**1.Sieve of Eratosthenes**

**Concept:**

Sieve is a technique to pre-compute Prime Numbers. It is done by marking all the multiples of the prime numbers as composite numbers, thus the unmarked once remain as primes.

The Sieve of Eratosthenes is one of the most efficient ways to find all of the smaller primes.

Its Time Complexity is O( N\*log(N)\*log(log(N)))

**Algorithm:**

1. Assume all numbers initially as prime ( Prime[0]=Prime[1]…Prime[N]=1 )

2. Mark Prime[0]=Prime[1]=0

3. From i=2 to Sqrt(Max) mark all the multiples of a number i=x , where Prime[x]=1

4. Hence in the Boolean Array, if Prime[x]=1, then x is a Prime Number.

**Function:**

void Sieve()

{

memset(Prime,1,sizeof(Prime));

int i,j;

Prime[0]=Prime[1]=0;

for(i=2;i<sqrt(N);i++)

{

if(Prime[i])

{

for(j=i\*i;j<=N;j+=i)

Prime[j]=0;

}

}

}

**Note :**

We can compute only primes untill small range such as 10^6 or at max 10^7

It takes 0.06 sec to pre-compute all primes below 10^6 and 0.6 sec for 10^7

**References :**

<https://en.wikipedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif>

**2.Segmented Sieve**

**Concept:**

Segmented Sieve is the technique to mark the prime numbers in the range [ L , R ] Segment.

Suppose we precompute primes upto 10. i.e, [ 2,3,5,7 ]

Now we want to find primes in between 20 and 30

Now we have to mark all the multiples of [2,3,5,7] in the range [20,30]

Now selecting 2 , mark [ 20,22,24,26,28,30 ]  
Now selecting 3 , mark [ 21,27 ] (Since 24,30 are allready marked)

Now selecting 5 , mark [ 25 ] (Since 20,30 are allready marked)

Now selecting 7 (No number to mark)

By observation,

There are no numbers to mark selecting 7.

To mark primes below N we computed upto sqrt(N) in Sieve.

Similarly we should have a minimum prime in our hand Max = sqrt(R)

**Algorithm:**

1. Precompute untill a certain range using Sieve()
2. Check if ‘R’ falls under the pre-computed range. If YES, then no need of Segmented Sieve()
3. If No, Mark all the multiples of the primes that we computed in Sieve() in the range 2 to sqrt(Max)

**Function:**

void SegmentSieve(int L,int R)

{

int Max = sqrt(R);

int i,j;

for(i = 2; i <= Max ; i++)

{

if( Prime[i] )

for(j = L – L%i ; j<=R ;j++ )

{

if( j>=L && Prime[j] && j!=i )

Prime[j]=0;

}

}

}

**Conditions:**

1. j>=L because we are finding primes between [L,R]
2. i<=Sqrt(R) because it is enough to check with sqrt(R) for prime
3. j!=i to avoid marking a prime to 0
4. Prime[j] Checking if it is not allready marked as composite

if j! = i and Prime[j] is 1 then it follows that j is a multiple of i since j=j+i so we can mark Prime[j] as composite i.e 0

Segmented Sieve can be further improved by taking the primes in a vector and iterating it untill we reach sqrt(R) value.

**Important Points:**

1.Any prime number takes the form 6\*n-1 , 6\*n+1

2.The hypotenuse of right angled triangle is always a 4\*n+1 multiple of prime.

**3. Tailing Zeros of Factorial**

**Concept:**

let’s look at how trailing zeros are formed in the first place. A trailing zero is formed when a multiple of 5 is multiplied with a multiple of 2. Now all we have to do is count the number of 5’s and 2’s in the multiplication.

For Example consider 100! = 100 \* 99 \* 98 \* … \* 2 \* 1

counting No of 5’s = selecting the numbers 5,10,15,25,30,35,40,45…..100 = 20  
which can be counted by 100/5

Note that we counted only one 5 in 25 , but 25 = 5\*5 it should be counted twice, again divide by  
25, similarly with 125….etc   
  
Therefore No of tailing 0’s in N! = [N/5] + [N/25] + [N/125] …..

