# Memorize these points

## Some important calculation of runtime/space complexity

### Recursive Runtimes and Space Complexities

int f(int n) {

if(n<=0) return 1;

return f(n-1)+f(n-1);

}

How many recursive calls?

**Runtime complexity of Recursive methods = O(branches ^ depth)**

At each step, we are calling recursion twice and number reduces by 1. So, total number of recursions is 2^n. So, **Runtime Complexiy = O(2^N) that is same as number of branches in the tree.**

f(4)

f (3) f(3)

f(2) f(2) f(2) f(2)

f(1) f(1) f(1) f(1) f(1) f(1) f(1) f(1)

f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0) f(0)

Height of tree is 4. Number of stack slots will be used is 4 = **O(n) is the space complexity that is same as height of the tree**.

Let’s take an example of Binary Search

BinarySearch(8)

BinarySearch (4) BinarySearch (4)

(not used)

BinarySearch (2) BinarySearch (2)

(not used)

BinarySearch (1) BinarySearch (1)

(not used)

At each level only 1 branch is used for comparison. So, number of branches is approx log 8. So, runtime complexity is log 8.

Heght of the tree is log 8. So, space complexity is log 8.

### When number of comparisons at each step doubles

1+2+4+8+16+……X

Runtime complexity = 2^n+1 -1 = O(2^n)

The sum of the sequence of powers of two is roughly equal o the next value in the sequence.

2^0 +2^1 +2^2 +2^3+2^4 = 2^5 - 1

### When number of comparisons at each step halvs

Let’s take Binary Search algorithm

At every step, number of elements available to compare halvs

N=16

N=8

N=4

N=2

N=1

N = 2^k

Log2 N = k

e.g.

16 = 2^4

log2 16 = 4

This is a bit hard to understand. Understand the explanation given in [Binary Search](#_Binary_Search) section.

Basically, when number of elements halvs at each step and number of comparisions at each step is 1 only, then runtime complexity is O(log n). This is the case with Binary Search.

In Quick Sort also, number of

### When number of comparisions increases by 1 at each step

(pg 46, 47 of CCA book)

for(int i=0; i<n; i++) {

for(int j=0; i<n; i++) {

…

}

}

Number of operations – n + n-1+ n-2 + n-3 +…..+ 1

So it is 1 + 2 + 3 + 4 + ….. n

It will come to n(n+1)/2, which will be O(n^2). This is how you calculate the total

<https://www.wikihow.com/Sum-the-Integers-from-1-to-N>



Similarly,

for(int i=0; i<n; i++) {

for(int j=i+1; i<n; i++) {

…

}

}

number of comparisons n-1 + n-2 + n-3 +…… 1

so, it is 1 + 2 + 3 + 4 + ….. n-1

In above fomula (n(n+1)/2), replace n by n-1

So, it will be (n-1)n/2, which will be O(n^2)

### How long it takes to Sort Strings (not integers)?

(pg 49 of CCA book)

To sort an array of integers, quick sort takes O(n log n), we know that. During quick sort, when comparison of 2 integers happens, it takes O(1). Look at Integer class’ compareTo method.

But in case of Strings, to compare two strings of size s takes O(s). So, sorting of strings will take O(sn log n).

## Tricking question

for(int x=2; x\*x <= n; x++) {

….

}

What is runtime complexity?

x\*x <= n

x^2 <= n

x <= sqrt(n)

So, runtime complexity= O(sqrt(n))

### Momizaiton example

Pg 53 of CCA book

Fibonacci example with memoization

## O(n!) example

Pg 51 of CCA book

## How to work with array?

binarySearchRecursive(array, 0, array.**length** - 1, elementToSearch)

* Always pass start and end element position in array to recursive method.
* One of the Exit condition will be if(start<end)…
* When you need to convert recursive method into iterative method, extra passed parameters to recursive method becomes local variables and after that that you need to add a while loop for reoccurring code.

