Q-1) Count the number of digits in the given number?

```
void count_digits_1(int num){
   int ct=0;
   while(num){
      num/=10;
      ct++;
   }
   cout<<ct<<endl;
}
```

We divide the number by 10 and simultaneously increment the counter until the number becomes zero. The final value in the counter will have the number of digits in the original number.

Complexity :

Time : O(n) where "n" is number of digits

Space: O(1)

```
void count_digits_2(int num){
   string n = to_string(num);
   cout<<n.length();
}
```

In the above mentioned approach, we convert the number to string and then use inbuilt .length() function to calculate the length of the string which in this case would be the total number of digits in the number.

Complexity :

Time : O(1)

Space: O(1)

```
void count_digits_3(int num){
   cout<<floor(log10f(num))+1<<endl;
}
```

Log of any number "n" to the base "x" denotes how many times x multiply to itself to become n. For example Log(100) to base 10 will be 2 as 10^2 = 100. Similarly log(1000) to base 10 will be 3. This means log of any number between 100 - 1000 to base 10 will be between(2,3) and the floor of that value + 1 will be 3 which is the number of digits in each of all the numbers from 100 to 999.

Complexity :

Time : O(1)

Space: O(1)

Q-2) Reverse the given number.

```
void reverse(int x)
{
   long long int rev = 0;
   while(x){
      rev = rev*10 + x%10;
      x/=10;
   }
   // if rev exceeds the signed 32-bit integer range [-(2^31), 2^31 - 1] then it
will have a garbage value
   // hence we use the below code to return 0 in that case
   cout<<(rev<pow(2,31)-1 && rev>-pow(2,31))*rev<<endl;
   //cout<<(rev<INT_MAX && rev>INT_MIN)*rev<<endl;
   // INT_MAX and INT_MIN can be used instead of pow(2,31)-1 and -
pow(2,31)

   /*
   Complexity
   time: O(n) where "n" is number of digits
   space: O(1)
   */

}
```

We take the last digit of number and store it in the variable "rev", then we multiply the rev with 10 and add the 2nd last digit to it and store the result in rev. This loop goes on until the number becomes 0.

Complexity:

Time : O(n) where "n" is number of digits

Space: O(1)

Q-3) Check if given number is palindrome or not?

```
string check_palindrome(int x)
{
   if (x<0) return "Not a Palindrome";
   long long int rev = 0, num=x;
   while(x>rev){
      rev = rev*10 + x%10;
      x/=10;
   }
   // the above while loop reverses only the last half of the integer
   if(x==rev || x==rev/10) // we use 2nd condition in case of odd num of
digits
      return "Palindrome";
   else
      return "Not a Palindrome";
   // works only for x in the signed 32-bit integer range [-(2^31), 2^31 - 1]
   /*
   Complexity
   time: O(n/2) where "n" is number of digits
   space: O(1)
   */
}
```

Brute force approach : Reverse whole number and check if it equals the original number or not.

Optimize approach: A number is considered as a palindrome if it reads the same forwards and backwards. Hence if the reverse of the 2nd half of a number equals the 1st half then the number is palindrome. Hence we run the loop and at each iteration we keep extracting the last digit of number and at the same time form the reverse of 2nd half: Refer (Ques -2 of Basic Maths section). When the current value in "rev" becomes >= current value of x, we break the loop. Next if there were even number of digits in the original number, then x must be equal to rev else x = rev/10 for a number to be palindrome.

Complexity:

Time: O(n/2) where "n" is number of digits

Space: O(1)

Q-4) Calculate the GCD of given two numbers.

```
void gcd_brute_force(int n, int m){
   if(n<m){
      n = n + m;
      m = n - m;
      n = n - m;
   }
   int gcd = m;
   while(gcd>0){
      if(n%gcd==0 && m%gcd==0){
         cout<<gcd<<endl;
         break;
      }
      gcd--;
   }
   /*
   we all know that the GCD value of two numbers cannot exceed the smallest of the two numbers
   hence we intiate the gcd from smallest number and decrement it while we get a number that divides both the number
   complexity
   worst case that is for gcd(23,21), it will be O(m) where m is the smallest number

   Complexity
   time: O(min(n,m)) where is number of digits
   space: O(1)
   */
}
```

we all know that the GCD value of two numbers cannot exceed the smallest of the two numbers hence we intiate the gcd from smallest number and decrement it while we get a number that divides both the number.

