**Contents**

[Function: 2](#_Toc144948830)

[STL Function: 2](#_Toc144948831)

[Stack & Queue & Deque & Priority\_Queue 4](#_Toc144948832)

[Others 4](#_Toc144948833)

[Math: 5](#_Toc144948834)

[Bits: 6](#_Toc144948835)

[Bitset Function 7](#_Toc144948836)

[Combination And Permutation: 8](#_Toc144948837)

[Geometry: 10](#_Toc144948838)

[Algorithms: 13](#_Toc144948839)

[Binary Search 13](#_Toc144948840)

[Number Theory: 13](#_Toc144948841)

[Extended Euclid 13](#_Toc144948842)

[Sum and Count of Divisor 13](#_Toc144948843)

[Number of divisors 14](#_Toc144948844)

[Bitwise Sieve Algorithm (find prime number) 14](#_Toc144948845)

[Sieve Algorithm (find prime number) 15](#_Toc144948846)

[Prime Factorization (Integer factorization) 15](#_Toc144948847)

[Prime Factorization using Sieve algorithm 16](#_Toc144948848)

[Binary Exponentiation using Iterative method 16](#_Toc144948849)

[Binary Exponentiation for 17](#_Toc144948850)

[Graph: 17](#_Toc144948851)

[Depth First Search(DFS) 17](#_Toc144948852)

[Breadth-first search (BFS) And 0/1 BFS 18](#_Toc144948853)

[Dijkstra’s Shortest Path Algorithm(Single Source Shortest Path) 19](#_Toc144948854)

[Floyd-Warshall Algorithm(All Pair Shortest Path) 19](#_Toc144948855)

[Data Structures: 20](#_Toc144948856)

[Segment tree 20](#_Toc144948857)

[Segment tree Lazy 21](#_Toc144948858)

[Binary Indexed tree(BIT) 23](#_Toc144948859)

[Mo’s offline Query 24](#_Toc144948860)

[Disjoint Set Union(DSU) 25](#_Toc144948861)

[Lowest Common Ancistor(LCA) 26](#_Toc144948862)

[Trie 29](#_Toc144948863)

[String: 30](#_Toc144948864)

[Double Hashing 30](#_Toc144948865)

[Others: 32](#_Toc144948866)

[Sorting pair Using Compare Function 32](#_Toc144948867)

[Minimum fraction 32](#_Toc144948868)

[Find N’th Fibonacci number using Binet's Formula 32](#_Toc144948869)

[Count words in a string using stringstream 32](#_Toc144948870)

[Find SubString of a stirng 33](#_Toc144948871)

[Find SubSequences / SubSet using Iterative 33](#_Toc144948872)

[Find SubSequences / SubSet using Recursion 34](#_Toc144948873)

[\_\_int128 Data-type 34](#_Toc144948874)

# **Function:**

* char ch = **toupper**('z'); **or**, ch=**(‘**z’ **^ 32);** => ch=’**Z**’;
* char ch = **tolower**('B'); **or**, ch=**(‘**B’ **| 32);** => ch=’**b**’;
* **abs(x) = abs(-x)** = x;
* **sqrt(x)**= √x; [ **sqrtl**(x) return long double; ]
* **pow**(x, y)= xy**;** [ **powl**(x, y)return long double;] **=> round(**pow(x, y)**);**
* **log10(3)** = 0.4771212; **log2(3)** = 1.584962501;
* **round(x.y)** => if **y>=5** then **ans = x+1**;
* **ceil(x.y)** => if **y>=01** then **ans = x+1;** or**, ans=(a+b-1)/b;** [**ceil** symbol=> ⌈x.y⌉= x+1;]
* **floor(x.y)** => if **x.y positive** then **ans=x**; if **x.y negative** then **ans=x-1**;

ex: floor(-2.3) = -3, floor(3.8) = 3;

