**Data structures and algorithms:**

**Chapter one : time complexity**

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:  
  
thetanotation

**1) O Notation:** The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.  
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.  
3n3 + 6n2 + 6000 = O(n3)  
Dropping lower order terms is always fine because there will always be a n0 after which O(n3) has higher values than On2) irrespective of the constants involved.  
For a given function g(n), we denote O(g(n)) is following set of functions.

O(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1\*g(n) <= f(n) <= c2\*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1\*g(n) and c2\*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.  
  
BigO**2) Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.  
If we use O notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:  
1. The worst case time complexity of Insertion Sort is O(n^2).  
2. The best case time complexity of Insertion Sort is O(n).  
  
The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and

n0 such that 0 <= f(n) <= c\*g(n) for

all n >= n0}

BigOmega**3) Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.  
  
Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The Omega notation is the least used notation among all three.  
  
For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and

n0 such that 0 <= c\*g(n) <= f(n) for

all n >= n0}.

**Worst case and best case time complexities:**

It is important to analyze an algorithm after writing it to find it's efficiency in terms of time and space in order to improve it if possible.  
  
When it comes to analyzing algorithms, the asymptotic analysis seems to be the best way possible to do so. This is because asymptotic analysis analyzes algorithms in terms of the input size. It checks how are the time and space growing in terms of the input size.  
  
In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.  
  
We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Below is the algorithm for performing linear search:

// Linearly search x in arr[].   
// If x is present then return the index,  
// otherwise return -1  
search(int arr[], int n, int x)  
{  
 int i;  
  
 for (i=0; i  
 {  
 if (arr[i] == x)  
 return true;  
 }  
  
 return false;  
}

**Worst Case Analysis (Usually Done)** In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(N), where N is the number of elements in the array.  
  
**Average Case Analysis (Sometimes done)**In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are [uniformly distributed](http://en.wikipedia.org/wiki/Uniform_distribution_%28discrete%29) (including the case of x not being present in array). So we sum all the cases and divide the sum by (N+1). Following is the value of average case time complexity.

Average Case Time = analysis1

= analysis2

= Θ(N)

**Best Case Analysis (Bogus)**: In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on N). So time complexity in the best case would be O(1)  
  
**Important Points**:

* Most of the times, we do the worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is a good piece of information.
* The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
* The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

**Time complexity analysis:**

We have already discussed *Asymptotic Analysis*,  *Worst, Average and Best Cases* and *Asymptotic Notations*. In this post, analysis of iterative programs with simple examples is discussed.

1. **O(1):**Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

// set of non-recursive and non-loop statements

For example [swap() function](http://geeksquiz.com/c-program-swap-two-numbers/" \t "_blank) has O(1) time complexity.  
A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

// Here c is a constant

for (int i = 1; i <= c; i++) {

// some O(1) expressions

}

1. **O(n):** Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example following functions have O(n) time complexity.
2. // Here c is a positive integer constant
3. for (int i = 1; i <= n; i += c) {
4. // some O(1) expressions
5. }
7. for (int i = n; i > 0; i -= c) {
8. // some O(1) expressions

}

1. **O(nc)**: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have O(n2) time complexity
3. for (int i = 1; i <=n; i += c) {
4. for (int j = 1; j <=n; j += c) {
5. // some O(1) expressions
6. }
7. }
9. for (int i = n; i > 0; i -= c) {
10. for (int j = i+1; j <=n; j += c) {
11. // some O(1) expressions

}

1. **O(Logn)**: Time Complexity of a loop is considered as O(Logn) if the loop variables is divided / multiplied by a constant amount.
3. for (int i = 1; i <=n; i \*= c) {
4. // some O(1) expressions
5. }
7. for (int i = n; i > 0; i /= c) {
8. // some O(1) expressions
9. }

For example [Binary Search(refer iterative implementation)](http://geeksquiz.com/binary-search/" \t "_blank) has O(Logn) time complexity. Let us see mathematically how it is O(Log n). The series that we get in first loop is 1, c, c2, c3, ... ck. If we put k equals to Logcn, we get cLogcn which is n.

1. **O(LogLogn)**: Time Complexity of a loop is considered as O(LogLogn) if the loop variables is reduced / increased exponentially by a constant amount.
3. // Here c is a constant greater than 1
4. for (int i = 2; i <=n; i = pow(i, c)) {
5. // some O(1) expressions
6. }
8. // Here fun() is function to find square root
9. // or cuberoot or any other constant root
10. for (int i = n; i > 1; i = fun(i)) {
11. // some O(1) expressions
12. }

**How to combine time complexities of consecutive loops?** When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

for (int i = 1; i <=m; i += c) {

// some O(1) expressions

}

for (int i = 1; i <=n; i += c) {

// some O(1) expressions

}

Time complexity of above code is O(m) + O(n) which is O(m+n)

If m == n, the time complexity becomes O(2n) which is O(n).

**How to calculate time complexity when there are many if, else statements inside loops?** As discussed earlier, the worst-case time complexity is the most useful among best, average and worst. Therefore we need to consider the worst case. We evaluate the situation when values in if-else conditions cause a maximum number of statements to be executed.  
  
For example, consider the linear search function where we considered the case when an element is present at the end or not present at all.  
  
When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

**Chapter two: mathematics**

Given an integral number **N**. The task is to find the count of digits present in this number.  
  
Let's say:

**N = 2019**  
  
Number of digits in N here is 4 and,  
the digits are: 2, 0, 1, 9.

**Some more Examples**:

**N = 1567**  
Number of digits = 4  
  
**N = 256**  
Number of digits = 3  
  
**N = 58964**  
Number of digits = 5

Solution 1

**Simple Solution**: A Simple Solution that comes in mind is:

1. Check whether the number N is equal to zero.
2. Increase the count of digits by 1 if N is not zero.
3. Reduce the number by dividing it by 10.
4. Repeat the above steps until the number is reduced to zero.

**Dry-run of above algorithm**: Consider an example, N = 58964. Initialize a variable **digitsCount** to zero which will store the count of digits. Keep incrementing *digitsCount* until N is not zero, and reduce it by dividing by 10 at each step.

**Iteration 1:** N **not equals** to 0  
Increment digitsCount, digitsCount = digitsCount + 1.  
digitsCount = 0 + 1 = 1.  
N = N/10 = 58964/10 = 5896.  
  
**Iteration 2:** N **not equals** to 0  
Increment digitsCount, digitsCount = digitsCount + 1.  
digitsCount = 1 + 1 = 2.  
N = N/10 = 5896/10 = 589.  
  
**Iteration 3:** N **not equals** to 0  
Increment digitsCount, digitsCount = digitsCount + 1.  
digitsCount = 2 + 1 = 3.  
N = N/10 = 589/10 = 58.  
  
**Iteration 4:** N **not equals** to 0  
Increment digitsCount, digitsCount = digitsCount + 1.  
digitsCount = 3 + 1 = 4.  
N = N/10 = 58/10 = 5.  
  
**Iteration 5:** N **not equals** to 0  
Increment digitsCount, digitsCount = digitsCount + 1.  
digitsCount = 4 + 1 = 5.  
N = N/10 = 5/10 = 0.  
  
**Iteration 6:** N becomes equal to 0.  
Terminate any further operation.  
Return value of digitsCount.  
  
Therefore, number of digits = 5.

**Analysis of above algorithm**: You can clearly see that the number of operations performed in the above solution is equal to the count of digits present in the number. So, the time complexity of the solution is **O(digitsCount)**.

Solution 2

**Better Solution**: A better solution is to use mathematics to solve this problem. The number of digits in a number say N can be easily obtained by using the formula:

number of digits in N = log10(N) + 1.

**Derivation**: Suppose the number of digits in the number **N** is **K**.  
  
Therefore, we can say that:

10K-1 <= N < 10K

Applying base-10 logarithm to both sides in the above equation, we get:

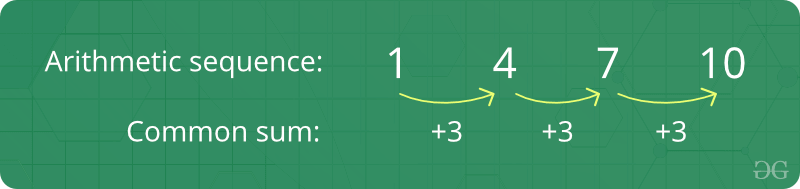
K-1 <= log10(N) < K.  
  
or, K - 1 + 1 <= log10(N) + 1 < K + 1  
or, K <= log10(N) + 1 < K + 1

Therefore,

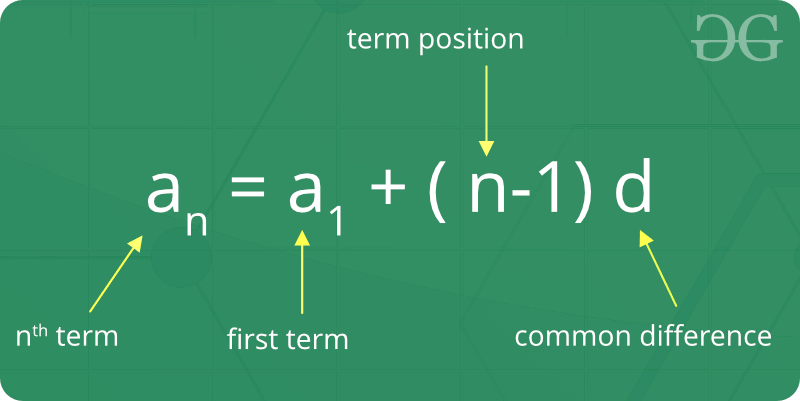
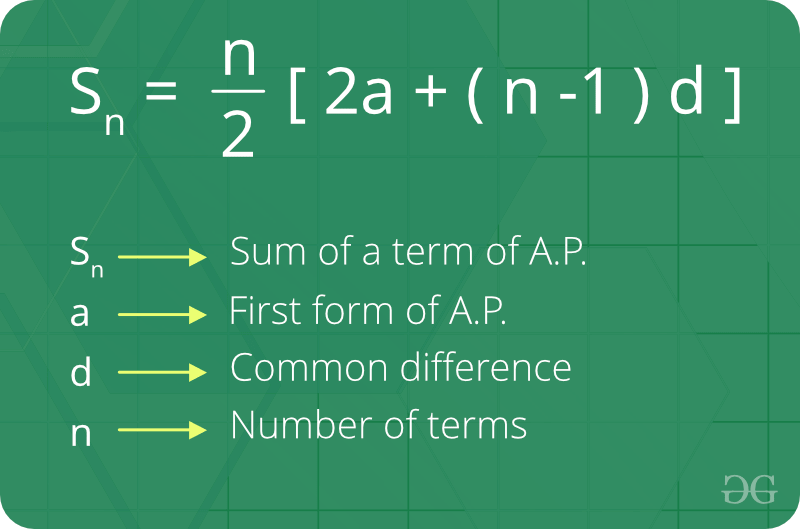
K = floor(log10(N) + 1)

**Analysis of above algorithm**: Since the above algorithm works in a single operation by using two mathematical operations i.e., finding logarithmic and floor value. Therefore, the time complexity of the solution is **O(1)**.

Arithmetic Progression

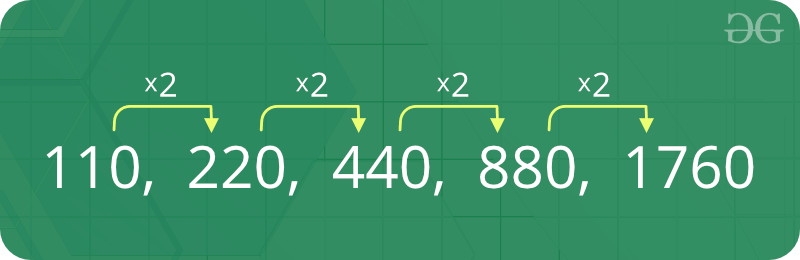
A sequence of numbers is said to be in an **Arithmetic progression** if the difference between any two consecutive terms is always the**same**. In simple terms, it means that the next number in the series is calculated by adding a fixed number to the previous number in the series. For example, 2, 4, 6, 8, 10 is an AP because the difference between any two consecutive terms in the series (common difference) is same (4 - 2 = 6 - 4 = 8 - 6 = 10 - 8 = 2).  
  
