Data Structures and Algorithms Assignment 2

1. **Use examples to explain the sorting algorithms.**

   A sorting algorithm is used to arrange elements of an array/list in a specific order.

   For simplicity, consider the data in the range 0 to 9.
   Input data: 1, 4, 1, 2, 7, 5, 2
   1) Take a count array to store the count of each unique object.
   Index: 0 1 2 3 4 5 6 7 8 9
   Count: 0 2 2 0 1 1 0 1 0 0

   2) Modify the count array such that each element at each index stores the sum of previous counts.
   Index: 0 1 2 3 4 5 6 7 8 9
   Count: 0 2 4 4 5 6 6 7 7 7

   The modified count array indicates the position of each object in the output sequence.

   3) Output each object from the input sequence followed by decreasing its count by 1.
   Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.
   Put data 1 at index 2 in output. Decrease count by 1 to place next data 1 at an index 1 smaller than this index.

2. **What Are the Benefits of Stacks?**

   - Efficient data management: Stack helps you manage the data in a LIFO (last in, first out) method, which is not possible with a Linked list and array.
   - Efficient management of functions: When a function is called, the local variables are stored in a stack, and it is automatically destroyed once returned.
   - Control over memory: Stack allows you to control how memory is allocated and deallocated.
   - Smart memory management: Stack automatically cleans up the object.
   - Not easily corrupted: Stack does not get corrupted easily; hence it is more secure and reliable.
   - Does not allow resizing of variables: Variables cannot be resized.

3. What is the difference between a stack and a queue?

Stacks	Queues
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
Stack is used in solving problems works on recursion.	Queue is used in solving problems having sequential processing.

4. What are the different forms of queues?

Simple Queue
In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.

Circular Queue
In a circular queue, the last element points to the first element making a circular link.

Priority Queue
A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Deque (Double Ended Queue)
In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.

5. Why should I use Stack or Queue data structures instead of Arrays or Lists, and when should I use them?

- Use a queue when you want to get things out in the order that you put them in.
- Use a stack when you want to get things out in the reverse order than you put them in.
- Use a list when you want to get anything out, regardless of when you put them in (and when you don't want them to automatically be removed).

6. What is the significance of Stack being a recursive data structure?

When you call into a function, your local variables are saved on a stack. When you return, they are restored to the saved values. But that means that in an ordinary case, each layer of the stack looks different.

In the recursive case, the function you are calling into is the same function you are in. So layers of the stack looks the same.

You can use this fact to emulate the function calls with a stack data structure made up if an array of objects that contain the same fields as your local variables, and an index into that array.

Start the index at zero.

At the point where you would recurse, you copy your locals into the record at that index and increment it. Then loop back to the top of the function — because that is where control would go if you called in.

At the point you would return, decrement the index and copy the indexed record back into the locals. The flow of control will already be at the point it would return to after the call finished.

So if you are working in a language that supports recursion poorly, you can optimize this to be faster than actual recursion. The return position is known, and does not need to be saved, and the regular structure of the array of saved values can sometimes to optimized. You can see the available optimization alternatives because you create the structure and have complete visibility and control.