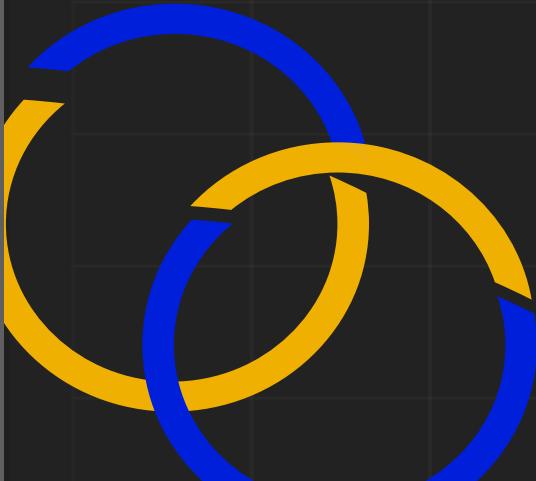
Approach:

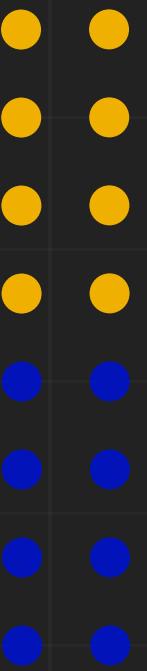
Choose Minimum (Maximum) Path Among All Possible Paths

Before

The Current State, Then Add Value For The Current State.

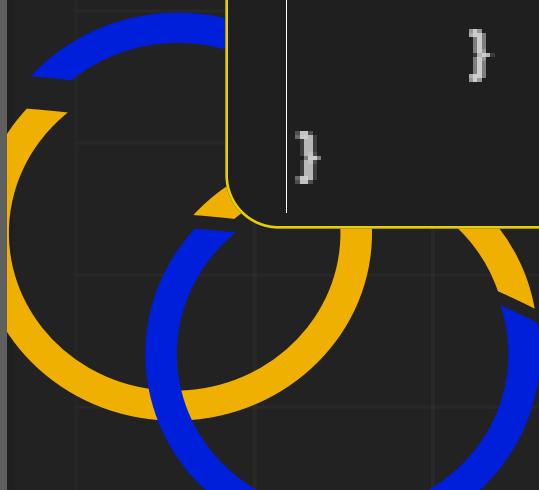
Formula:

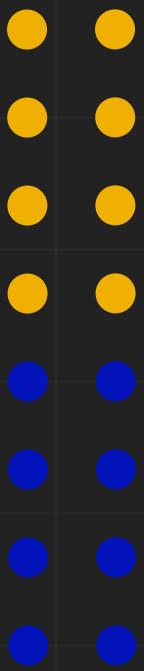
$$\text{Routes}[l] = \text{Min}(\text{RouteS}[l-1], \text{Routes}[l-2], \dots, \text{Routes}[l-K]) + \text{Cost}[l]$$




Binary Search Recursive

```
public int binarySearch(int search, int[] array, int start, int end) {  
    int middle = start + ((end - start) / 2);  
  
    if(end < start) {  
        return -1;  
    }  
    if (search == array[middle]) {  
        return middle;  
    } else if (search < array[ mid dle ]) {  
        return binary Search (search, array,  
start, middle - 1);  
    } else {  
        return binary Search (search, array,  
middle+1, end);  
    }  
}
```





Binary Search Iterative

```
public int binarySearch(int target, int[] array) {  
    int start = 0;  
    int end = array.length - 1;  
    while (start <= end) {  
        int middle = start + ((end - start) /  
2);  
        if (target == array[middle]) {  
            return target;  
        } else if (target < array[middle]) {  
            end = middle - 1;  
        } else {  
            start = middle + 1;  
        }  
    }  
    return -1;  
}
```



Distinct Ways Approach:

Choose minimum (maximum) path among all possible paths before the current state, then add value for the current state.