  
  
**Facts about Arithmetic Progression :**

1. **Initial term:** In an arithmetic progression, the first number in the series is called the initial term.
2. **Common difference:** The value by which consecutive terms increase or decrease is called the common difference.
3. The behavior of the arithmetic progression depends on the common difference d. If the common difference is positive, then the members (terms) will grow towards positive infinity, but if the common difference is negative, then the members (terms) will grow towards negative infinity.

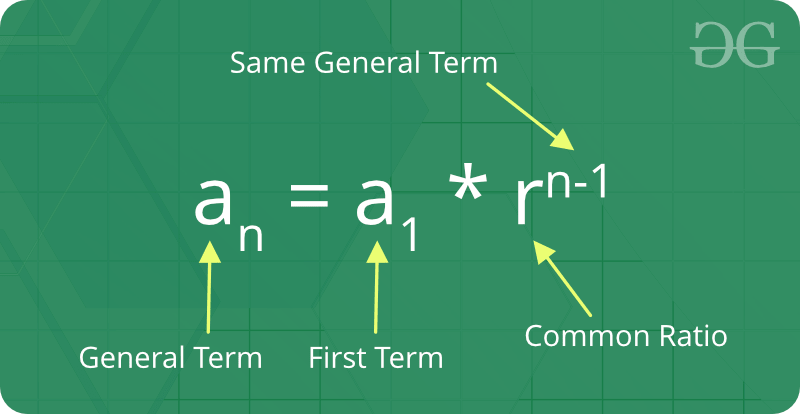
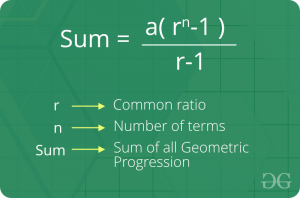
**Formula of nth term of an A.P :**  
If 'a' is the initial term and 'd' is the common difference.Thus, the explicit formula is:  
  
  
**Formula of sum of first n term of A.P:**  
  
**General Formulas to solve problems related to Arithmetic Progressions**: If ‘a’ is the first term and ‘d’ is the common difference:

* **nth term** of an AP = a + (n-1)\*d.
* **Arithmetic Mean** = Sum of all terms in the AP / Number of terms in the AP.
* **Sum of ‘n’ terms** of an AP = 0.5 n (first term + last term) = 0.5 n [ 2a + (n-1) d ].

Geometric Progression

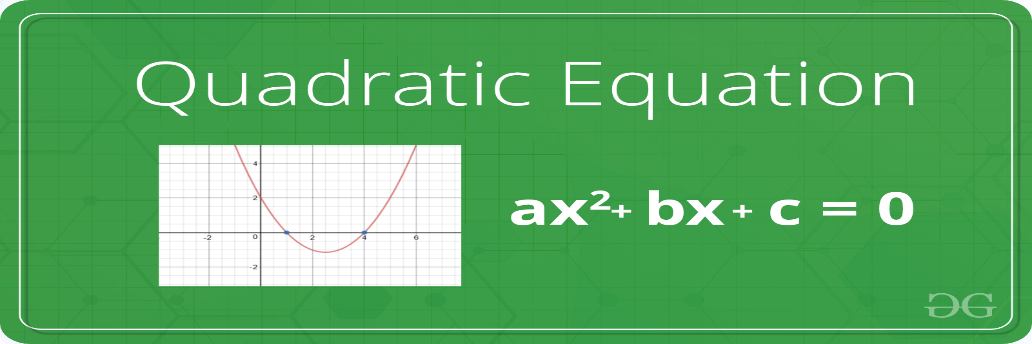
A sequence of numbers is said to be in a**Geometric progression** if the ratio of any two consecutive terms is always the same. In simple terms, it means that the next number in the series is calculated by multiplying a fixed number to the previous number in the series. For example, 2, 4, 8, 16 is a GP because ratio of any two consecutive terms in the series (common ratio) is the same (4 / 2 = 8 / 4 = 16 / 8 = 2).  
  
  
**Facts about Geometric Progression :**

1. **Initial term:** In a geometric progression, the first number is called the initial term.
2. **Common ratio:** The ratio of any two consecutive terms by taking the previous term in the denominator.
3. The behaviour of a geometric sequence depends on the value of the common ratio. If the common ratio is:
   * Positive, the terms will all be of the same sign as the initial term.
   * Negative, the terms will alternate between positive and negative.
   * Greater than 1, there will be exponential growth towards positive or negative infinity (depending on the sign of the initial term).
   * 1, the progression is a constant sequence.
   * Between -1 and 1 but not zero, there will be exponential decay towards zero.
   * -1, the progression is an alternating sequence.
   * Less than -1, for the absolute values there is exponential growth towards (unsigned) infinity, due to the alternating sign.

**Formula of nth term of a Geometric Progression :** If ‘a’ is the first term and ‘r’ is the common ratio. Thus, the explicit formula is:  
  
  
**Formula of sum of nth term of Geometric Progression:**  
  
**General Formulas to solve problems related to Geometric Progressions**:  
  
If ‘a’ is the first term and ‘r’ is the common ratio:

* **nth term of a GP** = a\*rn-1.
* **Geometric Mean** = nth root of the product of n terms in the GP.
* **Sum of ‘n’ terms** of a GP (r < 1) = [a (1 – rn)] / [1 – r].
* **Sum of ‘n’ terms** of a GP (r > 1) = [a (rn – 1)] / [r – 1].
* **Sum of infinite terms** of a GP (r < 1) = (a) / (1 – r).

**Quadratic equation:**

  
  
A **quadratic equation** is a second-order polynomial equation of a variable say **x**. The general form of a quadratic equation is given as:

**a\*x2 + b\*x + c = 0**  
  
Where a,b and c are real known values and,  
a can't be zero.

**Roots of an Equation**: The roots of an equation are the values for which the equation satisfies the given condition. For Example, the roots of equation **x2 - 7x - 12 = 0** are 3 and 4 respectively. If we replace the value of x by 3 and 4 individually in the equation, the equation will evaluate to zero.  
  
**A quadratic equation has two roots**. The roots of a quadratic equation can be easily obtained by using the quadratic formula:

roots = (-b ± √(b2 - 4ac))/2a

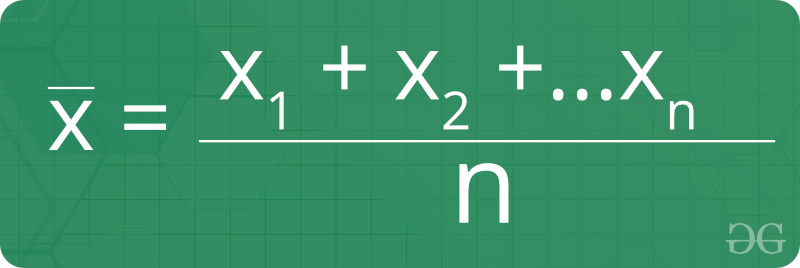
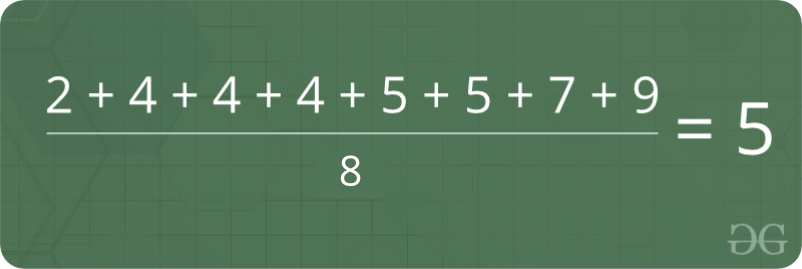
**Derivation**:

ax2 + bx + c = 0  
  
or, ax2 + bx = -c  
  
or, x2 + (b/a)x = -(c/a)  
  
or, x2 + (b/a)x + (b2/4a2) - (b2/4a2) = -(c/a)  
  
or, x2 + (b/a)x + (b2/4a2) = -(c/a) + (b2/4a2)  
  
or, (x + b/2a)2 = -(c/a) + (b2/4a2)  
  
or, (x + b/2a)2 = (b2 - 4ac) /4a2  
  
or, (x + b/2a) = ± √(b2 - 4ac) /2a  
  
or, x = (-b ± √(b2 - 4ac))/2a

There arises **three cases** as described below while finding the roots of a quadratic equation:

If **b2 < 4ac**, then roots are complex  
(not real).  
For example, roots of x2 + x + 1 = 0 are  
-0.5 + i1.73205 and -0.5 - i1.73205  
  
If **b2 = 4ac**, then roots are real   
and both roots are same.  
For example, roots of x2 - 2x + 1 = 0 are 1 and 1  
  
If **b2 > 4ac**, then roots are real   
and different.  
For example, roots of x2 - 7x - 12 = 0 are 3 and 4

Mean

  
**Mean**is defined as the average of a given set of data. Let us consider the sequence of numbers **2, 4, 4, 4, 5, 5, 7, 9**, the mean (average) of this given sequence is 5.  
  
  


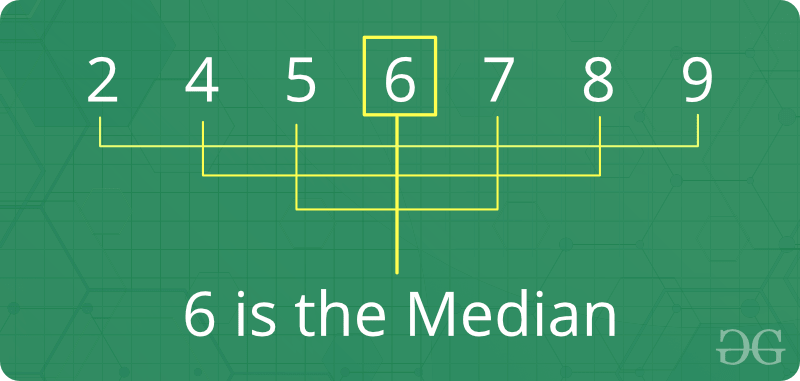
**Formula for finding Mean:**   
Where, **x1, x2,...xn** denotes the terms of the given sequence and **n**is the count of numbers present in the given sequence.  
  
**Facts about Mean :**

1. The mean (or average) is the most popular and well known measure of central tendency.
2. It can be used with both discrete and continuous data, although its use is most often with continuous data.
3. There are other types of means such as Geometric mean, Harmonic mean, and Arithmetic mean.
4. Mean is the only measure of central tendency where the sum of the deviations of each value from the mean is always zero.

Median

**Median**is the middle value of a set of data. To determine the median value in a sequence of numbers, the numbers must first be arranged in an ascending order.

* If the count of numbers in the sequence is ODD, the median value is the number that is in the middle, with the same amount of numbers below and above.
* If the count of numbers in the sequence is EVEN, the median is the average of the two middle values.

  
  
**Formula for finding Median :**

* If the count of numbers is odd: After sorting the sequence,

Median = {(N+1)/2}th value;  
N is the number of terms.

* If the count of numbers is even: After sorting the sequence,

Median = Average of (N/2)th and {(N/2) + 1}th value;  
N is the number of terms

**Facts about Median :**

1. Median is an important measure (compared to mean) for distorted data, because median is not so easily distorted. For example, median of {1, 2, 2, 5, 100) is 2 and mean is 22.
2. If the user adds a constant to every value, the mean and median increases by the same constant.
3. If the user multiplies every value by a constant, the mean and the median will also be multiplied by that constant.