## How to find mid of array?

binarySearchRecursive(array, 0, array.**length** - 1, elementToSearch)

void binarySearchRecursive(int[] array, int start, int end) {

…

mid = (end+start)/2

…

}

## How to choose Random number from array?

quickSort (A, 0, A.length – 1);

void quickSort(**int**[] A, **int** start, **int** end) {

…

**int** pivot = **new** Random().nextInt((end - start) + 1) + start;

or

**int** pivot = **new** Random().nextInt(A.length - start) + start;

…

}

## How to do shuffling?

void shuffle(int[] A) {

// if you start from i=0, new Random(0,0) will error out

for(int i=1; i < A.length;i++) {

// pick random number between 0 and i

// People sometimes choose random number between 0 and n-1, but it doesn't give uniformly random result

int randomIndex = new Random().nextInt(0, i);

exchange(A, i, randomIndex);

}

}

## How to write an in-place algorithm?

Think of the difference between Quick Sort using Aux arrays and in-place.

Just like Merge Sort, Quick Sort is also Divide and Concur algorithm.

Just like Merge Sort, you can create aux arrays in Quick Sort also, but better approach is to do quick sort in-place.

This teaches us a trick:

whenever you need to do something in-place, think of using an additional pointer. One pointer is for normal traversal of an array and another pointer increments on some special condition. Hard thing is to find this special condition.

## What makes the running time n! instead of n^2?

For(int i=0; i<n; i++) {

For(int j=0; j<n; j++) {

….

}

}

This is n^2 operations.

Let’s say, n=10 (0 to 9). You have 10 dots in a graph. Starting from 0, you need to find all combinations to reach to 9. There can be many combinations. 0->1->2…->9, 0->2->1->…9 etc.

This kind of behavior needs n! operations.

Greedy Algorithm or Dynamic Programming is a solution for n! problem. Greedy Algorithm gives close to optimal solution at the end and Dynamic Programming give most optimal solution at the end, but you cannot Dynamic Programming all the time. For Dynamic Programming is for Knacksap Problem. In Knacksap problem, you need to be told the max weight that Knacksap can hold and every item that you can put in knacksap has to have concrete weight. You cannot say whatever weight is available, I will put it in knacksap.

## How many ways to create a graph?

1. Hash Table



****

1. Matrix

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Tushar | Miral | Srikant | Anoop | Madhu | Rakesh | Yogita | Puja | Ronak | NotConnectedPerson1 | NotConnectedPerson2 |
| Tushar | 1 | 1 | 1 | 1 | 1 | 1 |  |  |  |  |  |
| Miral | 1 | 1 | 1 |  |  |  | 1 | 1 |  |  |  |
| Srikant | 1 | 1 | 1 |  |  |  |  |  | 1 |  |  |
| Anoop |  |  |  |  |  |  |  |  |  |  |  |
| Madhu |  |  |  |  |  |  |  |  |  |  |  |
| Rakesh |  |  |  |  |  |  |  |  |  |  |  |
| Yogita |  |  |  |  |  |  |  | 1 |  |  |  |
| Puja |  |  |  |  |  |  |  |  | 1 |  |  |
| Ronak |  |  |  |  |  |  |  |  |  |  |  |
| NotConnectedPerson1 |  |  |  |  |  |  |  |  |  |  | 1 |
| NotConnectedPerson2 |  |  |  |  |  |  |  |  |  |  |  |

Matrix is created using 2-D array

A[0][0] = [Tushar][Tushar]

A[0][1] = [Tushar][Miral]

and so on

## When to use Linear Programming?

<https://www.youtube.com/watch?v=M4K6HYLHREQ>

Under some constraint, you need to maximize something (e.g. profit) As mentioned in above youtube video, constraint is number of available hours for farming. In that constraint, you need to maximize the profit by putting some corns and some oats plants giving different amount of profits.

## When to use Binary Search Tree instead of Binary Search?

Binary Search needs sorted array. Whenever a new element needs to be added in an array, array needs to be sorted. This is not good.

BST works BEST on unordered array with O(log n). It works worse on ordered array with O(n) search time.

BST is a symmetric tree. Means left nodes of any nodes are always lesser and right nodes are always bigger.