* **trunc(x.y)** => if **x.y positive / negative** then **ans = x**; if ex: floor(-2.3)= -2, floor(3.8)= 3;
* **stoi(s)** => convert string to integer; [**stoll(s)->** for long long int value.]
* **to\_string(num)** => convert integer to string;
* **getline(cin,** name**)** => input a line. [ignore buffer\_use => **cin >> ws;** or, **fflush(stdin);** or, **cout << flush;** or, **cin.ignore();**]
* **s1.substr(s1\_pos, s2\_len)** => This function generates a new string with its value initialized to a copy of a sub-string of this object. **[** if string s1=”abcdef”; then, string s2=s1.substr(1,3); -> s2 = ”bcd”**;** string s2 = s1.substr(1); -> s2 = ”bcdef”**; ]**
* **next\_permutation():** It is used to rearrange the elements in the range [first, last) into the next lexicographically greater permutation. {{1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2}, {3,2,1}};

int arr[] = {1, 2, 3}; => **O(n\*n!)**

do{

//Add any conditions;

cout << arr[0] << " " << arr[1] << " " << arr[2] << "\n";

} while (**next\_permutation**(arr, arr + 3));

## **STL Function:**

* **size()** – Returns the number of elements in the vector.[ **v.size();** **mp.size();** **st.size();** ]
* **empty()** – Returns whether the container is empty.If **empty** return **true**(1), if **not** **empty** return **false**(0). [ **v.empty();** **mp.empty ();** **st.empty ();** ]
* **front()** – Returns a reference to the first element in the vector. [v.front();]
* **back()** – Returns a reference to the last element in the vector. [v.back();]
* **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.

[ **v.insert(v.begin(),** value**);** **mp.insert({key, value}); st.insert(key);** ]

* **erase()** – It is used to remove elements from a container from the specified position or range. [ **v.erase(v.begin() + position); mp.erase(value); st.erase(value); str.erase(v.begin() + position);** ]
* **clear()** – It is used to remove all the elements. **[name.clear();]**
* **mp.count(**K**)** - The function returns the number of times the key K is present in the map/set container. [**st.count(K);**] => O(logn);
* **max\_size()**- Returns the maximum number of elements a set container can hold.

[ **v.max\_size(); mp.max\_size(); st.max\_size();** ]

* **find()** - An iterator to the first element in the range that compares equal to val. If no elements match, the function returns last(**v.end()**). [ **find (v.begin(), v.end(),** val**);]**

=>O(n); [**mp.find(**val**); st.find(**val**);** ] => O(log(n))

* **lower\_bound()** – Let v={1,5,7}; if we search **1** return **0’s** index address, if we search **2** return **1’s** index address, if we search **0** return **0’s** index address, if we search **7<value** then return **v.end()** address. **(must be sorted)** => O(logn)

[**auto x=lower\_bound(v.begin(), v.end(), val);] [auto x = mp.lower\_bound(val); auto x = st.lower\_bound(val);** ] →(x->first ; x->second;)

* **upper\_bound()** - Let v={1,5,7}; if we search **1** return **1’s** index address, if we search **2** return **1’s** index address, if we search **0** return **0’s** index address, if we search **7<value** then return **v.end()** address. **(must be sorted)** =>O(logn)
* **srand(time(NULL));** // for different values each time we run it using the rand function.
* **rand()**: The rand() function is used in C++ to generate random numbers in the range [0, RAND\_MAX). Ex: a=**rand()**; a=(**rand()%10)+1** [=> **a>=1 && a<=10**]; if lb=20 and ub=40 Then, a=**(rand() % (ub - lb + 1)) + lb;** [=> **a>=20 && a<=40**];
* **Better then rand()** function:

mt19937 rng(chrono::steady\_clock::now().time\_since\_epoch().count()); // mt19937\_64 (long long)

auto my\_rand(long long l, long long r) // random value generator [l, r]

{

return uniform\_int\_distribution<long long>(l,r)(rng);

}

* **Erase Duplicate value in sorted vector:** v.erase(unique(v.begin(), v.end()), v.end());
* **merge():** Merge two sorted arrays using merge present algorithm header file. **The** **Arrays must be sorted. => O(**vec1.size() + vec2.size() **)**