**4. Factorial Length**

**Concept:**

In order to find number of digits in a Number, it is done using the formula

Ndigits = log10(N) + 1. (Since we express any number in base 10 using 10 power)

Example : N=10001 , Ndigits = log10(N)+1 = 4.0003+1 = 5

In similar way in order to find the no.of digits in a factorial, it is simply Ndigits = log10(N!)+1

but it is difficult to compute N! for large N values.

using the property log(N!) = log(N) + log(N-1)+ ……log1 , we can compute log10(N) using a loop from 1 to N.

double Ans=0.00;

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

Ans = Ans + log10(i);

int Res = Ans+1;

This gives the perfect and accurate results, but for large N such as N<=10^6 it gives TLE.. with T<=1000  
  
There are two approximated formulas for caliculating the length of factorial

**Sterlings’s formula**

N! = sqrt(2\*PI\*N)\* pow(N/e , N)\*(1+1/12N)

This gave many WA’s it fails for large N Values

**Kamenetsky’s formula**

log(N!) = log(2\*PI\*N)/2 + N\*(log(N)-1)/log10

or log(N!) = ceil( log10(2\*PI\*N)/2 + N\*log10(N/e) )

**(Proofs Not Found…..Try Later)**

**5. Exponentation and Modular Exponentation**

**Concept:**

Basic approach for caliculating x^N is O(N) multiplying x\*x\*x….N times, and basic approach for finding this modulus is also same ((x\*x)%m)\*x%m)……N times.

By observation, to get 2^64 , it is enough to compute 2^32 multiplying it two times gives 2^64, and finding 2^32 it is enough to caliculate 2^16….. So when N is even it is enough to find N/2 …. following the relationship, we can get a relation in recursion to find x^n.

https://upload.wikimedia.org/math/a/9/9/a99eb157482b37137bceac54af762eb1.png

Which reduces the time complexity from to O(logN).

And Modulus can be caliculated at every step , inorder to get Modular Exponentation.

**Function:**

void ModExponent(int base,int Exp, int Mod)

{

int Ans=1;

while(Exp>0)

{

if(Exp%2==1) Ans = (Ans\*base)%Mod;

Exp = Exp/2;

base = (base\*base)%Mod;

}

}

Inorder to just find the Exponent , we can simply remove Modulus.

Inorder to find the last digit of x^n, we can simply use this with Mod = 10

**Properties of Modulus :**

(a\*b)%Mod = ( (a%Mod) \* (b%Mod) )%Mod

X % 2^n = X & (2^n-1)

(a/b)%Mod = a\*ModInverse(b,Mod) iff Mod is Prime

a%Mod , if you don’t know if a is negative or positive or less than or greater than Mod

Then Res = (a%Mod+Mod)%Mod

**6.The Next Palindrome Number**

**Concept:**

Inorder to find the next palindrome for a given number, the first approach is to loop from N+1 untill the next palindrome is found. O(N) , but to find the next palindrome for very very large numbers such for example

N = 10000000000000000000000000000000000009   
A = 10000000000000000011000000000000000001

For which it requires to loop over 10^18 which will definitely time out!

A Better thought is to break the given string into 2 parts Even/Odd should be checked timely. There are certain conditions that make this problem easy.

**Algorithm:**

1.Break the string into 2 parts.

Lets say S1 = [0,Mid] and S2 = [Len,Mid+1] (S2 is the string from right to left)

2.Check if S1==S2 then it is palindrome , so the next palindrome will be just adding one number to S1 and S2=Rev(S1) , return S1+S2

3.if(S1>S2) then S2 = Rev(S1) and return S1+S2

4. If(S2>S1) then S1=S1+1 , and S2 = Rev(S1) … the result will be S1+S2

**Example :**

1.456895 (Even)  
 Here …..S1=456 , S2 = 598   
 S2>S1 => S1=457 …… then S2=Rev(S1)=> 754 and the Answer will be 457754

2.45689 (Odd)

Here…..S1=45 , S2 = 98 since it is odd , add 1 to middle number and then add 1 to it.

457 now leave 7 and print it in reverse order 45754

3.999999

S1=999 S2=999 , since it is a palindrome add S1=S1+1 => S1 = 1000 and leave 0 and print it in reverse order Ans = 1000001

Note that there are many conditions.