Complexity:

Time: O(min(n,m))

Space: O(1)

```
void gcd_optimized(int n, int m){
   n+=m;
   do{
      n-=m;
      if(n<m){
```

```
        n = n + m;
        m = n - m;
        n = n - m;
        }
    }while(n%m);
    cout<<m<<endl;
}
```

1) find the smallest of two numbers
2)check whether the smallest divides biggest
3) if not then decrement the value of biggest number by smallest number
4) go back to 1
5) if yes then print the current smallest number

gcd(60,315)
60 = 3x5x4
315 = (3^2)x5x7
gcd is 15
so we can rewrite it as
60 = 15*4
315 = 15*21
if we repeatedly deduct the smallest value from biggest number and update the numbers every time 15 will come out as common 15(21-4), 15(17-4), 15(13-4),...15(5-4), after this we will have two numbers 15 and 60, 15 divides both 15 and 60 hence gcd is 15

Complexity:
Time : O(n/m)
Space: O(1)

```
void gcd_super_optimized(int n,int m){
    int rem;
    if(n<m){
        n = n + m;
        m = n - m;
        n = n - m;
    }
    while(n%m){
        rem = n%m;
        n = m;
        m = rem;
    }
    /*
    */
    cout<<m<<endl;

    /*
```

```
   */

}
check euclidean method for finding gcd for understanding the logic
      gcd(35,9) -> 35%9 = 8 -> 9%8 = 1 -> 8%1 -> 0
      the last nonzero remainder is the gcd which is in this case is 1
Complexity:
      Time : O(n/m)
      Space: O(1)
```

Q-5) Check if the given number is Armstrong number or not.

```
void is_amstrong(int n){
   int k=0,sum=0,m=n;
   while(n){
      k++;
      n/=10;
   }
   n=m;
   while(n){
      sum+=pow(n%10,k);
      n/=10;
   }
   if(sum==m) cout<<"It is an amstrong number"<<endl;
   else cout<<"It is not an amstrong number"<<endl;
}
```

A number is said to be armstrong number if sum of each of its digits raised to power: total number of digits in the number is equal to the number itself.
We calculate the total number of digits in the number and store it in "k" and then we extract the last digit and raise it to power k and add that to "sum".
Complexity:

Time : O(n)+O(n) where "n" is number of digits
Space: O(1)

Q-6) Print all the divisors of the given number.

```
void divisors(int num){
   int i = 1;
   while(i<num){
      if(num%i==0){
         cout<<i<<" ";
      }
      i++;
   }
   cout<<num<<endl;
   /*
   time complexity: O(n)
   advantage: output is sorted
   */
}
```

Run the loop from 1 to num, if any number divides num then print it.
Complexity:
Time : O(n) where n is number
Space: O(1)

```
void divisors_optimized(int num){
   int i = 1, j=0;
   for(int i=1; i<=sqrt(num); i++){
      if(num%i==0){
         j = num/i;
         if(i==j){
            cout<<i<<endl;
         }
         else
            cout<<i<<" "<<j<<" ";
      }
   }
   /*
   */
}
```

all the divisors are paired to another divisor that when multiplied together makes a number so if we find 1 dividor, then the quotient is also a divisor.
the square root of the number splits the pairs

```
   36: 1, 2, 3, 4, 6, 9, 12, 18, 36
        |  |  |  |  ^  |  |  |  |
        |  |  |   -------  |  |  |
        |  |  |------------ |  |  |
```

Complexity:

Time : O(sqrt(n))

Space: O(1)

Disadvantage: output is not sorted

Q-7) Check if given number is prime or not.

```
bool isPrime(int n)
{
   if(n==1) return false;
   for(int i=2;i<=sqrt(n);i++){
      if(n%i==0){
         return false;
      }
   }
   return true;
   /*
   */
}
```

If there are no divisors till sqrt(n), that means there are no more divisors as the sqrt(n) splits the pairs of divisors. Refer Ques-6 in Basic Maths section for better understanding.

Complexity:

Time : O(sqrt(n))

Space: O(1)

Q-1) Print name "n" times.

```
#include<bits/stdc++.h>
using namespace std;
void printNtimes(string a, int n){
   if(n==0){
      return;
   }
   cout<<a<<" ";
   printNtimes(a,n-1);
}
int main()
{
   int num;
   string name;
   cout<<"Enter name and number: ";
   cin>>name>>num;
   printNtimes(name,num);
}
```

Complexity:

Time : O(n)

Space: O(n) -> worst case which will happen if the recursion stack becomes full.