Formula:

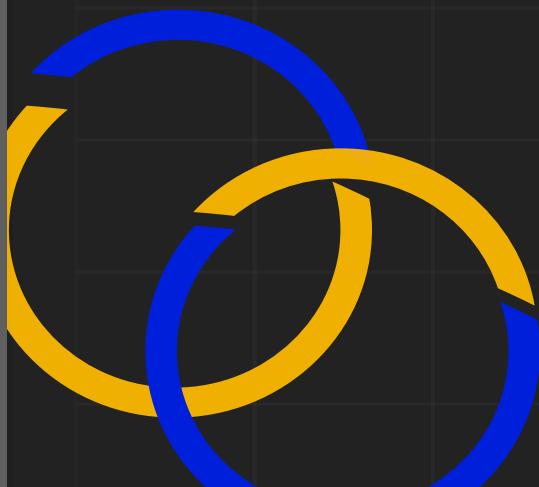
$$\text{routes}[i] = \text{routes}[i-1] + \text{routes}[i-2], \dots, + \text{routes}[i-k]$$

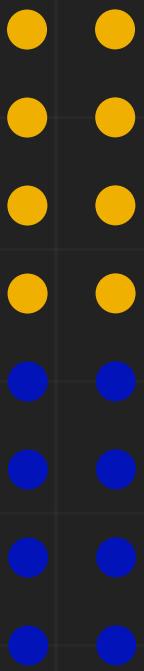
Merging Intervals Approach

Find all optimal solutions for every interval and return the best possible answer

Formula:

$$\text{dp}[i][j] = \text{dp}[i][k] + \text{result}[k] + \text{dp}[k+1][j]$$





DP On Strings

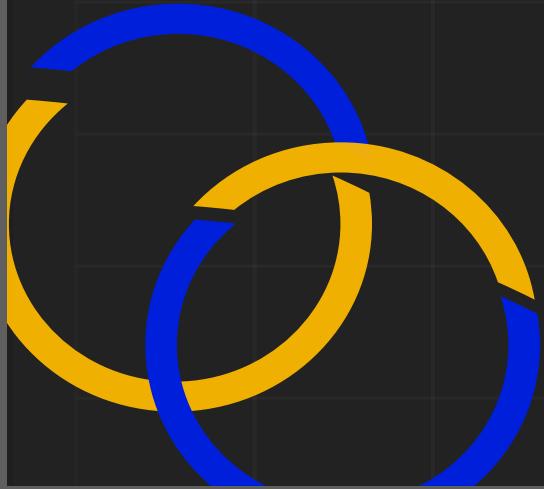
Approach:

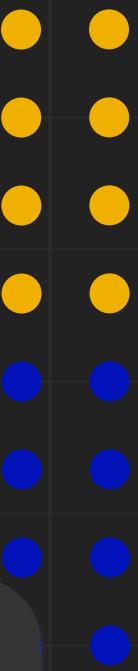
Compare 2 Chars Of String Or 2 Strings. Do Whatever You Do. Return.

Formula:

If $S1[I-1] == S2[J-1]$ Then $Dp[I][J] = //Code.$

Else $Dp[I][J] = //Code$





Bit Manipulation →

Sign Bit: 0 → Positive, 1 → Negative

AND $0 \& 0 \rightarrow 0$

$0 \& 1 \rightarrow 0$

$1 \& 1 \rightarrow 1$

OR $0 | 0 \rightarrow 0$

$0 | 1 \rightarrow 1$

$1 | 1 \rightarrow 1$

XOR $0 ^ 0 \rightarrow 0$

$0 ^ 1 \rightarrow 1$

$1 ^ 1 \rightarrow 0$

INVERT $\sim 0 \rightarrow 1$

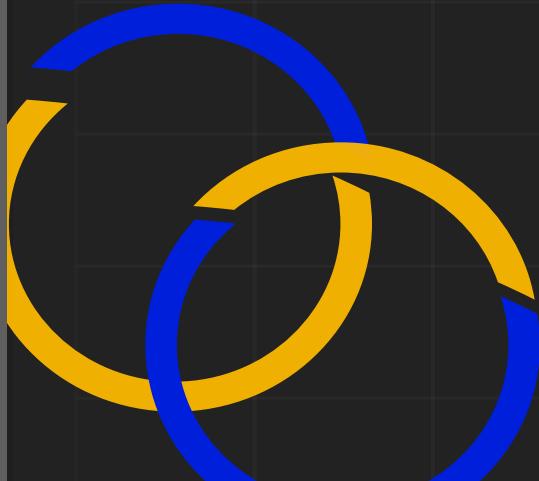
$\sim 1 \rightarrow 0$

Bonus:

Shifting

- Left Shift

$0001 << 0010$ (Multiply By 2)