**merge**(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), **back\_inserter**(finalVec));

**merge**(st1.begin(), st1.end(), st2.begin(), st2.end(), **inserter**(st[node], st.begin()))**;**

* **Policy based DS**: The complexity of the **insert** and **erase** functions is **O(log n).**

#include <ext/pb\_ds/assoc\_container.hpp>

#include <ext/pb\_ds/tree\_policy.hpp>

using namespace \_\_gnu\_pbds;

template <typename T> using **ordered\_set** = tree<T, null\_type, **less<T>**, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

template <typename T, typename R> using **ordered\_map** = tree<T, R, **less<T>**, rb\_tree\_tag, tree\_order\_statistics\_node\_update>;

// **\*s.find\_by\_order(k)**: K-th element in a set (counting from **zero**).

// **s.order\_of\_key(k)**: Number of items strictly smaller than k. (same as, lower\_bound of k)

// **less\_equal<T>** => for **ordered\_multiset** or, **ordered\_multimap.**

**ordered\_set<int> s; ordered\_map<int, ll>mp;** // we can change the data type.

* **gp\_hash\_table<int, int>:** Same as unordered\_map, but **faster** than unordered\_map.
* **Compare function** for **gp\_hash\_table** and **unordered\_map**.

struct pair\_hash {

template<typename T, typename U>

size\_t operator()(const pair<T, U>& p) const {

auto hash1 = hash<T>{}(p.first);

auto hash2 = hash<U>{}(p.second);

return hash1 ^ hash2;

}

};

gp\_hash\_table<pair<ll, ll>, int, **pair\_hash**> xyz;

## **Stack & Queue & Deque & Priority\_Queue :**

* **push():** Adds the element ‘**x**’. [ **s.push(**x**);** **q.push(**x**); pd.push(x);** ]
* In Deque, you can push both side [ **dq.push\_back(**x**); dq.push\_front(**x**);** ]
* **pop():** Deletes the element. [ **name.pop();** ]
* In Deque, you can pop both side [ **dq.pop\_back(**x**); dq.pop\_front(**x**);** ]
* **top():** Returns a reference to the top most element of the stack. [ **s.top(); pq.top();** ]
* **front():** Returns a reference to the first element of the queue. [ **q.front(); dq.front();** ]
* **back():** Returns a reference to the last element of the queue. [ **q.back();** **dq.back();**]
* **empty():** Returns whether the stack/queue is empty. It return **true** if the stack/queue is empty otherwise returns false. [**name.empty();**]
* **dq.at(**x**):** Returns a reference to the element at position **x** in the deque container object. Ex: dq = {10,3,15,20} ; ans= dq.at(2); =>ans=15
* **priority\_queue<**int**>max\_heapPQ;** =>In this queue elements are in non-increasing. [ same as **multiset <int, greater<int> > s;**]
* **priority\_queue<**int**,** vector<int>, **greater<int>>min\_heapPQ;** =>In this queue elements are in non-decreasing order. [similar to **multiset**]

## **Others:**

* **binary\_search();** Return true(1) or false(0). If found return **1**. Else return **0**;

**[ binary\_search(**start\_address**,** end\_address**, value\_to\_find);]**

* **max\_element():** Returnthe maximum value’s address of the vector.

[ **int max= \*max\_element(v.begin(),v.end());** ] => O(n)

* **min\_element():** Returnminimum value’s address of the vector.

[ **int min= \*min\_element(v.begin(),v.end());** ] => O(n)

* **accumulate():** Returnsummation of the vector.

[ **int sum= accumulate(v.begin(),v.end(), 0);** ] => O(n)

* **count():** Return count of the ‘val’ element.

[ **int c = count(v.begin(), v.end(),** val**)**; ] => O(n)

* **all of()**/**any\_of()**/**none of():** Are **all of**/**at least one**/**none of** the elements **greater than 0?** (you can change this condition) => **return true(1) or, false(0).**
* **all\_of(v.begin(), v.end(), [ ](int x){return x>0;} );**
* **any\_of(v.begin(), v.end(), [ ](int x){return x>0;} );**
* **none\_of(v.begin(), v.end(), [ ](int x){return x>0;} );**
* **iota()**: The algorithm iota() creates a range of sequentially increasing values.