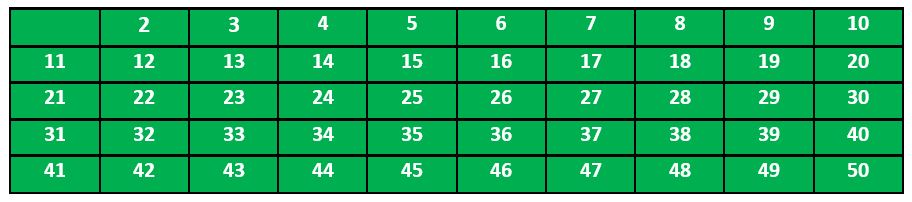
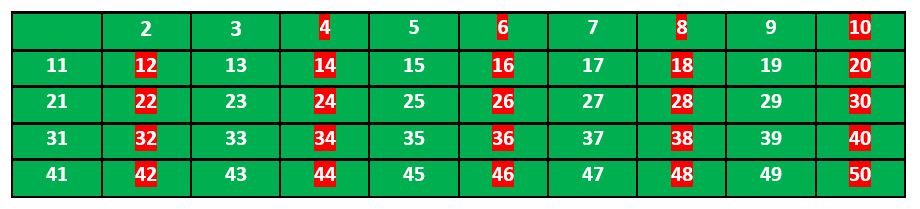
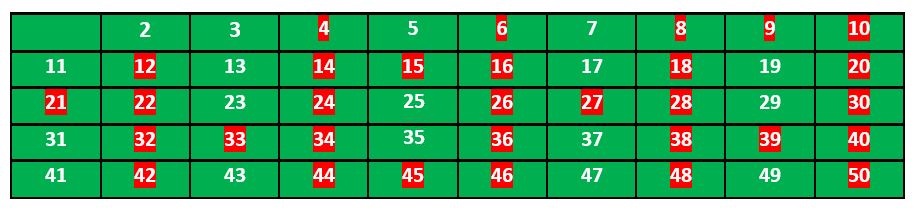
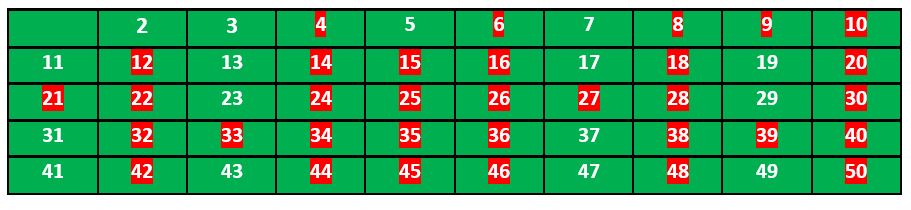
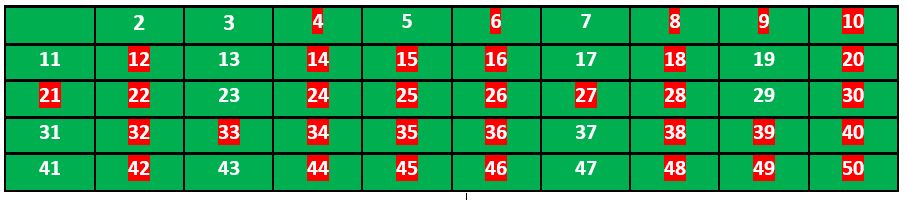
A **prime number** is a whole number greater than 1, which is only divisible by 1 and itself. First few prime numbers are : 2, 3, 5, 7, 11, 13, 17, 19, 23, ........  
  
  
  
**Naive Method to Check if a number is Prime**: Since a number is prime only if it is divisible by 1 and the number itself, the naive method to check for primality of a number would be to iterate from 1 to N and check if there aren't any factors of N except and 1 and N itself.  
  
**Algorithm**:

1. If N is less than 2, it is not prime. Return False.
2. else:  
   * Iterate from 2 to N-1 and check if any of the numbers between 2 and N-1 (both inclusive) divides N or not. If yes, then N is not prime, return False.
   * Otherwise, return True.

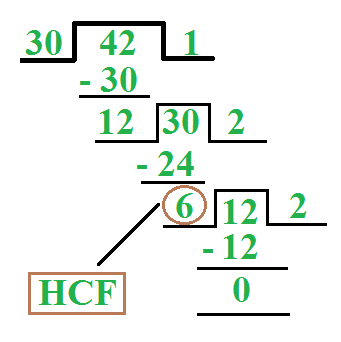
**Analysis of the above algorithm:** Since we are traversing linearly from 2 to N-1, the time complexity of the above algorithm will be linear **O(N)**.

**Sieve of Eratosthenes**

Using **Sieve of Eratosthenes** is the most efficient way of generating prime numbers up to a given number N.  
  
Following is the algorithm to find all the prime numbers less than or equal to a given integer *n* by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to *n*: (2, 3, 4, ..., *n*).
2. Initially, let *p* equal 2, the first prime number.
3. Starting from *p*2, count up in increments of *p* and mark each of these numbers greater than or equal to *p2* itself in the list. These numbers will be *p(p+1)*, *p(p+2)*, *p(p+3)*, etc..
4. Find the first number greater than *p* in the list that is not marked. If there was no such number, stop. Otherwise, let *p* now equal this number (which is the next prime), and repeat from step 3.  
   **Explanation with Example:** Let us take an example when n = 50. So we need to print all print numbers smaller than or equal to 50.  
     
   We create a list of all numbers from 2 to 50.  
   [](https://media.geeksforgeeks.org/wp-content/uploads/SieveofEratosthenes1.jpg)  
   According to the algorithm we will mark all the numbers which are divisible by 2 and are greater than or equal to the square of it.  
   [](https://media.geeksforgeeks.org/wp-content/uploads/SieveofEratosthenes2.jpg)  
   Now we move to our next unmarked number 3 and mark all the numbers which are multiples of 3 and are greater than or equal to the square of it.  
     
     
   We move to our next unmarked number 5 and mark all multiples of 5 and are greater than or equal to the square of it.  
   [](https://media.geeksforgeeks.org/wp-content/uploads/SieveofEratosthenes4.jpg)  
   We continue this process and our final table will look like below:  
   [](https://media.geeksforgeeks.org/wp-content/uploads/SieveofEratosthenes5.jpg)  
   So the prime numbers are the unmarked ones: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,

**Lcm and hcf**

**Factors and Multiples:**All numbers that divide a number completely, i.e., without leaving any remainder, are called factors of that number. For example, 24 is completely divisible by 1, 2, 3, 4, 6, 8, 12, 24. Each of these numbers is called a factor of 24 and 24 is called a multiple of each of these numbers.  
  
**LCM :**LCM stands for *Least Common Multiple*. The lowest number that is exactly divisible by each of the given numbers is called the least common multiple of those numbers. For example, consider the numbers 3, 31, and 62 (2 x 31). The LCM of these numbers would be 2 x 3 x 31 = 186.  
  
To **find the LCM of the given numbers**, express each number as its prime factors. The product of the highest power of the prime numbers that appear in the prime factorization of any of the numbers gives us the LCM.  
For example, consider the numbers 2, 3, 4 (2 x 2), 5, 6 (2 x 3). The LCM of these numbers is 2 x 2 x 3 x 5 = 60. The highest power of 2 comes from prime factorization of 4, the highest power of 3 comes from prime factorization of 3 and prime factorization of 6, and the highest power of 5 comes from prime factorization of 5.  
  
**HCF :**The term HCF stands for *Highest Common Factor*. The largest number that divides two or more numbers is the highest common factor (HCF) for those numbers. For example, consider the numbers 30 (2 x 3 x 5), 36 (2 x 2 x 3 x 3), 42 (2 x 3 x 7), 45 (3 x 3 x 5). 3 is the largest number that divides each of these numbers, and hence, is the HCF for these numbers.  
  
**HCF is also known as Greatest Common Divisor (GCD).**  
To **find the HCF of two or more numbers**, express each number as its prime factors. The product of the minimum powers of common prime terms in both of the prime factorization gives the HCF. This is the method we illustrated in the above step.  
  
Also, for finding the HCF of two numbers, we can proceed by the long division method. We divide the larger number by the smaller number (divisor). Now, we divide the divisor by the remainder obtained in the previous stage. We repeat the same procedure until we get zero as the remainder. At that stage, the last divisor would be the required HCF.  
  
For example, HCF of 30 and 42:  


Basic Euclidean Algorithm for HCF

The Euclidean algorithm is based on the below facts:

* If we subtract the smaller number from larger (we reduce larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
* Now instead of subtraction, if we divide the smaller number, the algorithm stops when the remainder is found to be 0.

Below is the recursive function for finding GCD using Euclidean Algorithm:

gcd(a, b)  
{  
 if (a == 0)  
 return b;  
  
 return gcd(b % a, a);  
}

**Time Complexity**: O(log(min(a, b)))  
  
**Important properties of LCM and HCF**:

1. For two numbers say, 'a' and 'b', LCM x HCF = a x b.
2. HCF of co-primes = 1.
3. For two fractions,  
   * HCF = HCF (Numerators) / LCM (Denominators)
   * LCM = LCM (Numerators) / HCF (Denominators)

**Factorial**:

In mathematics, the factorial of a number say **N**is denoted by **N!**. The factorial of a number is calculated by multiplying all the integers between 1 and N(both inclusive).  
  
For Example, **4!** = **4 \* 3 \* 2 \* 1**=**24**.  
  
That is,

**N!** = N\*(N-1)\*(N-2)\*....\*2\*1

**Note**: As per convention, **0! = 1**.

**Sample Problem**: Given a number N, the task is to count number of **trailing zeroes** in factorial of N. That is, number of zeroes at the end in the number **N!**.  
  
**For Example**:

**Input**: N = 5  
**Output**: 1   
Factorial of 5 is 120 which has one trailing 0.  
  
**Input**: N = 20  
**Output**: 4  
Factorial of 20 is 2432902008176640000 which has  
4 trailing zeroes.

An efficient way to solve this problem is to observe the properties of prime factorization. Consider prime factors of **N!**. A trailing zero is always produced by the prime factors **2 and 5**. If we can count the number of 5s and 2s in prime factorization of N!, our task is done.  
  
Consider the following examples:

* **N = 5**: There is one 5 and 3 2s in prime factors of 5! (2 \* 2 \* 2 \* 3 \* 5). So count of trailing 0s is 1.
* **N = 11**: There are two 5s and three 2s in prime factors of 11! (2 8 \* 34 \* 52 \* 7). So count of trailing 0s is 2.

We can easily observe that the number of 2s in prime factors is always more than or equal to the number of 5s. So if we count 5s in prime factors, we are done.  
  
Now, how to count the total number of 5s in prime factors of N!? A simple way is to calculate floor(N/5). For example, 7! has one 5, 10! has two 5s. It is not done yet, there is one more thing to consider. Numbers like 25, 125, etc have more than one 5. For example, if we consider 28!, we get one extra 5, and the number of 0s becomes 6. Handling this is simple, first divide N by 5 and remove all single 5s, then divide by 25 to remove extra 5s, and so on. Following is the summarized formula for counting trailing 0s.

**Trailing 0s in N!** = Count of 5s in prime factors of n!  
 = floor(n/5) + floor(n/25) + floor(n/125) + ....

**Permutation**

**Permutation** is the different arrangements of a given number of elements taken one by one, or some, or all at a time. For example, if we have two elements A and B, then there are two possible arrangements, AB and BA.  
  
Number of permutations when 'r' elements are arranged out of a total of 'n' elements is n **Pr = n! / (n - r)!**. For example, let n = 4 (A, B, C and D) and r = 2 (All permutations of size 2). The answer is 4!/(4-2)! = 12. The twelve permutations are AB, AC, AD, BA, BC, BD, CA, CB, CD, DA, DB and DC.  
  
**Important Properties of Permutation**:

1. n P n = n\*(n-1)\*(n-2)\*......\*1 = n!.
2. n P 0 = n! / n! = 1.
3. n P 1 = n.
4. n P n-1 = n!.
5. n P r/n P r-1 = n - r + 1.

**Permutation with repetition allowed**: The number of permutation or arrangements of N numbers with repetition allowed will be **NN**. For Example, permutaions of {1,2} with repetitions will be {{1,1}, {1,2}, {2,1},{2,2}}.  
  
**Permutation with duplicates**: The number of permutations or arrangements of N objects of which p1 are of one kind, p2 are of second kind, ..., pk are of k-th kind and the rest if any, are of different kinds is: **N! / (p1! \* p2! \*....\*pk!)**.

Combination

**Combination** is the different selections of a given number of elements taken one by one, or some, or all at a time. For example, if we have two elements A and B, then there is only one way to select two items, we select both of them.  
  
Number of combinations when 'r' elements are selected out of a total of 'n' elements is n C r = **n! / [ (r !) \* (n - r)! ]**. For example, let n = 4 (A, B, C and D) and r = 2 (All combinations of size 2). The answer is 4!/((4-2)!\*2!) = 6. The six combinations are AB, AC, AD, BC, BD, CD.  
  
