Q.1

**BIG O NOTATION**

The Big O notation, where O stands for 'order of', is concerned with what happens for very large values of n. For example, if a sorting algorithm performs n2 operations to sort just n elements, then that algorithm would be described as an o(n2) algorithm.
If f(n) and g(n) are the functions defined on a positive integer number n, then
f(n) = O(g(n))
That is, f of n is Big-0 of g of n if and only if positive constants c and n exist, such that f(n) ≤cg(n). It means that for large amounts of data, f(n) will grow no more than a constant factor than g(n). We have seen that the Big O notation provides a strict upper bound for f(n). This means that the function f(n) can do better but not worse than the specified value. Big O notation is simply written as f(n) E o(g(n)) or as f(n) = O(g(n)).
Here, n is the problem size and o(g(n)) = {h(n):
positive constants c, n, such that 0 ≤ h (n) ≤ cg(n), Vn 2 no). Hence, we can say that o(g(n)) comprises a set of all the functions h(n) that are less than or equal to cg(n) for all values of n > no.
If f(n) ≤ cg(n), c>0, V n ≥ng, then f(n) = 0(g(n)) and g(n) is an asymptotically tight upper bound for f(n).
Examples of functions in o(n3) include: n2o, n3, n3 +n, 540n3 + 10.

Best case O describes an upper bound for all combinations of input. It is possibly lower than the worst case. For example, when sorting an array the best case is when the array is already correctly sorted.

Worst case O describes a lower bound for worst case input combinations. It is possibly greater than the best case.

**OMEGA NOTATION (Ω)**

The Omega notation provides a tight lower bound for f(n). This means that the function can never do better than the specified value but it may do worse.
notation is simply written as, f(n) E (g(n)), where n is the problem size and (g(n)) (h(n): positive constants c>0, n, such that 0≤ cg(n) ≤h(n), nn).
Hence, we can say that (g(n)) comprises a set of all the functions h(n) that are greater than or equal to cg(n) for all values of n 2 n
If cg(n) ≤ f(n), c>0, Vn 2 no, then f(n) En(g(n)) and g(n) is an asymptotically tight lower bound for f(n).
Examples of functions in (n2) include: n2, n2o, n3 + n2, n3
To summarize,

Best case describes a lower bound for all combinations of input. This implies that the function can never get any better than the specified value. For example, when sorting an array the best case is when the array is already correctly sorted.

- Worst case describes a lower bound for worst case input combinations. It is possibly greater than best case. For example, when sorting an array the worst case is when the array is sorted in reverse order.
- If we simply write , it means same as best case N.

**THETA NOTATION (→)**

Theta notation provides an asymptotically tight bound for f(n). ℗ notation is simply written as, f(n) (g(n)), where n is the problem size and
(g(n)) = {h(n): 3 positive constants c,, c,, and n, such that o sc,g(n) ≤ h(n) ≤ c,g(n), nn).
Hence, we can say that (g(n)) comprises a set of all the functions h(n) that are between c,g(n) and c2g(n) for all values of n ≥ no.
If f(n) is between c,g(n) and cg(n),
n≥n, then f(n) E (g(n)) and g(n) is an asymptotically
tight bound for f(n) and f(n) is amongst h(n) in the set.
To summarize,

- The best case in C notation is not used.
- Worst case → describes asymptotic bounds for worst case combination of input values.
- If we simply write C, it means same as worst case C.

Q.2 i) since the ‘i’ or 2.a’ is not present
(We can have multiple logics here for Pseudocode, and its not present in the book )

Part 1 of the answer

```
Step 1: IF START = NULL
        Write "UNDERFLOW"
        Go to Step 10
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 to 7 while PTR != NULL

Step 4: SET PREPTR = PTR
```

```
Step 5: SET TEMP = PTR -> NEXT
Step 6: Repeat while TEMP != NULL
        IF TEMP -> DATA = PTR -> DATA
                SET PREPTR -> NEXT = TEMP -> NEXT
                FREE TEMP
                SET TEMP = PREPTR -> NEXT
        ELSE
                SET PREPTR = TEMP
                SET TEMP = TEMP -> NEXT
        [END OF IF]
[END OF LOOP]
Step 7: SET PTR = PTR -> NEXT
[END OF LOOP]

Step 8: Go to Sorting Process
```