**[ iota(a, a+5, 10);** => a[5]={10,11,12,13,14};]

* **is\_sorted():** Return bool value. If, vector are sorted return 1. Else return 0.

[ **bool x= is\_sorted(v.begin(), v.end());** ] => O(n)

* **gcd():** Return a and b gcd(**Greatest Common Divisor**) value. => O(logn)

[ int gcd**= \_\_gcd(a,b);** gcd(m\*a, m\*b) = m\*gcd(a, b)**;**  gcd(a/d, b/d) = gcd(a, b)/d**;** ]

* **lcm()**: Return a and b lcm(**Least Common Multiple**) value. => O(logn)

[ int lcm**= (a\*b)/\_\_gcd(a,b);** lcm(m\*a, m\*b) = m\*lcm(a, b)**;** ]

* lcm(a, b, c) =.
* **memset():** Initialize a **1D** vector with **-1**(or, **0**):=>O(n)

**memset(vec\_name, -1, sizeof(vec\_name)); ->** use any time.

* Initialize a **1D** vector with 1(or, any number):=>O(n)

**vector<int>** vec**(n , 1);** [ n=row ] **->** use only initialize time.

* Initialize a **2D** vector with 1(or, any number):=>O(n)

**vector<vector<int>>** vec**(n , vector<int> (m, 1));** [n=row, m=col]

# **Math:**

* p+(p+1)+…+(q - 1)+q = **(q + p)(q – p + 1) / 2;** [Ex: **7**+8+9+10+**11**=(11+7)(11-7+1) / 2 =45]
* 1+2+3+…+(n - 1)+n = **(n \* (n + 1)) / 2**; [Ex: 1+2+3+4+5=(5\*(5+1))/2 = 15]
* 1+3+5+…+ (2n - 3) + (2n - 1) = **N2**; [N-> number of size] [Ex: 1+3+5 = 32 = 9]
* 2+4+6+…+ (2n - 2) + 2n **= N \* (N + 1)**; [N-> number of size] [Ex: 2+4+6 = 3\*(3+1) =12]
* 12+22+32+ . . .+ (n - 1)2 + n2 = **n (n + 1) (2n + 1) / 6;** [Ex: 1+4+9= 3(3+1)(2\*3 +1)/6 =14]
* 13 + 23 + 33 + . . . + (n - 1)3 + n3 = **{n (n + 1) / 2} 2 ;** [Ex: 1+8+27= {3(3+1) / 2}2 =36]
* 12 + 32 + 52 + . . . + (2n - 3)2 +(2n - 1)2 = **N\*(4N2 - 1) / 3;** [Ex: 1+9+25 = 3\*(4\*32 - 1)/3 = 35]
* 13 + 33 + 53 + . . . + (2n - 3)3 +(2n - 1)3 = **N2 (2N2 - 1);** [Ex: 1+27+125 = 32 (2\*32 - 1) = 153]
* 14 + 24 + 34 + . . . + (n – 1)4 + n4 = **n(n+1) (2n+1) (3n2 + 3n – 1) / 30;**

[Ex: 1+16+81+256 = 4(4+1) (2\*4+1) (3\*42 + 3\*4 – 1) / 30 = 354]

* c a + c a+1 + · · · + c b = **(c b+1 − c a) / (c – 1);** [c != 1]
* 20 + 21 + 22 + 23 + . . . + 2(k - 1) = **2k – 1;** [Ex: 1+2+4+8+16+32 = 26 - 1 = 63]
* If F(n) = -1 + 2 - 3 + . . .+ (-1) n \* n
* If N **even** number, **ans = N/2;**
* If N **odd** number, **ans = ((N + 1) / 2) \* (-1));**
* N-th **Odd** number = **(2 \* N)-1**;
* N-th **Even** number = **2\*N**;
* a + a\*k + a\*k2 + … + b = **((b \* k) - a) / (k - 1).** [ex: 3 + 6 + 12 + 24 = ((24 \* 2) -3) / (2-1) = 45]
* a + (a+4) + (a+2\*4) + … + b= **(n \* (a + b)) / 2**. [n-> number of size]

[ex: 3 + 7 + 11 + 15 = (4 \* (3 + 15)) / 2 = 36.]