BST takes O(log n) for insert/delete. It takes avg O(log n) and worse O(n) for search. Worst case is when BST is created from sorted array. BST will have all elements on one side of the tree only. That's why height of the tree will be n because it won't be balanced.

Unbalanced Tree:



Comparing BST to Binary Search:

For Binary Search, you need sorted array, so Inserting and deleting in array to keep it sorted may take O(n). Search takes same as Binary Search.

So BST has an advantage over Binary Search for insert/delete/search when array is unordered.



**Which algorithm is used to create perfectly balanced tree?**

Red-Black Tree

### Which algorithm is used by Databases?

B-Tree

### Which algorithm is used for finding min or max?

Min-Heap, Max-Heap (prioritiy queue)

Priority Queue uses Heap Sort.

Heap Sort is very useful when you need to find min/max in O(1) time and insert an element in O(log n) time. It requires an aux array through. so O(n) space and total execution time is O(nlogn).

Priority Queue is based on Binary Heap (BinaryHeap.java in algorithms package).  
There is Min BinaryHeap and Max BinaryHeap.  
BinaryHeap look like a tree, but it is just a reordering of elements in an array. Based on index you can find higher priority the element.  
You can find min priority element on the top of Min BinaryHeap, you don't need to search for it like BST.

### When you encounter a problem that has inputs from multiple arrays (multiple sources), what should you think of?

Using Priority Queue to store inputs coming from multiple sources. Priority Queue.

### Arrays.sort, Collections.sort

* Arrays.sort uses 3-Way-QuickSort for int[], float[] etc. But it uses Merge Sort/Insertion Sort for Object[].

If positions for primitives are changed during sorting, then it's ok, but it's not ok for Objects.

* Collections.sort uses Arrays.sort internally.
* Heap Sort uses Binary Heap algorithm and Priority Queue uses Heap Sort.

Remember, Quick Sort is a in-place sorting algorithm but it is unstable.

From BSIS and M-HQ sorts, HQ are unstable.

### String Operations

For any String operation, remember below points

* String contains char[] and you can use char[] chars = str.toCharArray()  
  - str.charAt(i) is very useful
* Ask interviewer whether you should support ascii/extended ascii/unicode chars.  
   ASCII chars are english numbers+letters+special chars = 128. Extended ascii chars (total 256) contains many other special chars.  
   If interviewer say ascii is good, then use aux array of size 128 (char[] chars = new char[128])
* Default value of char[] is Character.MIN\_VALUE ('\u0000')
* Remember this pattern  
    
   char[] chars = new char[128]

for (int i = 0; i < str.length(); i++) {  
 char c = str.charAt(i);  
 chars[c] = c; // or chars[c]++  
 ...  
 }

* ASCII A-Z = 65-90, a-z = 97-122. There are some special chars in between 90 and 97.

<http://www.asciitable.com/>

LinkedList Operations

Major difference between String and LinkedList is String has charArray. It is easy to iterate an indexed array compared to LinkedList. So, you need extra intelligence to travers a LinkedList.

e.g. Palindrome Algorithm of String vs LinkedList.

It is so easy to work with String. You can traverse charArray from left and right together till you come in the middle and compare the elements.

In case of LinkedList, you need to use runners and stack to achieve the same thing.

- Can you use Doubly LinkedList?

This is a question for an interviewer. To make your computation easier, you can ask an interviewer whether you can use a doubley linkedlist for solving a problem. You can also ask whether you can keep length variable in LinkedList. You can increment this variable on each insert and decrement on each deletion. This will help you not to iterate through entire linkedlist when you need find a length of it.

- LinkedList class is just a wrapper of Head node  
  
 public class LinkedList {  
 private Node head;  
   
 public Node addToTail(Node newNode) {...}  
   
 public Node addAsHead(Node newNode) {...}  
   
 public Node delete(Node node) {...}  
   
 // peek just reads the head node and returns it. It doesn't remove the head node  
 public Node peek() {...}  
   
 // pop just reads the head node, removes it and returns it.  
 public Node pop() {...}  
 }  
   
 public class Node {  
 private int data;  
 private Node next;  
   
 public Node(int data) {  
 this.data = data;  
 }  
 }  
  
 - Runner Node(s)

Use Runner to traverse through a LinkedList.

