DSA Assignment 1

-Radhika Maheshwari
IIT Roorkee
radhika.maheshwari@desisascendeducare.in

Link to Github Repository : https://github.com/r4dhika/DSA_Assignment_1
Ans 1:
Linked list implementation : I have implemented the linked list using a class Node, having attributes value, next and prev. The linked list is a doubly linked list as it stores the pointers to the nodes next and previous to the current node. The functions implemented are:

1. ***insert_at_start(int num)***: a new node is inserted before the head, and then this new node is assigned as head pointer. The time and space complexity is O(1).
2. ***insert_at_end(int num)***: stored the tail pointer to prevent traversal, thus a new node is added after the tail and this new node is assigned as the tail pointer. The time and complexity is O(1). {*note that if i had used a singly linked list, then time complexity would be O(n) due to traversal.*}
3. ***remove_from_start()***: The head is moved to head->next, and the memory allocated to the previous head is freed. Time complexity and space is O(1).
4. ***remove_from_end()****:* The tail is moved to tail->prev, and the memory allocated to the previous tail is freed. Time and Space complexity is O(1).
5. ***front()***: returns the value stored in the head of the linked list. Time and space complexity is O(1).
6. ***last()***: returns the value stored in the tail of the linked list. Time and Space complexity is O(1).
7. ***Traversal()***: prints all the elements present in the linked list from head to tail. Time complexity is O(n) where n is the number of nodes, and space complexity is O(1).

MaxHeap implementation: I have implemented MaxHeap using a vector<int> heap, and also simultaneously stored the size of this vector. Following are the functions implemented for MaxHeap:

1. ***add(int num)***: The number is initially added to the end of the vector heap, then using Heapify_up() function, the heapness is maintained. The time complexity of adding num is O(log n). {*description in heapify_up() function*).
2. ***max()***: MaxHeap is maintained such that the root, i.e. the max element of the heap is stored at index 0 of the vector heap, thus this function returns heap[0]. Time complexity is O(1).
3. ***remove_maxt()***: First max element is stored in a variable to return value at the end, then the value of heap[0] is set to the value of the last element of heap and then heapify_down() is used to maintain heapness. The time complexity is O(log n).
4. ***heapify_down(int index)***: For any element at a certain index entered as parameter, it is placed at its right position by reorganizing the heap such that the element is pushed downwards. As the max number of swaps can be equal to the height of the heap, thus the complexity of heapify_down() will be O(log n).

5. ***heapify_up(int index)***: For any element at a certain index entered as a parameter, it is placed at its right position by reorganizing the heap such that the element is pushed upwards. As the max number of swaps can be equal to the height of the heap, thus the complexity of heapify_up() will be O(log n).

Dequeue Implementation: This is done using the above created linked list as specified in the question. All the functions have time and space complexity **O(1)**.

Ans 2.
To find the Kth largest element, I used the count sort method, thus the complexity will be O(m) where m is the largest element. As the value of each element goes up to 10^4 but the value of the number of elements goes up to 10^5, thus the function of time complexity is upper bound by n. Therefore, we can claim that the time complexity of the algorithm is **O(n)** as demanded in the question.

Ans 3.
I have first calculated all possible values of the sum of 2 elements of the given array. Then I used the lower_bound function of vector and other manipulations to check the closest sum achievable. This sum is stored in the variable achieved. The algorithm follows a greedy approach where the value of the variable achieved is updated upon better outcome.
The calculation of the sum of 2 elements had complexity = O(n^2).
Simultaneously, searching the third element using lower_bound() has complexity = O(log n).
Thus the overall complexity of the algorithm is **O((n^2)*(log n))**.

Ans 4.
I create a new vector<int> s(n), which is defined such that:
s[i] = maximum possible further jump that can be made from index i if only elements up to index i are considered.
Thus we get, `s[i] = max(s[i-1]-1, v[i]);`
Thus on iterating over the complete vector v, if there is any location in the middle(at index < n-1 for zero based indexing) where further the jump can not me made i.e. where the value of s[i] = 0; the jump to index n is not possible, thus return false, otherwise return true;
This algorithm requires only one for loop iterating over the vector<int> v(n), thus the complexity of this algorithm is **O(n)**.

Ans 5.
I have implemented the front middle back queue using the deque data structure of the Standard template library. The front middle back queue is made using two deques, q1, and q2. It is maintained using certain constraints that the number of elements in q1 and q2 are either equal or (q2)-(q1)=1; thus the middle element is easily pushed, popped and retrieved in O(1) time complexity.
All the functions in code require **O(1)** time.