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)
- **Optimized Search:** Linear array: O(log n), Dynamic array: O(log n)
- **Insertion:** Linear array: n/a, Dynamic array: O(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)

Binary Search Big O Notation

	Time	Space
Binary Search	O(log n)	O(1)

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[middle]) {
      return binarySearch(search, array, start, middle - 1);
   } else {
      return binarySearch(search, array, middle + 1, end);
   }
}
```

Linked List

Stores data with nodes that point to other nodes.

Time Complexity

- **Indexing:** O(n)
- **Search:** O(n)
- **Optimized Search:** O(n)
- **Append:** O(1)
- **Prepend:** O(1)
- **Insertion:** O(n)

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 (search < array[middle]) {
```

```
            end = middle - 1;
        } else {
            start = middle + 1;
        }
    }
    return -1;
}
```

HashTable

Stores data with key-value pairs.

Time Complexity
- **Indexing:** O(1)
- **Search:** O(1)
- **Insertion:** O(1)

Bonus:
- {1, -1, 0, 2, -2} into map

HashMap {-1, 0, 2, 1, -2} -> any order
LinkedHashMap {1, -1, 0, 2, -2} -> insertion order
TreeMap {-2, -1, 0, 1, 2} -> sorted
- Set doesn't allow duplicates.
- map.getOrDefaultValue(key, default value)

Stack/Queue/Deque

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

Implementation in Java:
- Stack<E> stack = new Stack();
- Queue<E> queue = new LinkedList();
- Deque<E> deque = new LinkedList();
- PriorityQueue<E> pq = new PriorityQueue();

Bit Manipulation

Sign Bit	0 -> Positive, 1 -> Negative
AND	0 & 0 -> 0 0 & 1 -> 0 1 & 1 -> 1
OR	0 | 0 -> 0 0 | 1 -> 1 1 | 1 -> 1
XOR	0 ^ 0 -> 0 0 ^ 1 -> 1 1 ^ 1 -> 0
INVERT	~ 0 -> 1 ~ 1 -> 0

Bonus:
- Shifting

– Left Shift
0001 << 0010 (Multiply by 2)
– Right Shift

0010 >> 0001 (Division by 2)

- Count 1's of n, Remove last bit

n = n & (n-1);

- Extract last bit

n&-n or n&~(n-1) or n^(n&(n-1))

- n ^ n -> 0
- n ^ 0 -> n

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)

DFS & BFS Big O Notation

	Time	Space
DFS	O(E+V)	O(Height)
BFS	O(E+V)	O(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	BFS
●Better when target is closer to Source. ●Stack -> LIFO ●Preorder, Inorder, Postorder Search ●Goes deep ●Recursive ●Fast	●Better when target is far from Source. ●Queue -> FIFO ●Level Order Search ●Goes wide

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) ● **Optimized Search:** Linear array: O(log n), Dynamic array: O(log n) ● **Insertion:** Linear array: n/a, Dynamic array: O(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)

Linked List

Stores data with nodes that point to other nodes.
Time Complexity ● **Indexing:** O(n) ● **Search:** O(n) ● **Optimized Search:** O(n) ● **Append:** O(1) ● **Prepend:** O(1) ● **Insertion:** O(n)

HashTable

Stores data with key-value pairs.
Time Complexity ● **Indexing:** O(1) ● **Search:** O(1) ● **Insertion:** O(1)
Bonus: ● {1, -1, 0, 2, -2} into map HashMap {-1, 0, 2, 1, -2} -> any order LinkedHashMap {1, -1, 0, 2, -2} -> insertion order TreeMap {-2, -1, 0, 1, 2} -> sorted ● Set doesn't allow duplicates. ● map.getOrDefaultValue(key, default value)

Stack/Queue/Deque

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

Implementation in Java:

- Stack<E> stack = new Stack();
- Queue<E> queue = new LinkedList();
- Deque<E> deque = new LinkedList();
- PriorityQueue<E> pq = new PriorityQueue();

DFS & BFS Big O Notation

	Time	**Space**
DFS	O(E+V)	O(Height)
BFS	O(E+V)	O(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	BFS
●Better when target is closer to Source. ●Stack -> LIFO ●Preorder, Inorder, Postorder Search ●Goes deep ●Recursive ●Fast	●Better when target is far from Source. ●Queue -> FIFO ●Level Order Search ●Goes wide ●Iterative ●Slow

BFS Impl for Graph

```
public boolean connected(int[][] graph, int start, int end) {
  Set<Integer> visited = new HashSet<>();
  Queue<Integer> toVisit = new LinkedList<>();
```

```
  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;
}
```

DFS Impl 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;
}
```

BFS Impl. for Level-order Tree Traversal

```
private void printLevelOrder(TreeNode root) {
  Queue<TreeNode> queue = new LinkedList<>();
  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);
    }
  }
}
```

Merge 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];
                k++;
                 i++;
           }
      }
```

Quick Sort

```
private void quicksort(int low, int high) {
         int i = low, j = high;
         int pivot = numbers[low + (high-low)/2];
         while (i <= j) {
                 while (numbers[i] < pivot) {
                       i++;
                 }
                 while (numbers[j] > pivot) {
                        j--;
                  }
                   if (i <= j) {
                       exchange(i, j);
                       i++;
                       j--;
                  }
         }
          if (low < j)
                 quicksort(low, j);
         if (i < high)
                 quicksort(i, high);
     }
```

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;
   }
}
```

Combinations Backtrack Pattern

```
- 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;
}

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.length; 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);
```

```
        }
    }
}
```

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);
         }
      }
   }
}
```

Subsets 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);
    }
}
```

Permutations Backtrack Pattern

```
- Permutations
public List<List<Integer>> permute(int[] nums) {
   List<List<Integer>> list = new ArrayList<>();
   // Arrays.sort(nums); // not necessary
   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);
      }
   }
}
```

BFS Impl for Graph

```
public boolean connected(int[][] graph, int start, int end) {
  Set<Integer> visited = new HashSet<>();
  Queue<Integer> toVisit = new LinkedList<>();
  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;
}
```

DFS Impl 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;
}
```

BFS Impl. for Level-order Tree Traversal

```
private void printLevelOrder(TreeNode root) {
  Queue<TreeNode> queue = new LinkedList<>();
  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 Impl. 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:
routes[i] = min(routes[i-1], routes[i-2], ... , routes[i-k]) + cost[i]

- Distinct Ways

Approach:
Choose minimum (maximum) path among all possible paths before the current state, then add value for the current state.
Formula:
routes[i] = routes[i-1] + routes[i-2], ... , + routes[i-k]

- Merging Intervals

Approach:
Find all optimal solutions for every interval and return the best possible answer
Formula:
dp[i][j] = dp[i][k] + result[k] + 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

- Decision Making

Approach:
If you decide to choose the current value use the previous result where the value was ignored; vice-versa, if you decide to ignore the current value use previous result where value was used.
Formula:
dp[i][j] = max({dp[i][j], dp[i-1][j] + arr[i], dp[i-1][j-1]});
dp[i][j-1] = max({dp[i][j-1], dp[i-1][j-1] + arr[i], arr[i]});