1. We have to check if the given string is all 9’s
2. Check if the string is even or odd (slightly differs in implementation)
3. After adding 1, if the string length changes (slightly differs in implementation)
4. After adding 1 and getting the string , Even or Odd so printing again from mid to back changes in implementation
5. the S2 is only used to check S1<S2 case, other wise it is not at all needed , we only manipulate S1 string.

**7.Fenwick Tree or Binary Indexed Tree(BIT)**

**Concept:**

Consider a Array [ 0…..N ] and we have to process the query to get the sum in the given range.

Lets have a query to find the sum in range [0,K].

This problem can be solved in the following ways.  
**Brute Force** : Running from i=0 to K and finding the Sum+=Arr[i].

**Dp Array :** We can maintain a Dp Prefix sum and get the Sum[K].

**Segment Tree :** We can always process range queries using Segment Tree.

**Update and Query Times for above mentioned**:

**Brute Force** Update takes O(1) and Query takes O(N) .

**Dp Array** Update takes O(N) to update entire array, and Query takes O(1).

**Segment Tree** Query time O(logN) , Update is very complex with Lazy Propagation.

Hence in This Case, **Fenwick Tree** or **BIT** is efficient which takes

**O ( N\*Log(N) ) -** To Create the Tree.

**O ( Log(N) ) -** To Update the Tree.

**O ( Log(N) ) -** For A Query.

**NOTE :-** This is because the No. of set bits in a BIT is atmost Log(N).

**Conceptual Representation :**

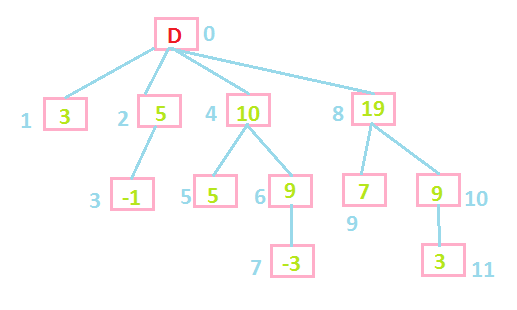
Consider an Array,

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | -1 | 6 | 5 | 4 | -3 | 3 | 7 | 2 | 3 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

**Elements** =

**Index**  =

**The Tree Will Look Like**



**NOTE :** This is just a theoritical representation, we will actually use Array to store the Tree.

Here D = Dummy, Which does not Contribute.   
The Parent-Child Relationship is as follows.

**To Get Parent of Node** :

To get the parent, we have to set the Right most set bit to 0.

For Example,

7 = 111 , setting Right most set bit to 0 becomes 110=> 6 (Parent).

5 = 101 , setting Right most set bit to 0 becomes 100=>4 (Parent).

8= 1000, setting Right most set bit to 0 becomes 0000=>0 (Parent).

4= 100, setting Right most set bit to 0 becomes 000=>0 (Parent).

We can efficiently set the right most bit to 0 by following operations.

1. Find 2’s Complement of that Number.
2. Make AND Operation with Original Number.
3. Subtract the result from the original Number.

**Filling Up The Tree : O( N\*Log(N) )**

Here we traverse the entire array, for each index i, we select Idx = i+1 first.  
Fill the Tree[Idx]=Arr[i].  
And also update the remaining tree which is effected by updating the current node by getting the next using the formula

**GetNext :**

1. 2’s Compliment of current Node.
2. Perform AND operation with original Number.
3. Add the Result to the Original Number.

There will be Log( Idx ) GetNexts for a selected Idx, which makes the algorithm to run in N\*Log(N) time for N elements in the worst case.

**Updating the Tree :**

Updating the tree is same as filling the tree, we use the same GetNext formula , we update the current Node and the GetNexts with the difference of Original Arr[i] and Updated Arr[i].

**Query :**

We can Ask Query to get the sum of the elements in the range [0,X] , how ever if we ever wanted to get a range query for example in the range [X1,X2] , then we can call the query two times namely [0,X2] and [0,X1-1] and subtracting those two will give the answer of the range [X1,X2].

To get the Sum in Range [0,X] , we choose X+1th Node  
Sum = Tree[X+1] + parent(X+1) + parent( parent(X+1) ) + …. Untill we reach parent 0.

**NOTE : GetNext Should be called Untill The max node is reached <= N+1.**