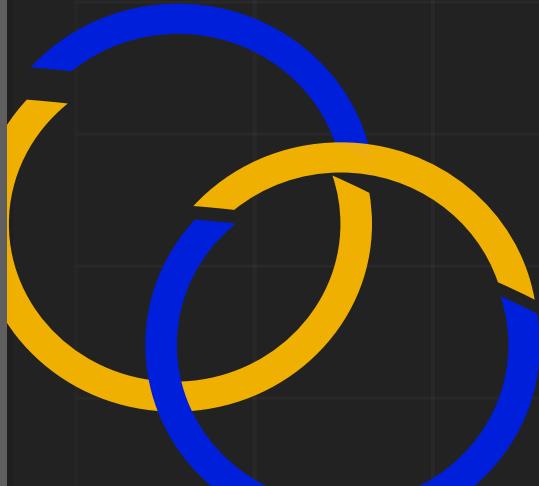


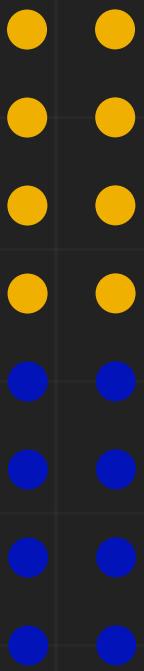


Sorting Big O Notation →

BEST AVERAGE SPACE

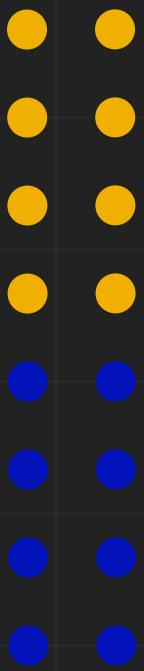
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n)$
Heap Sort	$O(n \log(n))$	$O(n \log(n))$	$O(1)$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(\log(n))$
Insertion Sort	$O(n)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(1)$





Quick Sort

```
private void mergesort(int low, int high) {  
    if (low < high) {  
        int middle = low + (high - low) / 2;  
        mergesort(low, middle);  
        mergesort(middle + 1, high);  
        merge(low, middle, high);  
    }  
}  
private void merge(int low, int middle, int  
high)  
{  
    for (int i = low; i <= high; i++) {  
        helper[i] = numbers[i];  
    }  
    int i = low;  
    int j = middle + 1;  
    int k = low;  
    while (i <= middle && j <= high) {  
        if (helper[i] <= helper[j]) {  
            numbers[k] = helper[i];  
            i++;  
        } else {  
            numbers[k] = helper[j];  
            j++;  
        }  
        k++;  
    }  
    while (i <= middle) {  
        numbers[k] = helper[i];  
        i++;  
        k++;  
    }  
}
```



Insertion Sort



```
void insertionSort(int arr[]) {  
    int n = arr.length;  
    for (int i = 1; i < n; ++i) {  
        int key = arr[i];  
        int j = i - 1;  
        while (j >= 0 && arr[j] > key) {  
            arr[j + 1] = arr[j];  
            j = j - 1;  
        }  
        arr[j + 1] = key;  
    }  
}
```



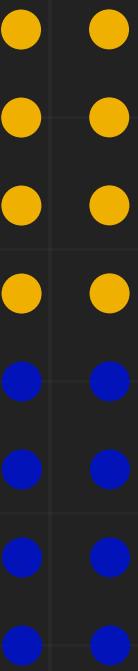


Binary Search Iterative



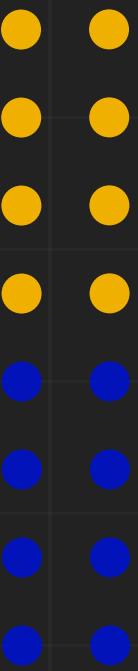
- Combination

```
public List<List <Integer>>
combinationSum(int[] nums, int target) {
    List<List <Integer>> list = new ArrayList
<>();
    Arrays.sort( nums );
    backtrack(list, new ArrayList <>(), nums,
target, 0);
    return list;
}
```