**Important Properties of Combination**:

1. n C 0 = n C n = 1.
2. n C r = n C n-r.
3. n C r + n C r-1 = n+1 C r.
4. n \* n-1 C r-1 = (n - r + 1)\* n C r-1.

**Modular arithmetic:**

Let us take a look at some of the **basic rules and properties** that can be applied in Modular Arithmetic (Addition, Subtraction, Multiplication etc.). Consider numbers **a** and **b** operated under modulo **M**.

1. (a + b) mod M = ((a mod M) + (b mod M)) mod M.
2. (a - b) mod M = ((a mod M) - (b mod M)) mod M.
3. (a \* b) mod M = ((a mod M) \* (b mod M)) mod M.

The above three expressions are valid and can be performed as stated. But when it comes to modular division, there are some limitations.  
  
There isn't any formula to calculate:

(a / b) mod M

For this we have to learn **modular inverse**.

Modular Inverse

The modular inverse is an integer 'x' such that.

a x ≡ 1 (mod M)

The value of x should be in {0, 1, 2, ... M-1}, i.e., in the ring of integer modulo M.  
  
The multiplicative inverse of "a modulo M" exists if and only if a and M are relatively prime (i.e., if gcd(a, M) = 1).  
  
**Examples:**

**Input**: a = 3, M = 11  
**Output**: 4  
Since (4\*3) mod 11 = 1, 4 is modulo inverse of 3  
One might think, 15 also as a valid output as "(15\*3) mod 11"   
is also 1, but 15 is not in ring {0, 1, 2, ... 10}, so not   
valid.  
  
**Input**: a = 10, M = 17  
**Output**: 12  
Since (10\*12) mod 17 = 1, 12 is modulo inverse of 3

**Methods of finding Modular Inverse**: There are two very popular methods of finding modular inverse of any number **a** under modulo **M**.

1. **Extended Euclidean Algorithm**: This method can be used when **a** and **M** are co-prime.
2. **Fermat Little Theorem**: This method can be used when **M** is prime.

Let us look at each of the above two methods in details:  
  
**Extended Euclidean algorithm** that takes two integers 'a' and 'b', finds their gcd and also find 'x' and 'y' such that,

ax + by = gcd(a, b)

To find the modulo inverse of 'a' under 'M', we put b = M in the above formula. Since we know that a and M are relatively prime, we can put value of gcd as 1.  
  
So, the formula becomes:

ax + My = 1

If we take modulo M on both sides, we get:

ax + My ≡ 1 (mod M)

We can remove the second term on the left side, as 'My (mod M)' would always be 0 for an integer y.  
  
Therefore,

ax ≡ 1 (mod M)

So the 'x' that we can find using [Extended Euclid Algorithm](https://www.geeksforgeeks.org/basic-and-extended-euclidean-algorithms/" \t "_blank) is modulo inverse of 'a'.  
  
**Fermat Little Theorem**: The Fermat’s little theorem states that if M is a prime number, then for any integer a, the number **aM – a** is an integer multiple of M.  
  
That is,

**aM ≡ a (mod M).**

Since, a and M are co-prime to each other then aM-1 is an integral multiple of M.  
That is,

aM-1 ≡ 1 (mod M)

If we multiply both sides by a-1, we get:

a-1 ≡ aM-2 mod M

Therefore, if **M is a prime number** to find modulo inverse of **a under M**, find **modular exponentiation of aM-2 under modulo M**.

**Chapter three : bitwise operator**

**Bitwise algorithm basics:**

The Bitwise Algorithms are used to perform operations at bit-level or to manipulate bits in different ways. The bitwise operations are found to be much faster and are some times used to improve the efficiency of a program.  
  
**For example**: To check if a number is even or odd. This can be easily done by using Bitwise-AND(&) operator. If the last bit of the operator is set than it is ODD otherwise it is EVEN. Therefore, if num & 1 not equals to zero than num is ODD otherwise it is EVEN.

**Bitwise Operators**

The operators that works at Bit level are called bitwise operators. In general there are six types of Bitwise Operators as described below:

* **& (bitwise AND)** Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. Suppose **A = 5** and **B = 3**, therefore **A & B = 1**.
* **| (bitwise OR)** Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1. Suppose **A = 5** and **B = 3**, therefore **A | B = 7**.
* **^ (bitwise XOR)** Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different. Suppose **A = 5** and **B = 3**, therefore **A ^ B = 6**.
* **<< (left shift)** Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
* **>> (right shift)** Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
* **~ (bitwise NOT)** Takes one number and inverts all bits of it. Suppose **A = 5**, therefore **~A = -6**.

**Important Facts about Bitwise Operators**:

* The left shift and right shift operators cannot be used with negative numbers.
* The bitwise XOR operator is the most useful operator from technical interview perspective. We will see some very useful applications of the XOR operator later in the course.
* The bitwise operators should not be used in place of logical operators.
* The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.
* The & operator can be used to quickly check if a number is odd or even. The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

**Bit algorithm important tactics:**

Let's look at some of the useful tactics of the Bitwise Operators which can be helpful in solving a lot of problems really easily and quickly.

1. **How to set a bit in the number 'num' :** If we want to set a bit at nth position in number 'num' ,it can be done using 'OR' operator( | ).  
   * First we left shift '1' to n position via (1 << n)
   * Then, use 'OR' operator to set bit at that position.'OR' operator is used because it will set the bit even if the bit is unset previously in binary representation of number 'num'.

1. **How to unset/clear a bit at n'th position in the number 'num' :**

Suppose we want to unset a bit at nth position in number 'num' then we have to do this with the help of 'AND' (&) operator.

* + First we left shift '1' to n position via (1 << n) than we use bitwise NOT operator '~' to unset this shifted '1'.
  + Now after clearing this left shifted '1' i.e making it to '0' we will 'AND'(&) with the number 'num' that will unset bit at nth position position.

1. **Toggling a bit at nth position :** Toggling means to turn bit 'on'(1) if it was 'off'(0) and to turn 'off'(0) if it was 'on'(1) previously.We will be using 'XOR' operator here which is this '^'. The reason behind 'XOR' operator is because of its properties.  
   * Properties of 'XOR' operator.  
     + 1^1 = 0
     + 0^0 = 0
     + 1^0 = 1
     + 0^1 = 1
   * If two bits are different then 'XOR' operator returns a set bit(1) else it returns an unset bit(0).

1. **Checking if bit at nth position is set or unset:**

It is quite easily doable using 'AND' operator.

* + Left shift '1' to given position and then 'AND'('&').

If the result of the AND operation is 1 then the bit at nth position is set otherwise it is unset.

1. **Divide by 2**:

x = x >> 1;

**Logic**: When we do arithmetic right shift, every bit is shifted to right and blank position is substituted with sign bit of number, 0 in case of positive and 1 in case of negative number. Since every bit is a power of 2, with each shift we are reducing the value of each bit by factor of 2 which is equivalent to division of x by 2.  
**Example**:

x = 18(00010010)  
x >> 1 = 9 (00001001)

1. **Multiplying by 2**:

x = x << 1;

**Logic**: When we do arithmetic left shift, every bit is shifted to left and blank position is substituted with 0 . Since every bit is a power of 2, with each shift we are increasing the value of each bit by a factor of 2 which is equivalent to multiplication of x by 2.  
**Example**:

x = 18(00010010)  
x << 1 = 36 (00100100)

1. **Find log base 2 of a 32 bit integer**:

int log2(int x)

{

int res = 0;

while (x >>= 1)

res++;

return res;

}

**Logic**: We right shift x repeatedly until it becomes 0, meanwhile we keep count on the shift operation. This count value is the log2(x).

1. **Flipping the bits of a number**: It can be done by a simple way, just simply subtract the number from the value obtained when all the bits are equal to 1 .  
   **For example**:

Number : Given Number  
Value : A number with all bits set in given number.  
Flipped number = Value – Number.  
  
Example :   
Number = 23,  
Binary form: 10111;  
After flipping digits number will be: 01000;  
Value: 11111 = 31;

1. **Swapping Two Numbers**: We can easily swap two numbers say **a** and **b** by using the XOR(^) operator by applying below operations:

a ^= b;  
b ^= a;   
a ^= b;

basic problems on bit manipulation:

Below are some problems which can be solved very easily using some of the basic concepts of Bit Manipulation. Let's look at each of these problems and the Bitwise approach of solving them:

* **Problem 1**: Given a number N, the task is to check whether the given number is a power of 2 or not.  
    
  **For Example**:

**Input** : N = 4  
**Output** : Yes  
22 = 4  
  
**Input** : N = 7  
**Output** : No  
  
**Input** : N = 32  
**Output** : Yes  
25 = 32

**Bitwise Solution**: If we subtract a number which is a power of 2 1 then all of it's unset bits after the only set bit become set; and the set bit become unset.  
  
For example, consider **4**( Binary representation: 100) and **16**(Binary representation: 10000), we get following after subtracting 1 from them:

3 –> 011  
15 –> 01111

You can clearly see that **bitwise-AND(&)** of 4 and 3 gives zero, similarly 16 and 15 also gives zero.  
  
So, if a number N is a power of 2 then **bitwise-AND(&)** of **N and N-1** will be zero. We can say that N is a power of 2 or not based on the value of N&(N-1).

* **Problem 2**: Given a number N, find the most significant set bit in the given number.  
    
  **Examples**:

**Input** : N = 10  
**Output** : 8  
Binary representation of 10 is 1010  
The most significant bit corresponds  
to decimal number 8.  
  
**Input** : N = 18  
**Output** : 16

**Bitwise Solution**: The most-significant bit in binary representation of a number is the highest ordered bit, that is it is the bit-position with highest value.  
  
One of the solution is first find the bit-position corresponding to the MSB in the given number, this can be done by calculating logarithm base 2 of the given number, i.e., **log2(N)** gives the position of the MSB in N.  
  
Once, we know the position of the MSB, calculate the value corresponding to it by raising 2 by the power of calculated position. That is, **value = 2log2(N)**.

* **Problem 3**: Given a number N, the task is to find the XOR of all numbers from 1 to N.  
    
  **Examples** :

**Input** : N = 6  
**Output** : 7  
// 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 = 7  
  
**Input** : N = 7  
**Output** : 0  
// 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 = 0

**Solution**:

* 1. Find the remainder of N by moduling it with 4.
  2. If rem = 0, then xor will be same as N.
  3. If rem = 1, then xor will be 1.
  4. If rem = 2, then xor will be N+1.
  5. If rem = 3 ,then xor will be 0.

**How does this work?**

When we do XOR of numbers, we get 0 as XOR value just before a multiple of 4. This keeps repeating before every multiple of 4.

Number Binary-Repr XOR-from-1-to-n  
1 1 [0001]  
2 10 [0011]  
3 11 [0000] <----- We get a 0  
4 100 [0100] <----- Equals to n  
5 101 [0001]  
6 110 [0111]  
7 111 [0000] <----- We get 0  
8 1000 [1000] <----- Equals to n  
9 1001 [0001]  
10 1010 [1011]  
11 1011 [0000] <------ We get 0  
12 1100 [1100] <------ Equals to n

Maximum and value explanation :

Given an array arr[] of N positive elements. The task is to find the Maximum AND Value generated by any pair of the element from the array.  
  
**Note**: AND is bitwise '&' operator.  
  
**Examples**:

**Input**: a[] = {4, 8, 12, 16}  
**Output**: 8  
The pairs 8 and 12 gives us the '&' value as 12.  
  
**Input**: a[] = {4, 8, 16, 2}  
**Output**: 0

A **naive** approach is to iterate for all the pairs using two for loops and check for the maximum '&' value of any pair.  
  
**Note**: This approach will not fit in the given time limit since the complexity of the above method jis O(N^2).

int findMaxium(int a[], int n)  
{  
 int maxi = 0;  
 for(int i = 0;i<n;i++)   
 {  
 for(int j = i+1;j<n;j++)  
 maxi = max(maxi, a[i] & a[j]);   
 }  
  
 return maxi;   
}

**Efficient Approach:** An efficient approach will be to look at this problem bitwise. Since we need to find the maximum '&' value. The first thing that strikes our mind is that the answer should have its LSB as far as possible. So, if two elements are considered as a pair, then their LSB should be set to as much left as possible. Let's take an example to understand this. Consider three elements {10, 8, 2}, so to get a maximum '&' value we need to take those elements whose LSB is as far as possible. In the given example, we can clearly see that 10(1010) and 8(1000) have their 4th-bit from the left set and hence will maximize the answer. Taking 2 and 10 will give our 2nd bit to be set which won't maximize our answer.  
  