Q-2) Print 1 to N using Recursion.

```cpp
#include<bits/stdc++.h>
using namespace std;

// a backtracking approach aka backward recursion: first goes to the depth and then print and comes out
void print1ton(int n)
{
   if(n==0){
      return;
   }
   print1ton(n-1);
   cout<<n<<" ";
}

// forward recursion: first, number is printed and then function is called
void print1ton_2(int i, int n){
   if(i>n){
      return;
   }
   cout<<i<<" ";
   print1ton_2(i+1,n);
}
int main()
{
   print1ton(5);
   cout<<endl;
   print1ton_2(1,3);
   return 0;
}
```

Complexity:

Time : O(N) -> Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N)

Space: O(N) -> In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space

Q-3) Print N to 1 using Recursion

```
#include<bits/stdc++.h>
using namespace std;

// forward recursion: first, number is printed and then function is called
void printnto1(int n)
{
   if(n==0){
      return;
   }
   cout<<n<<" ";
   printnto1(n-1);
}

// a backtracking approach aka backward recursion: first goes to the depth and then print and comes out
void printnto1_2(int i, int n){
   if(i>n){
      return;
   }
   printnto1_2(i+1,n);
   cout<<i<<" ";
}
int main()
{
   printnto1(7);
   cout<<endl;
   printnto1_2(1,4);
   return 0;
}
```

Complexity:

Time : O(N) -> Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N)

Space: O(N) -> In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space

Q-4) Calculate sum of first N numbers.

```
#include<bits/stdc++.h>
using namespace std;

int sum_1ton(int s){
   if(s==0){
      return 0;
   }
   s += sum_1ton(s-1);
   return s;
}

int sum_1ton_2(int i, int sum){
   if(i<1){
      return 0;
   }
   sum = i + sum_1ton_2(i-1,sum);
   return sum;
}

int main()
{
   int num;
   cin>>num;
   cout<<sum_1ton_2(num,0);
   return 0;
}
```

We can also use formula: sum of 1 to n = n(n+1)/2 but just for practice we are using recursion.

Complexity:

Time : O(n)

Space: O(n) worst case if the recursion stack is full

Q-5) Factorial of given number.

```
#include<bits/stdc++.h>
using namespace std;
int factorial(int n){
   if(n<1){
      return 1;
   }
   return n*factorial(n-1);
}

int main()
{
   int num;
   cin>>num;
   cout<<factorial(num);
   return 0;
}
```

Complexity:

Time : O(N) -> Since the function is being called n times, and for each function, we have only one printable line that takes O(1) time, so the cumulative time complexity would be O(N)

Space: O(N) -> In the worst case, the recursion stack space would be full with all the function calls waiting to get completed and that would make it an O(N) recursion stack space

Q-6) Reverse an array.

```
void reverse_array(int arr[], int i, int j){
   if(i>=j){
      return;
   }
   int temp;
   temp = arr[j];
   arr[j] = arr[i];
   arr[i] = temp;
   reverse_array(arr,i+1,j-1);
}
```

for swapping we can also use swap(arr[i],arr[j])
or

```
arr[i] = arr[i]+arr[j];
arr[j] = arr[i]-arr[j];
arr[i] = arr[i]-arr[j];
```

this will save extra space
we can also use library function reverse(arr,arr+n)
Complexity:

Time : O(n)
Space: O(1)

Q-7) Check if given string is palindrome or not.

```
bool is_alphanum(char c){
   return (c>='A' && c<='Z') || (c>='a' && c<='z') || (c>='0' && c<='9');
}

char to_lower(char c){
   return (c>='A' && c<='Z')?c+32:c;
}

// recursive approach
bool check_palindrome(string s, int i, int j, int l){
   if(i>=j) return true;
   while(!(is_alphanum(s[i]))){
      i++;
      if(i==l) break;
   }
   while(!(is_alphanum(s[j]))){
      j--;
      if(j<0) break;
   }
   if(i==l && j<0) return true;
   if(to_lower(s[i])==to_lower(s[j])){
      return true && check_palindrome(s,i+1,j-1,l);
   }
   else return false;
}

bool check_palindrome_without_recursion(string s){
   int i=0, l=s.size();
   int j=l-1;
   while(i<j){
      while(!(is_alphanum(s[i]))){
         i++;
         if(i==l) break;
      }
      while(!(is_alphanum(s[j]))){
         j--;
         if(j<0) break;
      }
      if(i==l && j==0) return true;
      if(!(to_lower(s[i++])==to_lower(s[j--]))) return false;
   }
```

```
    return true;
}
```

We use two pointer approach. i points to 0th index and j points to last. We increment i and decrement j until we find the alphanumeric character at both ends. Then we check if both the chars are same or not irrespective of upper case or lower case.
If in case all the letters are non-alphanumeric then i will reach the rightmost end and j will reach the leftmost end and in this case the string will be considered as a palindrome.
Complexity:

Time : O(n) -> actually takes n/2 time

Space: O(1)

Q-8) Fibonacci Number

```
// recursive approach
int fib(int n) {
   if(n==0) return 0;
   if(n==1) return 1;
   return fib(n-1) + fib(n-2);
}
Complexity:
      Time : O(2^n)
      Space: O(n) --> if stack overflows

int fib_without_recursion(int n){
   if(n==0) return 0;
   else if(n==1) return 1;
   int i=0, j=1;
   while(n>=2){
      j+=i;
      i=j-i;
      n--;
   }
   return j;
}
Complexity:
      Time : O(n)
      Space: O(1)
```