Binary Search Iterative

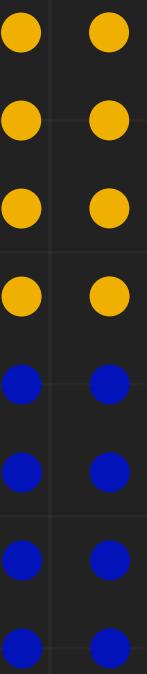
```
private void backtrack(List <List<Integer>>
list, List<Integer> tempList, int [] nums, int
remain, int start){
    if( remain < 0) return;
    else if(remain == 0) list.add(new ArrayList
< >(tempList));
    else{
        for(int i = start; i < nums.l ength;
i++) {
            tempList.add(nums[i]);
            // not i + 1 because we can reuse
            same elements
            backtrack(list, tempList, nums,
remain - nums[i], i);
            // not i + 1 because we can reuse
            same elements
            tempList.remove(tempList.size() -
1);
        }
    }
}
```



Subset Backtrack Pattern

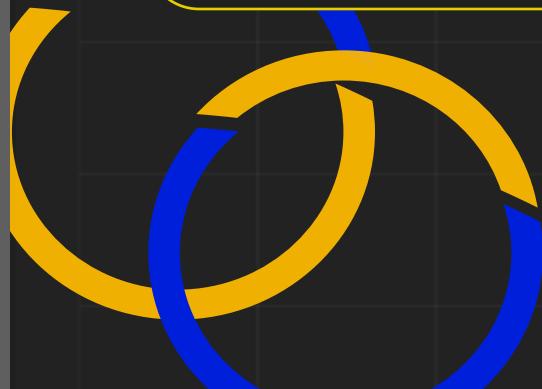
- Subsets

```
public List<List<Integer>> subsets( int[] nums) {  
    List<List<Integer>> list = new ArrayList <>();  
    Arrays.sort( nums);  
    backtrack(list, new ArrayList <>(), nums, 0);  
    return list;  
}  
  
private void backtrack(List<List<Integer>> list, List<Integer>  
tempList, int [] nums, int start){  
    list.add(new ArrayList<>(tempList));  
    for(int i = start; i < nums.length; i++){  
        // skip duplicates  
        if(i > start && nums[i] == nums[i-1])  
            continue;  
        // skip duplicates  
        tempList.add(nums[i]);  
        backtrack(list, tempList, nums, i + 1);  
        tempList.remove(tempList.size() - 1);  
    }  
}
```



Subset Backtrack Pattern

```
    }
    private void backtrack (List<List<Integer>> list,
    tempList, int [] nums, int start) {
        list.add(new ArrayList<>(tempList));
        for(int i = start; i < nums.l ength; i++) {
            if(i > start && nums[i] == nums[i-1])
                continue;
            // skip duplicates
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, i + 1);
            tempList.remove(tempList.size() - 1);
        }
    }
```





Palindrome Backtrack Pattern

- Palindrome Partitioning

```
public List<List<String>> partition(String s) {  
    List<List<String>> list = new ArrayList<>();  
    backtrack (list, new ArrayList <>(), s, 0);  
    return list;  
}  
  
public void backtrack (List<List<String>> list, List<String>  
tempList, String s, int start){  
    if(start == s.length())  
        list.add(new ArrayList <>(tempList));  
    else{  
        for(int i = start; i < s.length(); i++){  
            if(isPalindrome(s, start, i)){  
                tempList.add(s.substring(start, i + 1));  
                backtrack(list, tempList, s, i + 1);  
                tempList.remove(tempList.size() - 1);  
            }  
        }  
    }  
}
```



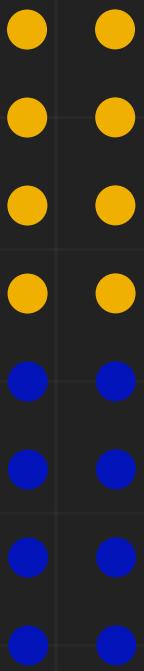
Permutation Backtrack Pattern



- Permutations

```
public List<List<Integer>> permute(int[] nums) {  
    List<List<Integer>> list = new ArrayList <>();  
    // Arrays.sort(nums); // not necess ary  
    backtrack(list, new ArrayList <>(), nums);  
    return list;  
}  
  
private void backtrack(List<List<Integer>>list, List<Integer>  
tempList, int [] nums){  
    if(tempList.size() == nums.length){  
        list.add(new ArrayList<>(tempList));  
    } else{  
        for(int i = 0; i < nums.length; i++){  
            // element already exists, skip
```





```
if(tempList.contains(nums[i])) continue;  
// element already exists, skip  
tempList.add(nums[i]);  
backtrack(list, tempList, nums);  
tempList.remove(tempList.size() - 1);  
}  
}
```