Don't do head=head.next. You will end up moving head pointer to some other node in the LinkedList.  
   
 head  
 runner  
 |  
 v  
 --------------------  
 | 1 | 2 | 3 | 4 | 5 |  
 --------------------

Whenever you need to iterate through a linked list, always create a runner node.  
Do not iterate by moving head=head.next, otherwise you will end up moving head pointer to somewhere else in the linked list.

You should do

Node runner = head;  
 while(...) runner = runner.next;  
   
   
(VERY IMP) More than one Runner Technique:  
   
The runner technique means that you iterate through the linked list with two pointers simultaneously, with one ahead of the other.  
The "fast" node might be ahead by affixed amount, or it might be hopping multiple nodes for each one node that the "slow" node iterates through.  
   
For example, suppose you had a linked list a1->a2->.....->an->b1->b2->.....->bn and you wanted to arrange it into a1->b1->a2->b2->...->an->bn. You do not need to know the length of the linked list(but you do know that the length is an even number).  
   
You could have one pointer p1(the fast pointer) move every two elements for every one move that p2 makes.  
When p1 hits the end of the linked list, p2 will be at the midpoint. Then, move p1 back to the front and begin "weaving" the elements. On each iteration, p2 selects and element and inserts it after p1.  
  
- Recursion  
  
You can write a normal Iterative traversal also, but if you want to use the recursion, then you can do following.  
  
 public Node search(int data) {  
 return search(head, data)  
 }  
 public Node search(Node runner, int data) {  
 if(runner == null || runner.data == data) return runner;  
 return(runner.next, data);  
 }

- Using extra buffer for linkedlist algorithms?

##### Using map or set as extra buffer

Many times when you traverse a linkedlist using runners (pointers), you may end up with runtime complexity O(n^2).

**(IMP) Ask interviewer, are you allowed to use extra buffer?**

If he says yes, you can use map/set as extra buffer.

**NOTE: Set internally uses Map.** So, searching anything in Set will take O(1) only.

e.g. Remove Duplicates from LinkedList algorithm (RemoveDups.java)

##### Using stack extra buffer

e.g. PalindromeLinkedList.java, ReturnKthToLastElement.java

NOTE:

In case of String’ Palindrome StringPalindrome.java, you don’t need any complexity because String provides you indexed charArray using str.toCharArray(). It’s easy to iterate indexed array compared to a LinkedList.

#### - Do Not modify an object sent as parameter

e.g. DeleteMiddleNode.java

**private static void** delete(Node head, Node nodeToBeDeleted) {  
 **if** (head == **null**) **return**;  
  
 Node R = head;  
 Node prevOfR = **null**;  
 **while** (R != **null**) {  
 **if** (R.equals(nodeToBeDeleted)) {  
 **if** (prevOfR != **null**) {  
 prevOfR.**next** = R.**next**;  
 } **else** {  
 head = R.**next**; *// This doesn't work* R.**next** = **null**;  
 }  
 **break**;  
 }  
 prevOfR = R;  
 R = R.**next**;  
  
 }  
}

In above code, you are trying to manipulate sent object (head), but you are forgetting that  
When caller calls a method, situation is like below  
  
 sent head from caller -----|  
 | -> 5  
 param head-------------------|  
  
When you modify incoming parameter, situation will be as follows:  
  
 sent head from caller --------> 5  
 param head----------------------> 2  
  
It won't change the actual ‘head’ object sent by a caller  
  
Solutions:  
  
1) Whenever you need to do that, you wrap that param with some other class and send that class object as a param.  
 e.g. SinglyLinkedList  
 private static void delete(SinglyLinkedList LL, Node nodeToBeDeleted) {  
  
 Now, when you do LL.head = 2. It will actually update the content sent LL object.  
  
