Second question
(2.a)

```
Reverse_func ( list * my_list) {
        list  prev  // pointing to prev node
        list  Next  // pointing to next node
        list  current //pointing to current node

        prev = NULL   // initiation      T(1)
        current = my_list
        next = current -> next           T(1)

        while (1) {                      T(n)
                current -> next = prev          T(n-1)
                prev = current                  T(n-1)
                current = next                  T(n-1)
                if (Next -> next == NULL)      T(n-1)
                        current -> next = prev  T(n-k)
                        break
                else
                        Next = Next -> next     T(k)


        return current T(1)

end_func
```

While we have only one while loop that depends on the number of elements in the list, the time complexity of the function is T (n).

- The function is in-situ function, since the function doesn't move the element from its memory location, rather than this it changes the connection between the elements while reversing.

2.b)

```
void blendroot(Tree* root)
{

    if (root == NULL || root->left == NULL && root->right == NULL) {
        return;
    }

    if (root->left != NULL) {    T(1)

        blendroot(root->left);                    // T(lg n)

        Tree* tmp = root->right;
        root->right = root->left;
        root->left = NULL;
        Tree* Tre = root->right;
        while (Tre->right != NULL) {            // T(k) ; independent
of the number of elements in the array
            Tre = Tre->right;
        }
          Tre->right = tmp;
    }
}
    void move_list(Tree* T,List* A ){
   if (T == NULL)
        return ;
    move_list(T->right,A);        //T(lgn)
    A->push(T->data);
    move_list(T->left,A);       //T(lgn)
     }
```

In the previous code, I used two functions. One to blend the root in the list, and the other one is to move the elements to a list structure in order.

Summing up, the asymptotic time complexity tends to be of T(lg n)

(2.c)

```
Tree* sortRecursion(List **my_list, int n)
{

    if (n <= 0)
        return NULL;

    Tree *left = sortRecursion(my_list, n/2);
    Tree *root = new Tree();
    root->data = (*my_list)->data;
    root->left = left;
    (*my_list) = (*my_list)->next;
    root->right = sortRecursion(my_list, n - n /2-1 );
    return root;
}
```

Looking at the recursion in the previous code:

Using master method T(n) = 2 T(n/2) + T(1); // neglecting the constant addition

Θ (T(n)) =Θ (n^ lg 2)

While f(n) = Θ (1)

Which means Θ (f(n)) =Θ (n^ lg 2-1)

Then the function belongs to Ω(n) , the lower bound of Θ (T(n))

Searching complexity in binary trees consumes far less time than the general used methods, since it depends on the height of the tree not the number of elements, which means it has time complexity of Θ

(h), while the other methods require the compiler to access every single node (element), which means they are of time complexity Θ (n).

Resources :

- Slides
- https://www.google.com/search?client=firefox-b-d&q=stack+properties
- https://www.youtube.com/watch?v=dGJtKxBpP00