Option 1:

```
public void RemoveDuplicates(Node<T> head)
{
   // Iterate through the list
   Node<T> iter = head;
   while(iter != null)
   {
      // Iterate to the remaining nodes in the list
      Node<T> current = iter;
      while(current!= null && current.Next != null)
      {
         if(iter.Value == current.Next.Value)
         {
            current.Next = current.Next.Next;
         }
         current = current.Next;
      }
      iter = iter.Next;
   }
}
```

OR

Option 2:

```
void removeDuplicates(struct Node* head) {
   struct Node *current, *prev, *temp;
   current = head;
```

```
    while (current != NULL && current->next != NULL) {
        prev = current;
        temp = current->next;

        while (temp != NULL) {
            if (current->data == temp->data) {
                prev->next = temp->next;
                free(temp);
                temp = prev->next;
            } else {
                prev = temp;
                temp = temp->next;
            }
        }
        current = current->next;
    }
}
```

Part 2 of answer

```
Step 9: IF START = NULL OR START -> NEXT = NULL
        Return
[END OF IF]
Step 10: Repeat Steps 11 to 13 while PTR != NULL
Step 11: SET INDEX = PTR -> NEXT
Step 12: Repeat while INDEX != NULL
        IF PTR -> DATA > INDEX -> DATA
                SWAP PTR -> DATA and INDEX -> DATA
        SET INDEX = INDEX -> NEXT
[END OF LOOP]
Step 13: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 14: EXIT
```

Along w code

Sort krneke bhi multiple logics ho sakte hai :

```
struct node *sort_list(struct node *start)
{
        struct node *ptr1, *ptr2;
        int temp;
        ptr1 = start;
        while(ptr1 -> next != NULL)
```

```
        {
              ptr2 = ptr1 -> next;
              while(ptr2 != NULL)
              {
                     if(ptr1 -> data > ptr2 -> data)
                     {
                            temp = ptr1 -> data;
                            ptr1 -> data = ptr2 -> data;
                            ptr2 -> data = temp;
                     }
                     ptr2 = ptr2 -> next;
              }
              ptr1 = ptr1 -> next;
        }
          return start;  // Had to be added
}

void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Q.2 (ii)

Part 1 of answer

Step 1: IF AVAIL = NULL
          Write OVERFLOW
          Go to Step 11
[END OF IF]
Step 2: SET NEW NODE
          AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7: PTR = PTR -> NEXT
[END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START

Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT

Along w code

```
struct node *insert_beg(struct node *start)
{
	struct node *new_node, *ptr;
	int num;
	printf("\n Enter the data : ");
	scanf("%d", &num);
new_node = (struct node *)malloc(sizeof(struct node)); new_node -> data = num;
ptr = start;
while(ptr -> next != start)
ptr = ptr -> next; ptr -> next = new_node;
new_node -> next = start; start = new_node; return start;
}
```

Part 2 of answer

Step 1: IF START = NULL
    Write UNDERFLOW
    Go to Step 8
    [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR-> NEXT = START
Step 7: FREE PTR
Step 8: EXIT

Along w code

```
struct node *delete_end(struct node *start)
{
struct node *ptr, *preptr; ptr = start;
while(ptr -> next != start) {
preptr = ptr; ptr = ptr -> next;
}
preptr -> next = ptr -> next; free(ptr);
return start;
}
```

Q.3 (ii)
Step 1: Start
Step 2: Input the number N
Step 3: Initialize an empty stack
Step 4: Repeat Steps 5 and 6 while N > 0
Step 5: Extract the last digit of N using digit = N % 10
Step 6: Push digit onto the stack and update N = N / 10
[END OF LOOP]
Step 7: Initialize reversedNum = 0 and place = 1
Step 8: Repeat Steps 9 and 10 while the stack is not empty
Step 9: Pop the top element from the stack
Step 10: Update reversedNum = reversedNum + (popped_digit * place)
Step 11: Multiply place = place * 10
[END OF LOOP]
Step 12: Store reversedNum as the final reversed number
Step 13: Print reversedNum
Step 14: End

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int stk[MAX], top = -1;
void push(int val) {
   stk[++top] = val;
}
int pop() {
   return stk[top--];
}
int main() {
   int num, val, i, reversedNum = 0, place = 1;
   printf("\n Enter a number: ");
   scanf("%d", &num);
   while (num > 0) {
      push(num % 10);
      num /= 10;
   }
   while (top != -1) {
      val = pop();
      reversedNum = reversedNum * 10 + val;
   }
   printf("\n Reversed Number: %d\n", reversedNum);
   return 0;
}
```