So since the constraints permit till 10^4, hence the '&' value will also be less than that. 10^4 will range in 2^0 to 2^14, which means we need to start our checking from the 15th bit. Initially we loop from 15 to 0 and check for the count of numbers whose that particular bit is set. Once we get the count more than 2, the answer will have that bit set, and for the next bit from the left to be set we need to check for both the previous all bits and the current i-th bit. The previous bits can be added to the current bit using a '|' operator. In this way, we get all the positions of the bit which are set, which can be easily represented as a number.  
  
**Note**: We have started checking from bits 31 so that if the constraints were high, it can easily fit in.

// Utility function to check number of elements  
// having set msb as of pattern  
int checkBit(int pattern, int arr[], int n)  
{  
 int count = 0;  
 for (int i = 0; i < n; i++)  
 if ((pattern & arr[i]) == pattern)  
 count++;  
 return count;  
}  
   
// Function for finding maximum and value pair  
int maxAND (int arr[], int n)  
{  
 int res = 0, count;  
   
 // iterate over total of 30bits from msb to lsb  
 for (int bit = 31; bit >= 0; bit--)  
 {  
 // find the count of element having set msb  
 count = checkBit(res | (1 << bit),arr,n);  
   
 // if count >= 2 set particular bit in result  
 if ( count >= 2 )   
 res |= (1 << bit);   
 }  
   
 return res;  
}

Chapter four: recursion

**What is Recursion?**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are [Towers of Hanoi (TOH)](http://quiz.geeksforgeeks.org/c-program-for-tower-of-hanoi/), [Inorder/Preorder/Postorder Tree Traversals](https://www.cdn.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/), [DFS of Graph](https://www.cdn.geeksforgeeks.org/depth-first-traversal-for-a-graph/), etc.

**What is the base condition in recursion?**

In a recursive program, the solution to the base case is provided and the solution of bigger problem is expressed in terms of smaller problems.

int fact(int n)  
{  
 if (n < = 1) // base case  
 return 1;  
 else   
 return n\*fact(n-1);   
}

In the above example, the base case for n < = 1 is defined and a larger value of a number can be solved by converting to a smaller one till the base case is reached.  
  


**How a particular problem is solved using recursion?**

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.  
  
**Why Stack Overflow error occurs in recursion?** If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

int fact(int n)  
{  
 // wrong base case (it may cause  
 // stack overflow).  
 if (n == 100)   
 return 1;  
  
 else  
 return n\*fact(n-1);  
}

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

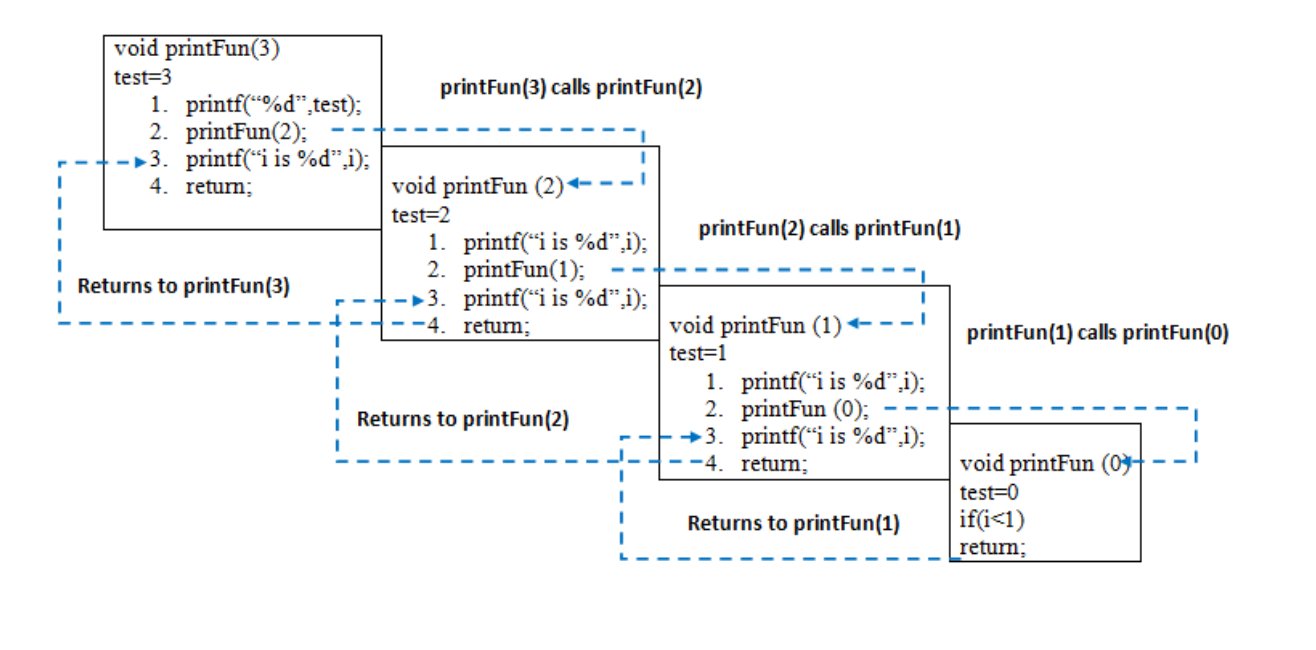
**How memory is allocated to different function calls in recursion?**

When any function is called from main(), the memory is allocated to it on stack. A recursive function calls itself, the memory for the called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.  
  
Let us take the example of how recursion works by taking a simple function:

void printFun(int test)  
{  
 if (test < 1)  
 return;  
 else  
 {  
 print test;  
 printFun(test-1); // statement 2  
 print test;  
 return;  
 }  
}  
  
// Calling function printFun()  
int test = 3;  
printFun(test);

**Output** :

3 2 1 1 2 3

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints ‘3’. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statements 1 to 4 are pushed in the stack. Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it returns to **printFun(1)**. Remaining statements of **printFun(1)**are executed and it returns to **printFun(2)** and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.  
  
  
  
  
**Disadvantage of Recursion**: Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa. Recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. A recursive program also has greater time requirements because of function calls and return overhead.  
  
**Advantages of Recursion**: Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of the stack data structure.

**Problem 1**

Given an unsorted array of **N** elements and an element **X**. The task is to write a recursive function to check whether the element X is present in the given array or not.  
  
**Example**:

**array[] = {1, 2, 3, 4, 5}  
X = 3.**  
  
The function should return True, as 3 is   
present in the array.

**Solution**: The idea is to compare the first element of the array with X. If the element matches with X then return True otherwise recur for the remaining part of the array.  
  
The **recursive function** will somewhat look like as shown below:

// **arr[]** is the given array   
// **l** is the lower bound in the array  
// **r** is the upper bound  
// **x** is the element to be searched for  
// l and r defines that search will be   
// performed between indices l to r  
  
bool recursiveSearch(int arr[], int l,   
 int r, int x)   
{   
 if (r < l)   
 return false;   
 if (arr[l] == x)   
 return true;   
 if (arr[r] == x)   
 return true;   
  
 return recursiveSearch(arr, l + 1,   
 r - 1, x);   
}

**Time Complexity**: The above algorithm runs in O(N) time where N is the number of elements present in the array.  
**Space Complexity**: There is no extra space used however the internal stack takes O(N) extra space for recursive calls.

Problem 2

Given a string, the task is to write a recursive function to check if the given string is palindrome or not.  
  
**Examples**:

**Input** : string = "malayalam"  
**Output** : Yes  
Reverse of malayalam is also  
malayalam.  
  
**Input** : string = "max"  
**Output** : No  
Reverse of max is not max.

**Solution**: The idea to write the recursive function is simple and similar to the above problem:

1. If there is only one character in the string, return true.
2. Else compare first and last characters and recur for remaining substring.

**Recursive Function**:

// **s** and **e** defines the start and end index of string  
  
bool isPalindrome(char str[], int s, int e)   
{   
 // If there is only one character   
 if (s == e)   
 return true;   
   
 // If first and last   
 // characters do not match   
 if (str[s] != str[e])   
 return false;   
   
 // If there are more than   
 // two characters, check if   
 // middle substring is also   
 // palindrome or not  
 if (s < e)   
 return isPalindrome(str, s + 1, e - 1);   
   
 return true;   
}

**Tail recursion:**As we read before, that recursion is defined when a function invokes/calls itself.  
  
**Tail Recursion**: A recursive function is said to be following Tail Recursion if it invokes itself at the end of the function. That is, if all of the statements are executed before the function invokes itself then it is said to be following Tail Recursion.  
  
**example;**

// This is a Tail Recursion

void printN(int N)

{

if(N==0)

return;

else

cout<<N<<" ";

printN(N-1);

}

The above function call for **N = 5**will print:

5 4 3 2 1

**Which one is Better-Tail Recursive or Non-Tail-Recursive?**

The tail-recursive functions are considered better than non-tail recursive functions as tail-recursion can be optimized by the compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function’s stack frame is of no use.

**Can a non-tail recursive function be written as tail-recursive to optimize it?**

Consider the following function to calculate factorial of N. Although it looks like Tail Recursive at first look, it is a non-tail-recursive function. If we take a closer look, we can see that the value returned by **fact(N-1)** is used in **fact(N)**, so the call to fact(N-1) is not the last thing done by fact(N).

int fact(int N)   
{   
 if (N == 0)   
 return 1;   
   
 return N\*fact(N-1);   
}

The above function can be written as a Tail Recursive function. The idea is to use one more argument and accumulate the factorial value in the second argument. When N reaches 0, return the accumulated value.

int factTR(int N, int a)   
{   
 if (N == 0)   
 return a;   
   
 return factTR(N-1, N\*a);   
}

**subset generation problem:**

Given a set represented as a string, write a recursive code to print all the subsets of it. The subsets can be printed in any order.  
  
**Examples:**

**Input** : set = "abc"  
**Output** : "". "a", "b", "c", "ab", "ac", "bc", "abc"  
  
**Input** : set = "abcd"  
**Output** : "" "a" "ab" "abc" "abcd" "abd" "ac" "acd"  
 "ad" "b" "bc" "bcd" "bd" "c" "cd" "d"

The idea is to consider two cases for every character. (i) Consider current character as part of the current subset (ii) Do not consider current character as part of the current subset.

// CPP program to generate power set

#include <bits/stdc++.h>

using namespace std;

// str : Stores input string

// curr : Stores current subset

// index : Index in current subset, curr

void powerSet(string str, int index = 0,

string curr = "")

{

int n = str.length();

// base case

if (index == n) {

cout << curr << endl;

return;

}

// Two cases for every character

// (i) We consider the character

// as part of current subset

// (ii) We do not consider current

// character as part of current

// subset

powerSet(str, index + 1, curr + str[index]);

powerSet(str, index + 1, curr);

}

// Driver code

int main()

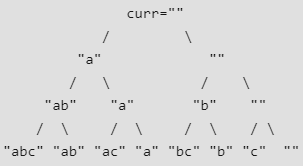
Run

Java

**Output:**

abc  
ab  
ac  
a  
bc  
b  
c

Let us understand the recursion with an example "abc". Every node in the below tree represents the string **curr**.



At root, index = 0.  
At next level of tree index = 1  
At third level, index = 2  
At fourth level index = 3 (becomes equal to string length), so we print the subset.

**JOSEPHUS PROBLEM:**

There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number k which indicates that k-1 persons are skipped and kth person is killed in a circle. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.  
  
  
For example, if n = 5 and k = 2, then the safe position is 3. Firstly, the person at position 2 is killed, then the person at position 4 is killed, then the person at position 1 is killed. Finally, the person at position 5 is killed. So the person at position 3 survives.  
If n = 7 and k = 3, then the safe position is 4. The persons at positions 3, 6, 2, 7, 5, 1 are killed in order, and the person at position 4 survives.  
  
  
  
The problem has following recursive structure.

josephus(n, k) = (josephus(n - 1, k) + k-1) % n + 1  
 josephus(1, k) = 1

**How does this recursion work?** When we kill k-th person, n-1 persons are left, but numbering starts from k+1 and goes in modular way.  
  