2) Node delete(Node head, Node nodeToBeDeleted) {  
 ...  
 Node newHead = 2;  
 ...  
  
 return newHead;  
 }

**private static void** delete\_1(SinglyLinkedList LL, Node nodeToBeDeleted) {  
 **if** (LL.**head** == **null**) **return**;  
  
 Node R = LL.**head**;  
 Node prevOfR = **null**;  
 **while** (R != **null**) {  
 **if** (R.equals(nodeToBeDeleted)) {  
 **if** (prevOfR != **null**) {  
 prevOfR.**next** = R.**next**;  
 } **else** {  
 LL.**head** = R.**next**;  
 R.**next** = **null**;  
 }  
 **break**;  
 }  
 prevOfR = R;  
 R = R.**next**;  
  
 }  
}  
*// OR***private static** Node delete\_2(Node head, Node nodeToBeDeleted) {  
 **if** (head == **null**) **return** head;  
  
 Node R = head;  
 Node prevOfR = **null**;  
 Node newHead = head;  
 **while** (R != **null**) {  
 **if** (R.equals(nodeToBeDeleted)) {  
 **if** (prevOfR != **null**) {  
 prevOfR.**next** = R.**next**;  
 } **else** {  
 newHead = R.**next**;  
 R.**next** = **null**;  
 }  
 **break**;  
 }  
 prevOfR = R;  
 R = R.**next**;  
 }  
 **return** newHead;  
}

#### - How to check whether LinkedList has odd or even size?

1 -> 2 -> 3 -> 4 -> null

a

move runner ‘a’ two steps at a time till (a==null or a.next==null).

If(a==null) then it’s a even size.

If(a.next==null) then it’s odd size

e.g. PalindromeLinkedList.java

### Stack And Queue

Stack and Queue are created using linkedlist.

Important:

* Stack is a LinkedList where items are added and removed to/from head(top). 'head' in Stack is called 'top'.
* Queue is a LinkedList where items are added at tail and removed from head.
* Stack is useful for recursions.
* Queue is useful for BFS (Breadth First Search) and for implementing a Cache. LRUCache.java is an example of using a Queue for Caching.

class MyStack<T> {

Node<T> top;

public T pop(){…}

public T peek(){…}

public T push(T item){…}

public boolean isEmpty(){…}

}

class MyQueue<T> {

Node<T> first;

Node<T> last;

public T remove(){…}

public T peek(){…}

public T add(T item){…}

public boolean isEmpty(){…}

}

Stack is LIFO and Queue is FIFO.

LinkedList doesn't create an array to store elements. It maintains references between two nodes of elements.

Popping activity is same in both in stack and queue, first element is popped and new first element is set as old first element's next

Important thing is base class for LinkedList. If you remember Node class, then Stack and Queue algorithms are easy to create.

Why can't we use Array instead of LinkedList?

Because Array has to be declared with fixed size and if you don't know how many elements you are dealing with then it's very hard to use Array.

You can use Resizable Array instead of Array. Read document for more details.

java.util.Stack extends Vector which is based on Resizable Array

java.util.Queue has many forms BlockingQueue, ArrayBlockingQueue, LinkedBlockingQueue etc. It provides client a choice to use Fixed size Array or LinkedList.

Important Stack methods:

pop() - Removes the top item from the stack.

push(item) - Add an item to the top of the stack.

peek() - Return the top of the stack (does not remove an item like pop())

isEmpty() - Returns tru if and only if the stack is empty.

Important Queue methods:

add(item) - Add an item to the end of the list.

remove() - remove the first item in the list.

peek() - Return the top of the stack.(does not remove an item like remove())

isEmpty() - Return true if and only if the stack is empty.

### What is stable and unstable sort?

[**http://programmers.stackexchange.com/questions/247440/what-does-it-mean-for-a-sorting-algorithm-to-be-stable**](http://programmers.stackexchange.com/questions/247440/what-does-it-mean-for-a-sorting-algorithm-to-be-stable)

A stable sort is one which preserves the original order of the input set, where the comparison algorithm does not distinguish between two or more items.

