Data Structures and Algorithms Assignment 7

1. What is the distinction between a list and an array?

- A **list** is a data structure that's built into Python and holds a collection of items.
- List items are enclosed in square brackets.
- Lists are **ordered** – i.e. the items in the list appear in a specific order. This enables us to use an index to access to any item.
- Lists are **mutable**, which means you can add or remove items after a list's creation.
- List elements do not need to be **unique** and item duplication is possible.
- Elements can be of different data types.

- An **array** is also a data structure that stores a collection of items. Like lists, arrays are **ordered**, **mutable**, enclosed in **square brackets**, and able to store **non-unique** items.
- But when it comes to the array's ability to store different data types, the answer is not as straightforward. It depends on the kind of array used.

- **Arrays need to be declared. Lists don't**.
- **Arrays can store data very compactly** and are more efficient for storing large amounts of data.
- **Arrays are great for numerical operations**; lists cannot directly handle math operations.

2. What are the qualities of a binary tree?

- A binary tree is a finite set of nodes that is either empty or consist a root node and two disjoint binary trees called the left subtree and the right subtree**.**
- In other words, **a binary tree is a non-linear data structure in which each node has maximum of two child nodes. The tree connections can be called as branches**.
- Trees are used to represent data in hierarchical form.
- Binary tree is the one in which each node has maximum of two child- node.
- The order of binary tree is '2'.
- Binary tree does not allow duplicate values.
- While constructing a binary, if an element is less than the value of its parent node, it is placed on the left side of it otherwise right side.

3. What is the best way to combine two balanced binary search trees?

1. Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.
2. Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.
3. The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.
4. A balanced tree from the merged array. This step takes O(m+n) time.
5. Time complexity of this method is O(m+n) which is better than other methods. This method takes O(m+n) time even if the input BSTs are not balanced.

4. How would you describe Heap in detail?

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

  key(α) ≥ key(β)

- As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types –

  1. **Min-Heap** − Where the value of the root node is less than or equal to either of its children.

  2. **Max-Heap** − Where the value of the root node is greater than or equal to either of its children.

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

- **Note** − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

5. In terms of data structure, what is a HashMap?

- **Hashmap** is hash table or scatter table. It is a special form of a data table with a special index structure. With the help of hashmaps, data objects can be quickly searched and found in large amounts of data.

- A mathematical hash function calculates the position of the data object in the table. The data is stored as key-value pairs. The values and their position can be identified and retrieved via the key. There is no need to search through many data objects. The special feature of hash maps is that the time required for searching and finding remains constant compared to other index structures such as tree structures and is independent of the table size.

6. How do you explain the complexities of time and space?

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.

Similarly, Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Lets start with a simple example. Suppose you are given an array A and an integer x and you have to find if x exists in array A.
Simple solution to this problem is traverse the whole array A and check if the any element is equal to x.

```
for i : 1 to length of A
   if A[i] is equal to x
      return TRUE
return FALSE
```

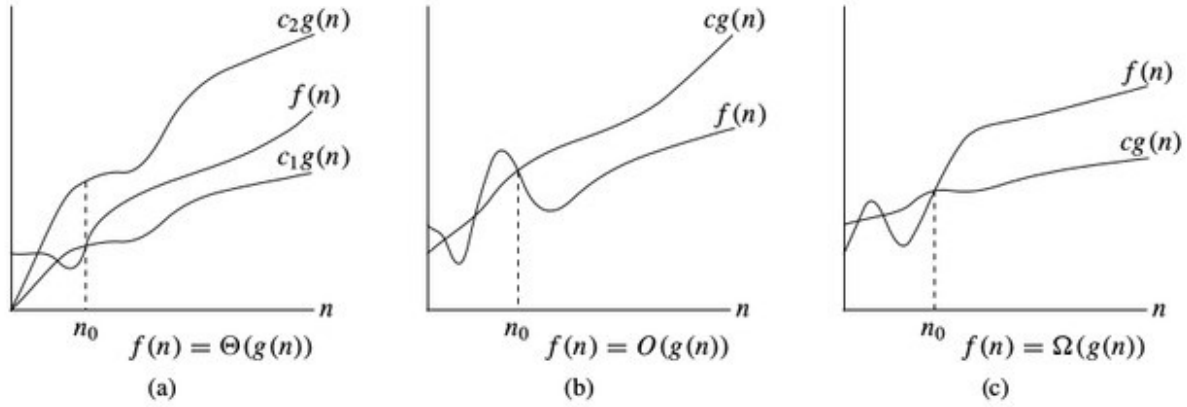
Each of the operation in computer take approximately constant time. Let each operation takes c time. The number of lines of code executed is actually depends on the value of x. During analyses of algorithm, mostly we will consider worst case scenario, i.e., when x is not present in the array A. In the worst case, the **if** condition will run N times where N is the length of the array A. So in the worst case, total execution time will be (N∗c+c). N∗c for the **if** condition and c for the **return** statement (ignoring some operations like assignment of i).
As we can see that the total time depends on the length of the array A. If the length of the array will increase the time of execution will also increase.

Order of growth is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

O**-notation:**
To denote asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n)) (pronounced “big-oh of g of n”) the set of functions:

O(g(n))= { f(n) : there exist positive constants c and n0 such that 0≤f(n)≤c∗g(n) for all n≥n0 }
Ω**-notation:**
To denote asymptotic lower bound, we use Ω-notation. For a given function g(n), we denote by Ω(g(n)) (pronounced “big-omega of g of n”) the set of functions:
Ω(g(n))= { f(n) : there exist positive constants c and n0 such that 0≤c∗g(n)≤f(n) for all n≥n0 }
Θ**-notation:**
To denote asymptotic tight bound, we use Θ-notation. For a given function g(n), we denote

by Θ(g(n)) (pronounced “big-theta of g of n”) the set of functions:
Θ(g(n))= { f(n) : there exist positive constants c1,c2 and n0 such
that 0≤c1∗g(n)≤f(n)≤c2∗g(n) for all n>n0 }

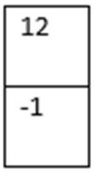


7. How do you recursively sort a stack?

1. First Iteration keeps on calling the recursive function to pop out the top element until the stack is empty.
    - Top element = 6 (Top in the stack frame #1)
    - Top element = -3 (Top in the stack frame #2)
    - Top element = 23 (Top in the stack frame #3)
    - Top element = 12 (Top in the stack frame #4)
    - Top element = -1 (Top in the stack frame #5)

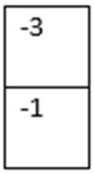
2. Now the program gets to the stack frame of the final element i.e. -1 and correspondingly calls the sorted insert function. The stack becomes,

3. It gets to the next stack frame i.e. #4 and the current element now is 12, stack top is -1, as 12 > -1, 12 will directly be inserted in the stack.

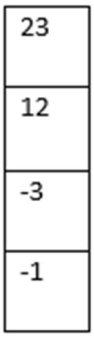


4. For the next iteration current element is 23, stack top is 12 hence push it again directly.

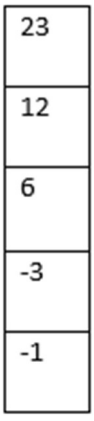
5. Now current element is -2 and the stack top is 23 and -2 < 23 hence the new top will be stored and it will recursively call the sortedinsert function with the current -2 until it gets to an element where the stack top is smaller than -2 or the stack becomes empty and then push it.



6. Push back all the elements in top after the previous element has been placed in its right place.



7. The above step gets repeated for the next element also and the final stack becomes.



We get the final sorted stack