Time complexity

i) Binary search

Code:

In binary search we must find out if a number exists in a sorted array of a given length which must be inputed by the user. How we will be implementing this is by defining three variables; beginning, middle, and end. Now if the number to be searched is greater than the middle number of the array, we will update the beginning to now be the middle number. If it is smaller than the middle number we will update the end to be the middle number. This way we have halved the given array. We will implement this loop again and again until the final array has only our number to be searched left.

```
#include<iostream.h>
#include<conio.h>

void main()
{

  int arr[100],beg,mid,end,i,n,num;

  cout << "\n Enter the size of an array ";
  cin >> n;

  cout << "\n Enter the values in sorted order (asc or desc) \n";

  for(i = 0; i < n;i++)
  {
     cin >> arr[i];
  }


  beg = 0;
  end = n-1;

  cout << "\n Enter a value to be searched in an array ";
  cin >> num;

  /* Run loop, while beg is less than end. */

  while( beg <= end)
  {

     mid = (beg+end)/2;

    /* If value is found at mid index,
       the print the position and exit. */

     if(arr[mid] == num)
     {
       cout << "\nItem found at position "<< (mid+1);

        exit(0);

     } else if(num > arr[mid]) {

       beg=mid+1;
     } else if (num < arr[mid]) {

        end=mid-1;

     }

  }

   cout << "Number does not found.";
}
```

Here we want to divide an array of N elements until we have only one element left so N(1/2)^k = 1
Applying log of base 2, log base2 N = k. Time complexity O (log base2 N)

ii) Bubble Sort

In this algorithm we swap adjacent numbers to sort a given series of numbers into ascending order. We do this by swapping two numbers adjacent to each other if and only if the number to the right is less than the number to the left. This way at the end of one iteration the greatest number is placed at the right most end. And with every subsequent iteration, another number from the right gets fixed.

```
#include <stdio.h>

void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
```

This is how a basic bubble sort programme looks like.
Please note here we have applied the condition j< n-i-1 because we want to freeze the right most elements after every iteration.
Now for i=n-1 the j loop will run 0 times
For i= n-2 the j loop will run 1 time
For i=n-3 it will run 2 times and so on till i=0 and j loop runs n-1 times

Here the the if statement used to swap elements won’t really make a difference because assuming worst case scenario it will do three actions every time to swap the numbers and can be treated as a variable independent of n

So total number of times the loop will run is 0+1+2+…+n-1 = (n-1)*n/2 therefore highest order of n is 2 and time complexity is O (n^2)

iii) Recursive Factorial algorithm

In this type of algorithm we usually have a function that works along this basic template

f(n)= n*f(n-1)

So for eg f(1)=1 and we need to find out f(n)

f(2)= 2*f(1)
f(3)=3*f(2)= 3*2*f(1)
f(4)= 4*f(3)= 4*3*2*f(1)
.
.
f(n)= n*(n-1)*(n-2)….2*1*f(1)

```
factorial(n):
   if n is 0
       return 1
   return n * factorial(n-1)
```

This will translate to something like this

```
n*factorial(n-1)
   *(n-1)factorial(n-2)
         *(n-2)factorial(n-3)
              …………
                      1*factorial(0)
```

Therefore we can say that it has n steps until the function is executed
Therefore the time complexity is O (n)

Sudoku problem

*disclaimer: I tried this code on my own but in the end had to resort for help on how to check for validity of the 3x3 grids :/

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;
#define UNASSIGNED 0
#define N 9

bool UnassignedLocation(int grid[N][N], int &row, int &col);
bool isSafe(int grid[N][N], int row, int col, int num);

/*Here we will assign values to all unassigned locations for Sudoku solution
 */
bool Sudoku(int grid[N][N])
{
    int row, col;
    if (!UnassignedLocation(grid, row, col))
       return true;
    for (int num = 1; num <= 9; num++)
    {
    if (isSafe(grid, row, col, num))
    {
        grid[row][col] = num;
        if (Sudoku(grid))
            return true;
        grid[row][col] = UNASSIGNED;
    }
    }
    return false;
}

/* We need to search the grid to find an entry that is still unassigned. */
bool UnassignedLocation(int grid[N][N], int &row, int &col)
{
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (grid[row][col] == UNASSIGNED)
                return true;
    return false;
}

/* This should return whether any assigned entry n the specified row matches
   the given number. */
bool UsedInRow(int grid[N][N], int row, int num)
{
    for (int col = 0; col < N; col++)
    if (grid[row][col] == num)
        return true;
    return false;
}

/* And this should returns whether any assigned entry in the specified column matches
   the given number. */
bool UsedInCol(int grid[N][N], int col, int num)
{
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
            return true;
    return false;
}
```

```
/* This will return whether any assigned entry within the specified 3x3 box matches
   the given number. */
bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num)
{
   for (int row = 0; row < 3; row++)
      for (int col = 0; col < 3; col++)
         if (grid[row+boxStartRow][col+boxStartCol] == num)
            return true;
   return false;
}

/* This returns whether it will be legal to assign num to the given row,col location.
 */
bool Valid(int grid[N][N], int row, int col, int num)
{
   return !UsedInRow(grid, row, num) && !UsedInCol(grid, col, num) &&
     !UsedInBox(grid, row - row % 3 , col - col % 3, num);
}

void printGrid(int grid[N][N])
{
   for (int row = 0; row < N; row++)
   {
      for (int col = 0; col < N; col++)
         cout<<grid[row][col]<<"  ";
      cout<<endl;
   }
}

int main()
{
   int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
                     {5, 2, 0, 0, 0, 0, 0, 0, 0},
                     {0, 8, 7, 0, 0, 0, 0, 3, 1},
                     {0, 0, 3, 0, 1, 0, 0, 8, 0},
                     {9, 0, 0, 8, 6, 3, 0, 0, 5},
                     {0, 5, 0, 0, 9, 0, 6, 0, 0},
                     {1, 3, 0, 0, 0, 0, 2, 5, 0},
                     {0, 0, 0, 0, 0, 0, 0, 7, 4},
                     {0, 0, 5, 2, 0, 6, 3, 0, 0}};
   if (Sudoku(grid) == true)
      printGrid(grid);
   else
      cout<<"No solution exists"<<endl;
   return 0;
}
```