Consider a sorting algorithm that sorts cards by rank, but not by suit. The stable sort will guarantee that the original order of cards having the same rank is preserved; the unstable sort will not.



From BSIS(bubble,selection,insertion,shell) and M-HQ(merge, heap,quick) sorting algorithms, HQ are unstable.

You can also think differently – In whichever Sort, swaps(exchanges) of elements happen between distance elements (not adjustant elements), they are unstable. This is explained in Coursera video of ‘6-5 Stability (5-39).mp4’.

# When to Use What

## Arrays

In many situations the array is the first kind of structure you should consider when storing

and manipulating data. Arrays are useful when

• The amount of data is reasonably small.

• The amount of data is predictable in advance.

If you have plenty of memory, you can relax the second condition; just make the array big

enough to handle any foreseeable influx of data.

If insertion speed is important, use an unordered array. If search speed is important, use

an ordered array with a binary search. Deletion is always slow in arrays because an

average of half the items must be moved to fill in the newly vacated cell. Traversal is fast

in an ordered array but not supported in an unordered array.

Vectors, such as the Vector class supplied with Java, are arrays that expand

themselves when they become too full. Vectors may work when the amount of data isn't

known in advance. However, there may periodically be a significant pause while they

enlarge themselves by copying the old data into the new space.

## Linked lists

Consider a linked list whenever the amount of data to be stored cannot be predicted in

advance or when data will frequently be inserted and deleted. The linked list obtains

whatever storage it needs as new items are added, so it can expand to fill all of available

memory; and there is no need to fill "holes" during deletion, as there is in arrays.

Insertion is fast in an unordered list. Searching and deletion are slow (although deletion is

faster than in an array), so, like arrays, linked lists are best used when the amount of data

is comparatively small.

A linked list is somewhat more complicated to program than an array, but is simple

compared with a tree or hash table

## Binary Search Trees

A binary tree is the first structure to consider when arrays and linked lists prove too slow.

A tree provides fast O(logN) insertion, searching, and deletion. Traversal is O(N), which

is the maximum for any data structure (by definition, you must visit every item). You can

also find the minimum and maximum quickly, and traverse a range of items.

An unbalanced binary tree is much easier to program than a balanced tree, but

unfortunately, ordered data can reduce its performance to O(N) time, no better than a

linked list. However, if you're sure the data will arrive in random order, there's no point

using a balanced tree.

## Balanced Trees

Of the various kinds of balanced trees, we discussed red-black trees and 2-3-4 trees.

They are both balanced trees, and thus guarantee O(logN) performance whether the

input data is ordered or not. However, these balanced trees are challenging to program,

with the red-black tree being the more difficult. They also impose additional memory

overhead, which may or may not be significant.

The problem of complex programming may be reduced if a commercial class can be

used for a tree. In some cases a hash table may be a better choice than a balanced tree.

Hash-table performance doesn't degrade when the data is ordered.

There are other kinds of balanced trees, including AVL trees, splay trees, 2-3 trees, and

so on, but they are not as commonly used as the red-black tree.

## Hash Tables

Hash tables are the fastest data storage structure. This makes them a necessity for

situations in which a computer program, rather than a human, is interacting with the data.

Hash tables are typically used in spelling checkers and as symbol tables in computer

language compilers, where a program must check thousands of words or symbols in a

fraction of a second.

Hash tables may also be useful when a person, as opposed to a computer, initiates dataaccess

operations. As noted above, hash tables are not sensitive to the order in which

data is inserted, and so can take the place of a balanced tree. Programming is much

simpler than for balanced trees.

Hash tables require additional memory, especially for open addressing. Also, the amount

of data to be stored must be known fairly accurately in advance, because an array is

used as the underlying structure.

A hash table with separate chaining is the most robust implementation unless the amount

of data is known accurately in advance, in which case open addressing offers simpler

programming because no linked list class is required.

Hash tables don't support any kind of ordered traversal or access to the minimum or

maximum items. If these capabilities are important, the binary search tree is a better

choice..