* even **±** even = **even;** even **±** odd = **odd;** odd **±** odd = **even**;
* even **×** even = **even;** even **×** odd = **even;** odd **×** odd = **odd**;
* Number of digits in N= **floor(log10(**N**)) + 1;**
* Number of trailing **zeros** in **N!** => **while(N) sum+=N/5, N/=5; [**Ex: 10! = 36288**00;]**
* For a grid of size **(N x N)** the total number of **squares** formed: **((n\*(n+1)) \* (2n+1)) / 6;**
* **5 minutes** Clock Angular Value is **30°**. **[ 1 min = 6°]**
* Angle between clock minute and hour, **ans = abs ((0.5 \* 11 \* m) – (30 \* h));**
* For smaller angle, **if (ans >180) ans = 360 – ans;**
* The number of ways of selecting one or more things from N different things is given by

2N -1. (combination)

* Number of possible of N bits = 2N. [4bits, 24 =16 => 0 to 15 number possible with using 4 bits] (2n -1) → highest value.
* N = **2x =>** x = **log2(N)**. Ex: 64 = 2**6** [ log2(64) = **6**].
* **logu(x)=**  [k-> any base (2,10)]; **loga(k)=** ; **ax = b ;=> x = loga b;**
* **(A \* B) = ((A % Mod) \* (B % Mod)) % Mod;** **<=** [Same As **+,-** Operator]
* **(A / B) = ((A % Mod) \* ( BinExp(B , Mod-2) % Mod)) % Mod;**

# **Bits:**

* **Bitwise AND( & ):** **(1 & 1)= 1;**
* **Bitwise OR( | ):** **(0 | 1)= 1;** **(1 | 0)= 1;** **(1 | 1)= 1;**
* **Bitwise ExOR( ^ ):** **(0 ^ 1)= 1; (1 ^ 0)=1;**
* **Bitwise NOT( ~ ):** inverts all bits of it. [ a = 10012 **->** **(~a) = 0110**]
* **Right Shift( >> ):** right shifting an integer “x” with an integer “y” denoted as ‘(x>>y)‘ is equivalent to **dividing** x with **2^y**. Ex: let’s take N=32; which is 100000 in Binary Form. Now, if N=(N>>2) then N will become N=N / (2^2). Thus, N=32 / (2^2) = 8 which can be written as 1000. [ 18 = (10010)2 → **(18>>1) = 01001; (18>>2) = 00100;**]
* **Left Shift( << ):** left shifting an integer “x” with an integer “y” denoted as ‘(x<<y)’ is equivalent to **multiplying** x with **2^y**. Ex: let’s take N=22; which is 00010110 in Binary Form. Now, if N=(N<<2) then N will become N=N \* (2^2). Thus, N=22 \* (2^2) = 88 which can be written as 01011000. [ 3 = (11)2; => **(3<<1) = 110; (3<<2) = 1100**;]
* **(N&1) == 1** **->** N **odd** number; **(N&1) == 0** **->** N **even** number;
* (**N / 2**) **== (N >> 1);** (**N \* 2**) == **(N << 1);**
* (**2^N**) **== (1**LL **<< N); => N = (1LL <<** (long long)**log2**(N) **);**
* A quick way to **swap** **a** and **b** => [ **a ^= b, b ^= a, a ^= b**;]
* **is\_power\_of\_two(val)** => **(val & (val - 1)) == 0;**
* **CheckBit(val, pos) => (val & (1LL << pos));**
* **SetBit(val, pos) => (val |= (1LL << pos));**
* **ClearBit(val, pos) => (val &= ~(1LL << pos));**
* **FlipBit(val, pos) => (val ^= ~(1 << pos));**
* **MSB(mask) => 63 - \_\_builtin\_clzll(mask);** [Most Significant Bit position]
* **LSB(mask) => \_\_builtin\_ctzll(mask);** [Least Significant Bit position]
* **\_\_builtin\_popcount(**x**):** This function is used to count the number of one’s(set bits) in an integer(32 bits).Similarly you can use **\_\_builtin\_popcountll(**x**)** for **long long** data types (64 bits). Ex: x = **5** (101) => ans=2 ;
* **\_\_builtin\_clz(**x**):** It counts number of zeros before the first occurrence of one(set bit) of the integer(**32** bit).( **clz = count leading zero’s.**)