(k+1)-th person in the original circle is now the first person.  
n-th person in the original circle is now (n-k)-th person.  
1-st person in the original circle is now (n-k+1)-th person.  
(k-1)-th person in the original circle is now (n-1)-th person.  
  
So we add (k-1) to the returned position to handle all cases and keep the modulo under n. Finally, we add 1 to the result.  
  
This solution is not easy to think at the first moment.  
  
A simple solution that comes to our mind is  
  
(josephus(n-1, k) + k) % n  
  
We add k because we shifted the positions by k after the first killing. The problem with the above solution is that the value of (josephus(n-1, k) + k) can become n and overall solution can become 0. But positions are from 1 to n. To ensure that, we never get n, we subtract 1 and add 1 later. This is how we get  
  
(josephus(n-1, k) + k - 1) % n + 1  
  
Following is a simple recursive implementation of the Josephus problem. The implementation simply follows the recursive structure mentioned above.

#include <stdio.h>

int josephus(int n, int k)

{

if (n == 1)

return 1;

else

/\* The position returned by josephus(n - 1, k)

is adjusted because the recursive call

josephus(n - 1, k) considers the original

position k%n + 1 as position 1 \*/

return (josephus(n - 1, k) + k-1) % n + 1;

}

// Driver Program to test above function

int main()

{

int n = 14;

int k = 2;

printf("The chosen place is %d",

josephus(n, k));

return 0;

}

Time Complexity: O(n)

**PERMUTATION OF STRING:**

Given a string, print all permutations of it.

**Input** : str = "ABC"  
**Output** : ABC ACB BAC BCA CAB CBA

**Idea:** We iterate from first to last index. For every index i, we swap the i-th character with the first index. This is how we fix characters at the current first index, then we recursively generate all permutations beginning with fixed characters (by parent recursive calls). After we have recursively generated all permutations with the first character fixed, then we move the first character back to its original position so that we can get the original string back and fix the next character at first position.

**Illustration:**We swap 'A' with 'A'. Then we recursively generate all permutations beginning with A. While returning from the recursive calls, we revert the changes made by them using the same swap again. So we get the original string "ABC".  
Then we swap 'A' with 'B' and generate all permutations beginning with 'B'.  
Similarly, we generate all permutations beginning with 'C'  
  
[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/NewPermutation.gif)

// C++ program to print all

// permutations with duplicates allowed

#include <bits/stdc++.h>

using namespace std;

/\* Function to print permutations of string

This function takes three parameters:

1. String

2. Starting index of the string

3. Ending index of the string. \*/

void permute(string &str, int l, int r)

{

if (l == r)

cout << str << " ";

else

{

for (int i = l; i <= r; i++)

{

swap(str[l], str[i]);

permute(str, l+1, r);

swap(str[l], str[i]);

}

}

}

/\* Driver program to test above functions \*/

int main()

{

string str = "ABC";

permute(str, 0, str.length()-1);

**Output:**

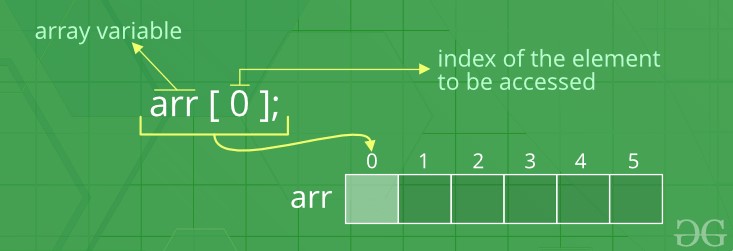
ABC ACB BAC BCA CBA CAB

**Chapter five: arrays**

**introduction**

An array is a collection of items of the same data type stored at contiguous memory locations. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).  
  
For simplicity, we can think of an array as a fleet of stairs where on each step a value is placed (let’s say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step that they are on.  
  
**Remember**: “Location of the next index depends on the data type that we use”.  
  
The above image can be looked at as a top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by their index in the array (in a similar way where you could identify your friends by the step on which they were on in the above example).  
  
**Defining an Array**: Array definition is similar to defining any other variable. There are two things that are needed to be kept in mind, **the data type of the array elements** and the **size** of the array. The size of the array is fixed and the memory for an array needs to be allocated before use, the size of an array cannot be increased or decreased dynamically.  
  
Generally, arrays are declared as:

**dataType arrayName[arraySize];**  
  
An array is distinguished from a normal variable   
by brackets [ and ].

**Accessing array elements**: Arrays allows to access elements randomly. Elements in an array can be accessed using indexes. Suppose an array named **arr** stores N elements. Indexes in an array are in the range of **0 to N-1**, where the first element is present at 0-th index and consecutive elements are placed at consecutive indexes. Element present at ith index in the array **arr[]** can be accessed as arr[i].  
  
The below image shows an array **arr[]** of size 5:  
  
  
**Advantages of using arrays:**

* Arrays allow random access of elements. This makes accessing elements by their position faster.
* Arrays have better [cache locality](https://en.wikipedia.org/wiki/Locality_of_reference" \t "_blank" \o "cache locality) that can make a pretty big difference in performance.

**Examples**:

// A character array in C/C++/Java

char arr1[] = {'g', 'e', 'e', 'k', 's'};

// An Integer array in C/C++/Java

int arr2[] = {10, 20, 30, 40, 50};

// Item at i'th index in array is typically accessed

// as "arr[i]". For example arr1[0] gives us 'g'

// and arr2[3] gives us 40.

**Searching in an Array**

Searching for an element in an array means to check if a given element is present in the array or not. This can be done by accessing elements of the array one by one starting from the first element and checking whether any of the elements matches with the given element.  
  
We can use [loops](https://www.geeksforgeeks.org/loops-in-c-and-cpp/) to perform the above operation of array traversal and access the elements, using indexes.  
  
Suppose the array is named **arr[**] with size **N** and the element to be searched is referred to as **key**. Below is the algorithm to perform the search operation in the given array.

for(i = 0; i < N; i++)  
{  
 if(arr[i] == key)  
 {   
 print "Element Found";  
 }  
 else  
 {  
 print "Element not Found";  
 }  
}

**Time Complexity** of this search operation will be O(N) in the worst case as we are checking every element of the array from 1st to last, so the number of operations is N.

**Insertion and deletion in array:**

Insertion in Arrays

Given an array of a given size. The task is to insert a new element in this array. There are two possible ways of inserting elements in an array:

1. Insert elements at the end of the array.
2. Insert an element at any given index in the array.

**Special Case**: A special case is needed to be considered is whether the array is already full or not. If the array is full, then the new element can not be inserted.

Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len*number of elements already present in this array.

* **Insert an element K at the end in arr[]**: The first step is to check if there is any space left in the array for the new element. To do this check,

if(len < N)  
 // space left  
else  
 // array is full

If there is space left for the new element, insert it directly at the end at position **len + 1** and index **len**:

arr[len] = k;

***Time Complexity*** of this insert operation is constant, i.e. O(1) as we are directly inserting the element in a single operation.

* **Insert an element K at position, pos in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,

if(len < N)  
 // space left  
else  
 // array is full

Now, if there is space left, the element can be inserted. The index of the new element will be **idx = pos - 1**.  
  
Now, before inserting the element at the index idx, shift all elements from the index idx till end of the array to the right by 1 place. This can be done as:

for(i = len-1; i >= idx; i--)  
{  
 arr[i+1] = arr[i];  
}

After shifting the elements, place the element K at index idx.

arr[idx] = K;

***Time Complexity*** in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the right.

Deletion in Arrays

To delete a given element from an array, we will have to first search the element in the array. If the element is present in the array then delete operation is performed for the element otherwise the user is notified that the array does not contains the given element.  
  
Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len*number of elements already present in this array.  
  
**Deleting an element K from the array arr[]**: Search the element K in the array arr[] to find the index at which it is present.

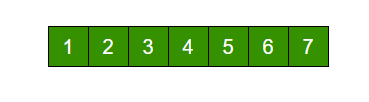
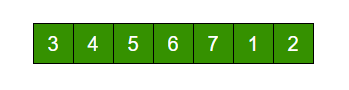
for(i = 0; i < N; i++)  
{  
 if(arr[i] == K)  
 idx = i; return;  
 else  
 Element not Found;  
}

Now, to delete the element present at index **idx**, left shift all of the elements present after *idx* by one place and finally reduce the length of the array by 1.

for(i = idx+1; i < len; i++)  
{  
 arr[i-1] = arr[i];  
}  
  
len = len-1;

***Time Complexity*** in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the left.

**Array rotation:**

As the term rotation signifies, array rotation means to rotate the elements of an array by given positions.  
  
Consider the below array:  
  
The above array is rotated counter-clockwise(towards left) by 2 elements. After rotation, the array will be:  
  
  
Visually, the process of counter clock-wise array rotation(rotated by say K elements) looks like:

* Shift all elements after K-th element to the left by K positions.
* Fill the K blank spaces at the end of the array by first K elements from the original array.

**Note**: The similar approach can also be applied for clockwise array rotation.

Implementations

* **Simple Method**: The simplest way to rotate an array is to implement the above visually observed approach by using extra space.  
  1. Store the first K elements in a temporary array say temp[].
  2. Shift all elements after K-th element to the left by K positions in the original array.
  3. Fill the K blank spaces at the end of the original array by the K elements from the temp array.

Say, arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2  
1) Store first K elements in a temp array  
 temp[] = [1, 2]  
2) Shift rest of the arr[]  
 arr[] = [3, 4, 5, 6, 7, 6, 7]  
3) Store back the K elements from temp  
 arr[] = [3, 4, 5, 6, 7, 1, 2]

***Time Complexity:*** O(N), where N is the number of elements in the array.  
***Auxiliary Space:*** O(K) where K is the number of places by which elements will be rotated.

* **Another Method (Without extra space)**: We can also rotate an array by avoiding the use of temporary array. The idea is to rotate the array one by one K times.

leftRotate(arr[], d, n)  
start  
 For i = 0 to i < d  
 Left rotate all elements of arr[] by one  
end

To rotate an array by 1 position to the left:

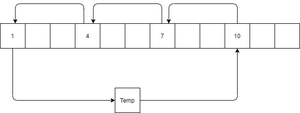
* 1. Store the first element in a temporary variable say temp.
  2. Left shift all elements after the first element by 1 position. That is, move arr[1] to arr[0], arr[2] to arr[1] and so on.
  3. Initialize arr[N-1] with temp.

**To rotate an array by K position to the left, repeat the above process K times.**  
Take the same example,

arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2  
  
**Rotate arr[] one by one 2 times.**  
  
After 1st rotation: [2, 3, 4, 5, 6, 7, 1]  
After 2nd rotation: [ 3, 4, 5, 6, 7, 1, 2]

***Time Complexity:*** O(N\*K), where N is the number of elements in the array and K is the number of places by which elements will be rotated.  
***Auxiliary Space:*** O(1).

* **Juggling Algorithm**: This is an extension of the above method. Instead of moving one by one, divide the array in different sets, where number of sets is equal to GCD of N and K and move the elements within sets.  
    
  If GCD is 1 as is for the above example array (N = 7 and K = 2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[I+d] to arr[I] and finally store temp at the right place.  
    
  Here is an example for N = 12 and K = 3. GCD of N and K is 3:

Let arr[] be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}  
  
a) Elements are first moved in first set – (See below   
 diagram for this movement)  
  
  
  
 arr[] after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}  
  
b) Then in second set.  
 arr[] after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}  
  
c) Finally in third set.  
 arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}

***Time Complexity:*** O(N), where N is the number of elements in the array.  
***Auxiliary Space:*** O(1).

**Reversing an array:**

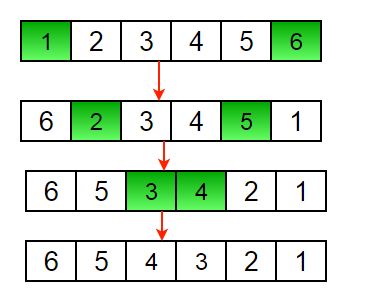
Reversing an array means reversing the order of elements in the given array.  
  
**Problem**: Given an array of N elements. The task is to reverse the order of elements in the given array.  
  
**For Example**:

**Input** : arr[] = {1, 2, 3}  
**Output** : arr[] = {3, 2, 1}  
  
**Input** : arr[] = {4, 5, 1, 2}  
**Output** : arr[] = {2, 1, 5, 4}

**Iterative Solution**

* **Method 1 (Using Temporary Array)**: The idea is to first copy all of the elements of the given array in a temporary array. Then traverse the temporary array from end and replace elements in original array by elements of temp array.  
    
  This method will take extra space in order of O(N).
* **Method 2 (Efficient)**: This method is efficient then the above method and avoids using extra spaces. The idea is to traverse the array from both ends and keep swapping elements from both ends until middle of the array is reached.

1) Initialize start and end indexes as   
 start = 0, end = N-1  
2) In a loop, swap arr[start] with arr[end]   
 and change start and end as follows :  
 start = start + 1,   
 end = end – 1

For Example:  
  
**Time Complexity**: O(N), where N is the number of elements in the array.

**Recursive Solution**

The recursive approach is almost similar to that of the method 2 of the iterative solution. Below is the recursive algorithm to reverse an array:

1) Initialize start and end indexes as   
 start = 0, end = n-1  
2) Swap arr[start] with arr[end]  
3) Recursively call reverse for rest of the array.

Below is the recursive function to reverse an array:

void rvereseArray(arr[], start, end)

{

if (start >= end)

return;

// Swap elements at start and end

temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

// Recursive Function calling

rvereseArray(arr, start + 1, end - 1);

}

**Time Complexity**: O(N), where N is the number of elements in the array.

**Sliding window technique:**

This technique shows how a nested for loop in few problems can be converted to single for loop and hence reducing the time complexity.  
  
Let’s start with a problem for illustration where we can apply this technique:

Given an array of integers of size 'n'.  
Our aim is to calculate the maximum sum of 'k'   
consecutive elements in the array.  
  
Input : arr[] = {100, 200, 300, 400}  
 k = 2  
Output : 700  
  
Input : arr[] = {1, 4, 2, 10, 23, 3, 1, 0, 20}  
 k = 4   
Output : 39  
We get maximum sum by adding subarray {4, 2, 10, 23}  
of size 4.  
  
Input : arr[] = {2, 3}  
 k = 3  
Output : Invalid  
There is no subarray of size 3 as size of whole  
array is 2.

The **Naive Approach** to solve this problem is to calculate sum for each of the blocks of K consecutive elements and compare which block has the maximum sum possible. The time complexity of this approach will be O(n \* k).

**Window Sliding Technique**

The above problem can be solved in Linear Time Complexity by using Window Sliding Technique by avoiding the overhead of calculating sum repeatedly for each block of k elements.  
  
The technique can be best understood with the window pane in bus, consider a window of length **n** and the pane which is fixed in it of length **k**. Consider, initially the pane is at extreme left i.e., at 0 units from the left. Now, co-relate the window with array arr[] of size n and plane with current\_sum of size k elements. Now, if we apply force on the window such that it moves a unit distance ahead. The pane will cover next **k** consecutive elements.  
  
Consider an array **arr[]** = {5 , 2 , -1 , 0 , 3} and value of **k** = 3 and **n** = 5  
  
**Applying sliding window technique**:

1. We compute the sum of first k elements out of n terms using a linear loop and store the sum in variable window\_sum.
2. Then we will graze linearly over the array till it reaches the end and simultaneously keep track of maximum sum.
3. To get the current sum of block of k elements just subtract the first element from the previous block and add the last element of the current block .

The below representation will make it clear how the window slides over the array.  
  
This is the initial phase where we have calculated the initial window sum starting from index 0 . At this stage the window sum is 6. Now, we set the maximum\_sum as current\_window i.e 6.  
  
  
Now, we slide our window by a unit index. Therefore, now it discards 5 from the window and adds 0 to the window. Hence, we will get our new window sum by subtracting 5 and then adding 0 to it. So, our window sum now becomes 1. Now, we will compare this window sum with the maximum\_sum. As it is smaller we wont the change the maximum\_sum.  
  
  
Similarly, now once again we slide our window by a unit index and obtain the new window sum to be 2. Again we check if this current window sum is greater than the maximum\_sum till now. Once, again it is smaller so we don't change the maximum\_sum.  
  
Therefore, for the above array our maximum\_sum is 6.  
  


**Prefix sum array:**

Given an array arr[] of size N, the task is to generate the *prefix sum array* of the given array.  
  
**Prefix Sum Array**: The prefix sum array of any array, arr[] is defined as an array of same size say, prefixSum[] such that the value at any index **i** in prefixSum[] is sum of all elements from indexes **0 to i** in arr[].  
That is,

prefixSum[i] = arr[0] + arr[1] + arr[2] + . . . . + arr[i]  
  
for all **0 <= i <= N**.

**Examples**:

**Input** : arr[] = {10, 20, 10, 5, 15}  
**Output** : prefixSum[] = {10, 30, 40, 45, 60}  
  
**Explanation** : While traversing the array, update   
the element by adding it with its previous element.  
prefixSum[0] = 10,   
prefixSum[1] = prefixSum[0] + arr[1] = 30,   
prefixSum[2] = prefixSum[1] + arr[2] = 40 and so on.

Below function generates a prefix sum array for a given array arr[] of size N:

void fillPrefixSum(int arr[], int N, int prefixSum[])

{

prefixSum[0] = arr[0];

// Adding present element

// with previous element

for (int i = 1; i < N; i++)

prefixSum[i] = prefixSum[i-1] + arr[i];

}

**Finding sum in a Range**: We can easily calculate the sum with-in a range [i, j] in an array using the prefix sum array. Since the array prefixSum[i] stores the sum of all elements upto **i**. Therefore, **prefixSum[j] - prefixSum[i]** will give:

sum of elements upto j-th index - sum of elements upto i-th element

The above total sum will exclude the i-th element.  
Therefore, we can get the sum of elements in range [i,j] by:

prefixSum[j] - prefixSum[i-1]

The above formula will not work in the case when **i = 0**.  
Therefore,

sumInRange = prefixSum[j] , if i = 0  
otherwise,  
sumInRange = prefixSum[j] - prefixSum[i-1] , if (i != 0).

**Sample Problem**: Consider an array of size N with all initial values as 0. Perform given 'm' add operations from index 'a' to 'b' and evaluate highest element in array. An add operation adds 100 to all elements from index a to b (both inclusive).  
  
**Example**:

**Input** : n = 5 // We consider array {0, 0, 0, 0, 0}  
 m = 3.  
 a = 2, b = 4.  
 a = 1, b = 3.  
 a = 1, b = 2.  
**Output** : 300  
  
**Explanation** :   
After I operation -  
A : 0 100 100 100 0  
  
After II operation -  
A : 100 200 200 100 0  
  
After III operation -  
A : 200 300 200 100 0  
  
Highest element : 300

***Solution using Prefix Sum***:

1 : Run a loop for 'm' times, inputting 'a' and 'b'.  
2 : Add 100 at index 'a' and subtract 100 from index 'b+1'.  
3 : After completion of 'm' operations, compute the prefix sum array.  
4 : Scan the largest element and we're done.

What we did was adding 100 at ‘a’ because this will add 100 to all elements while taking prefix sum array. Subtracting 100 from ‘b+1’ will reverse the changes made by adding 100 to elements from ‘b’ onward.

For better understanding :

After I operation -  
A : 0 100 0 0 -100   
Prefix Sum Array : 0 100 100 100 0  
  
After II operation -  
A : 100 100 0 -100 -100  
Prefix Sum Array : 100 200 200 100 0  
  
After III operation -  
A : 200 100 -100 -100 -100  
Prefix Sum Array : 200 300 200 100 0  
  
Final Prefix Sum Array : 200 300 200 100 0   
  
The required highest element : 300

**Implementing arrays using STL :**

We already have discussed the basic declaration of arrays. Arrays can also be implemented using some built-in classes available in the C++ Standard Template Library.  
  
Some of the most commonly used classes for implementing sequential lists or arrays are:

* Vector
* List

Let's look at each of these classes in details.

Vector

Vector in C++ STL is a class that represents a dynamic array. The advantages of vector over normal arrays are,

* We do not need to pass size as an extra parameter when we pass vector.
* Vectors have many in-built functions for erasing an element, inserting an element etc.
* Vectors support dynamic sizes, we do not have to initially specify the size of a vector. We can also resize a vector.
* There are many other functionalities vector provide.

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.  
  
To use the Vector class, include the below header file in your program:

**#include< vector >**

**Declaring Vector**:

vector< *Type\_of\_element* > *vector\_name*;  
  
Here, *Type\_of\_element* can be any valid C++ data type,  
or can be any other container also like Pair, List etc.

Some important and commonly used functions of Vector class are:

* **begin()** – Returns an iterator pointing to the first element in the vector.
* **end()** – Returns an iterator pointing to the theoretical element that follows the last element in the vector.
* **size()** – Returns the number of elements in the vector.
* **capacity()** – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
* **empty()** – Returns whether the container is empty.
* **push\_back()** – It push the elements into a vector from the back.
* **pop\_back()**– It is used to pop or remove elements from a vector from the back.
* **insert()** – It inserts new elements before the element at the specified position.
* **erase()** – It is used to remove elements from a container from the specified position or range.
* **swap()** – It is used to swap the contents of one vector with another vector of same type and size.
* **clear()** – It is used to remove all the elements of the vector container.
* **emplace()** – It extends the container by inserting new element at position.
* **emplace\_back()** – It is used to insert a new element into the vector container, the new element is added to the end of the vector.

Below program illustrate the above methods:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> v;

// Push elements

for (int i = 1; i <= 5; i++)

v.push\_back(i);

cout << "Size : " << v.size();

// checks if the vector is empty or not

if (v.empty() == false)

cout << "\nVector is not empty";

else

cout << "\nVector is empty";

cout << "\nOutput of begin and end: ";

for (auto i = v.begin(); i != v.end(); ++i)

cout << \*i << " ";

// inserts at the beginning

v.emplace(v.begin(), 5);

cout << "\nThe first element is: " << v[0];

Run

**Output**:

Size : 5  
Vector is not empty  
Output of begin and end: 1 2 3 4 5   
The first element is: 5  
The last element is: 20  
Vector size after erase(): 0

--------------------------------------------------------------------------------------------------------------------------

List

Lists are sequence containers that allow non-contiguous memory allocation. List in C++ STL implements a doubly linked list and not arrays. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about doubly linked lists. For implementing a singly linked list, we can use **forward\_list** class in C++ STL.  
  
To use the List class, include the below header file in your program:

**#include< list >**

**Declaring List**:

list< *Type\_of\_element* > *list\_name*;  
  
Here, *Type\_of\_element* can be any valid C++ data type,  
or can be any other container also like Pair, List etc.

Some important and commonly used functions of List are:

* **front()** – Returns the value of the first element in the list.
* **back()** – Returns the value of the last element in the list.
* **push\_front(g)** – Adds a new element ‘g’ at the beginning of the list.
* **push\_back(g)** – Adds a new element ‘g’ at the end of the list.
* **pop\_front()** – Removes the first element of the list, and reduces the size of the list by 1.
* **pop\_back()** – Removes the last element of the list, and reduces the size of the list by 1.
* **begin()** and **end()** – begin() function returns an iterator pointing to the first element of the list.
* **empty()** – Returns whether the list is empty(1) or not(0).
* **insert()** – Inserts new elements in the list before the element at a specified position.
* **reverse()** – Reverses the list.
* **size()** – Returns the number of elements in the list.
* **sort()** – Sorts the list in increasing order.

Below program illustrate the above functions:

#include <iostream>

#include <list>

#include <iterator>

using namespace std;

//function for printing the elements in a list

void showlist(list <int> g)

{

list <int> :: iterator it;

for(it = g.begin(); it != g.end(); ++it)

cout << '\t' << \*it;

cout << '\n';

}

int main()

{

list <int> gqlist1, gqlist2;

for (int i = 0; i < 10; ++i)

{

gqlist1.push\_back(i \* 2);

gqlist2.push\_front(i \* 3);

}

cout << "\nList 1 (gqlist1) is : ";

showlist(gqlist1);

cout << "\nList 2 (gqlist2) is : ";

showlist(gqlist2);

Run

**Output**:

List 1 (gqlist1) is : 0 2 4 6   
8 10 12 14 16 18  
  
List 2 (gqlist2) is : 27 24 21 18   
15 12 9 6 3 0  
  
gqlist1.front() : 0  
gqlist1.back() : 18  
gqlist1.pop\_front() : 2 4 6 8   
10 12 14 16 18  
  
gqlist2.pop\_back() : 27 24 21 18   
15 12 9 6 3  
  
gqlist1.reverse() : 18 16 14 12   
10 8 6 4 2  
  
gqlist2.sort(): 3 6 9 12   
15 18 21 24 27

**Iterators in c++ STL:**

Iterators are used to point at the memory addresses of [STL](http://quiz.geeksforgeeks.org/the-c-standard-template-library-stl/" \t "_blank) containers. They are primarily used in a sequence of numbers, characters etc. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access content at that particular location using them.  
  
**Basic Operations of iterators** :-

* **begin()** :- This function is used to return the **beginning position** of the container.
* **end()** :- This function is used to return the***after* end position** of the container.
* // C++ code to demonstrate the working of

// iterator, begin() and end()

#include<iostream>

#include<iterator> // for iterators

#include<vector> // for vectors

using namespace std;

int main()

{

vector<int> ar = { 1, 2, 3, 4, 5 };

// Declaring iterator to a vector

vector<int>::iterator ptr;

// Displaying vector elements using begin() and end()

cout << "The vector elements are : ";

for (ptr = ar.begin(); ptr < ar.end(); ptr++)

cout << \*ptr << " ";

return 0;

}

Run

**Output:**

The vector elements are : 1 2 3 4 5

* **advance()**:- This function is used to **increment the iterator position**till the specified number mentioned in its arguments.  
  // C++ code to demonstrate the working of

// advance()

#include<iostream>

#include<iterator> // for iterators

#include<vector> // for vectors

using namespace std;

int main()

{

vector<int> ar = { 1, 2, 3, 4, 5 };

// Declaring iterator to a vector

vector<int>::iterator ptr = ar.begin();

// Using advance() to increment iterator position

// points to 4

advance(ptr, 3);

// Displaying iterator position

cout << "The position of iterator after advancing is : ";

cout << \*ptr << " ";

return 0;

}

Run

**Output:**

The position of iterator after advancing is : 4

* **next()** :- This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.
* **prev()** :- This function **returns the new iterator** that the iterator would point **after decrementing the positions** mentioned in its arguments.  
  // C++ code to demonstrate the working of

// next() and prev()

#include<iostream>

#include<iterator> // for iterators

#include<vector> // for vectors

using namespace std;

int main()

{

vector<int> ar = { 1, 2, 3, 4, 5 };

// Declaring iterators to a vector

vector<int>::iterator ptr = ar.begin();

vector<int>::iterator ftr = ar.end();

// Using next() to return new iterator

// points to 4

auto it = next(ptr, 3);

// Using prev() to return new iterator

// points to 3

auto it1 = prev(ftr, 3);

// Displaying iterator position

cout << "The position of new iterator using next() is : ";

cout << \*it << " ";

cout << endl;

Run

**Output:**

The position of new iterator using next() is : 4   
The position of new iterator using prev() is : 3

* **inserter()** :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted**.  
  // C++ code to demonstrate the working of

// inserter()

#include<iostream>

#include<iterator> // for iterators

#include<vector> // for vectors

using namespace std;

int main()

{

vector<int> ar = { 1, 2, 3, 4, 5 };

vector<int> ar1 = {10, 20, 30};

// Declaring iterator to a vector

vector<int>::iterator ptr = ar.begin();

// Using advance to set position

advance(ptr, 3);

// copying 1 vector elements in other using inserter()

// inserts ar1 after 3rd position in ar

copy(ar1.begin(), ar1.end(), inserter(ar,ptr));

// Displaying new vector elements

cout << "The new vector after inserting elements is : ";

for (int &x : ar)

cout << x << " ";

return 0;

Run

**Output:**

The new vector after inserting elements is : 1 2 3 10 20 30 4 5

**Problems in array:**

### ****Problem #1 : Range Sum Queries using Prefix Sum****

**Description :** We are given an Array of **n** integers, We are given **q** queries having indices **l and r**. We have to find out sum between the given range of indices.

**Input**   
[4, 5, 3, 2, 5]  
3  
0 3  
2 4  
1 3  
**Output**  
14 (4+5+3+2)  
10 (3+2+5)  
10 (5+3+2)

**Solution :** The numbers of queries are large. It will be very inefficient to iterate over the array and calculate the sum for each query separately. We have to devise the solution so that we can get the answer of the query in constant time. We will be storing the sum upto a particular index in prefix sum Array. We will be using the prefix sum array to calculate the sum for the given range.

prefix[] = Array stores the sum (A[0]+A[1]+....A[i]) at index i.  
if l == 0 :  
 sum(l,r) = prefix[r]  
else :  
 sum(l,r) = prefix[r] - prefix[l-1]

**Pseudo Code**

// n : size of array  
// q : Number of queries  
// l, r : Finding Sum of range between index l and r   
// l and r (inclusive) and 0 based indexing  
void range\_sum(arr, n)  
{  
 prefix[n] = {0}  
 prefix[0] = arr[0]  
 for i = 1 to n-1 :  
 prefix[i] = a[i] + prefix[i-1]  
  
 for (i = 1 to q )  
 {  
 if (l == 0)   
 {  
 ans = prefix[r]  
 print(ans)  
 }  
 else   
 {  
 ans = prefix[r] - prefix[l-1]  
 print(ans)  
 }  
 }  
}

**Time Complexity :** Max(O(n),O(q))  
**Auxiliary Space :** O(n)

### ****Problem #2 : Equilibrium index of an array****

**Description -** Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. We are given an Array of integers, We have to find out the first index **i** from left such that -

A[0] + A[1] + ... A[i-1] = A[i+1] + A[i+2] ... A[n-1]

**Input**  
[-7, 1, 5, 2, -4, 3, 0]  
**Output**  
3  
A[0] + A[1] + A[2] = A[4] + A[5] + A[6]

**Naive Solution :** We can iterate for each index i and calculate the leftsum and rightsum and check whether they are equal.

for (i=0 to n-1)  
{  
 leftsum = 0  
 for (j = 0 to i-1)  
 leftsum += arr[i]  
 rightsum = 0  
 for (j = i+1 to n-1)  
 rightsum += arr[i]  
  
 if leftsum == rightsum :  
 return i  
}

**Time Complexity :** O(n^2)  
**Auxiliary Space :** O(1)

**Tricky Solution :** The idea is to first get the total sum of array. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get the right sum by subtracting the elements one by one. Then check whether the Leftsum and the Rightsum are equal.  
**Pseudo Code**

// n : size of array  
int eqindex(arr, n)  
{  
 sum = 0  
 leftsum = 0  
 for (i=0 to n-1)  
 sum += arr[i]  
  
 for (i=0 to n-1)  
 {  
 // now sum will be righsum for index i  
 sum -= a[i]  
 if (sum == leftsum )  
 return i  
 leftsum += a[i]  
 }  
}

**Time Complexity :** O(n)  
**Auxiliary Space :** O(1)

### ****Problem #3 : Largest Sum Subarray****

**Description :** We are given an array of positive and negative integers. We have to find the subarray having maximum sum.

**Input**  
[-3, 4, -1, -2, 1, 5]  
**Output**  
7  
(4+(-1)+(-2)+1+5)

**Solution :**A simple idea is to look for all the positive contiguous segments of the array (max\_ending\_here is used for this), and keep the track of maximum sum contiguous segment among all the positive segments (max\_so\_far is used for this). Each time we get a positive sum compare it with max\_so\_far and if it is greater than max\_so\_far, update max\_so\_far.  
**Pseudo Code**

//n : size of array  
int largestsum(arr, n)  
{  
 max\_so\_far = INT\_MIN  
 max\_ending\_here = 0  
  
 for (i=0 to n-1)  
 {  
 max\_ending\_here += arr[i]  
 if max\_so\_far < max\_ending\_here :  
 max\_so\_far = max\_ending\_here  
  
 if max\_ending\_here < 0 :  
 max\_ending\_here = 0  
 }  
  
 return max\_so\_far  
}

**Time Complexity :** O(n)  
**Auxiliary Space :** O(1)

### ****Problem #4 : Merge two sorted Arrays****

**Description :** We are given two sorted arrays **arr1[ ]** and **arr2[ ]**of size **m** and **n** respectively. We have to merge these arrays and store the numbers in arr3[ ] of size **m+n**.

**Input**  
1 3 4 6  
2 5 7 8  
**Output**  
1 2 3 4 5 6 7 8

**Solution :** The idea is to traverse both the arrays simultaneously and compare the current numbers from both the Arrays. Pick the smaller element and copy it to arr3[ ] and advance the current index of the array from where the smaller element is picked. When we reach at the end of one of the arrays, copy the remaining elements of another array to arr3[ ].  
**Pseudo Code**

// input arrays - arr1(size m), arr2(size n)  
void merge\_sorted(arr1, arr2, m, n)  
{  
 arr3[m+n] // merged array  
 i=0,j=0,k=0  
 while(i < m && j < n)   
 {  
 if arr1[i] < arr2[j] :  
 arr3[k++] = arr1[i++]  
 else :  
 arr3[k++] = arr2[j++]  
 }  
 while(i < m)  
 arr3[k++] = arr1[i++]  
 while(j < n)  
 arr3[k++] = arr2[j++]  
}

**Time Complexity :** O(m+n)  
**Auxiliary Space :** O(m+n)

**XoR linked list – a memory efficient doubly linked list | set 1**

1An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. It is represented as follows in the image below. From the below image, it can be depicted out that the address of the previous node is retained and carried over for computation by the previous pointer while that of the next node is after pointers similarly.

 Now there is a memory-efficient version of Doubly Linked List that can be created using only one space for the address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.

Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Efficient) representations of the Doubly Linked List. 

Now here we will be discussing out both the ways in order to perch out how XOR representation behaves differently from ordinary representation.

1. Ordinary Representation
2. XOR List Representation

**Way 1: Ordinary Representation**

Node A:

prev = NULL, next = add(B) // previous is NULL and next is address of B

Node B:

prev = add(A), next = add(C) // previous is address of A and next is address of C

Node C:

prev = add(B), next = add(D) // previous is address of B and next is address of D

Node D:

prev = add(C), next = NULL // previous is address of C and next is NULL

**Way 2: XOR List Representation**

Let us call the address variable in XOR representation npx (XOR of next and previous)

while traversing XOR Linked List we can traverse the XOR list in both forward and reverse directions. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address.

For example: When we are at node C, we must have the address of B. XOR of add(B) and *npx*of C gives us the add(D).

**Illustration:**

Node A:

npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:

npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C

Node C: 

npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D

Node D:

npx = add(C) XOR 0 // bitwise XOR of address of C and 0

npx(C) XOR add(B)

=> (add(B) XOR add(D)) XOR add(B) // npx(C) = add(B) XOR add(D)

=> add(B) XOR add(D) XOR add(B) // a^b = b^a and (a^b)^c = a^(b^c)

=> add(D) XOR 0 // a^a = 0

=> add(D) // a^0 = a

Similarly, we can traverse the list in the backward direction. Now straightaway coming down to the implementation part in order to figure out better.

Below is the implementation of the above approach:

// C++ Implementation of Memory

// efficient Doubly Linked List

// Importing libraries

#include <bits/stdc++.h>

#include <cinttypes>

using namespace std;

// Class 1

// Helper class(Node structure)

class Node {

public : int data;

// Xor of next node and previous node

Node\* xnode;

};

// Method 1

// It returns Xored value of the node addresses

Node\* Xor(Node\* x, Node\* y)

{

return reinterpret\_cast<Node\*>(

reinterpret\_cast<uintptr\_t>(x)

^ reinterpret\_cast<uintptr\_t>(y));

}

// Method 2

// Insert a node at the start of the Xored LinkedList and

// mark the newly inserted node as head

void insert(Node\*\* head\_ref, int data)

{

Run

**Output**

The nodes of Linked List are:

10000 1000 100 10

**Time Complexity:** O(n)  
**Auxiliary Space:**O(1)