Ex: x= **16** (00000000 00000000 00000000 00010000) => ans = 27

* **\_\_builtin\_parity(**x**):** This function returns true(1) if the number has **odd** parity else it returns false(0) for **even** parity.( **parity= count the number of one’s**)

Ex: x = **7** (111) => ans = 1; x = **6** (110) => ans = 0;

* **\_\_builtin\_ctz(**x**):** Count number of zeros from **last to first** occurrence of one(set bit) of the given integer.( **ctz = count trailing zeros**;) Ex: x = **16** (00010000) => ans = 4 ;

## **Bitset Function:**

**bitset<** highest\_Bit\_number **> name(data);**

* **bitset<**64**>** b1(val); or**, bitset<**4**>**b2("1011"); => auto-convert to binary;
* **to\_ulong():** Converts the contents of the **bitset** to an **unsigned long integer**; [ Ex: b1 = 1001, int val = b1.**to\_ulong**(); => val = 9;]
* **to\_string():** Converts the contents of the **bitset** to a **string**;

[ Ex: b1 = 1001, s1 = b1.to\_string(); => s1= ”1001”; ]

* **flip(**position**)**: flip function flips all bits (**1 to 0** and **0 to 1**).

[ Ex: b1 = 1001; b1.**flip**(1); =>b1 = 1011**;** b1.flip(1); =>b1=1001;]

* **count():** returns the total number of **set bits**(1); [Ex: b1=1001; bit= b1.count(); => bit =2;]
* **any():** function to check if **any of its bits are set or not**; [Ex: b1=1001, any\_set = b1.any(); => any\_set = 1; b2=0000; any\_set = b1.any(); => any\_set = 0;]
* **set():** b1.set(**pos**) makes bset[pos] = 1;(i.e. default is 1). b1.set(**pos**, **value**) makes bset[pos] = 0 or, 1; [Ex: b1=1001; b1.set**( )** =>b1 = 1111; b1.set(1)=> b1= 1011; b1.set(0)=> b1=1001; b1.set(3, 0);=> b1=0001; b1.set(2,1)=> b1= 1101;]
* **reset():** reset function makes all bits 0;

[Ex: b1 = 1001; b1.reset()=> b1 = 0000; b1.reset(3)=> b1 = 0001;]

# **Combination And Permutation:**

* nCr = Or, nPr = nCr \* r!

**Combination(C):**

* If, **Order** **Doesn’t** **Matter** and **Repetition** **Allowed** then,

Possibilities, nCr =

* If, **Order** **Doesn’t** **Matter** and **Repetition Not** **Allowed** then,

Possibilities, nCr =

* **Properties**: nCr = nC(n-r), nC0 = nCn = 1, nC1= nC(n-1) = n, nCr + nC(r-1) = (n+1)Cr,

nCx = nCy => x = y or, x + y = n.

* **nCr** has **maximum** **value** if:
* **r = n/2 ;** when n is **Even.**
* **r = (n+1)/2;** when n is **Odd.**
* **Short Technique:**  **nCr =**

Ex: 20C3 = = 1140 , 10C8 = 10C**10-8** = 10C**2** = = 45, 20C**15** = 20C**20-15** = 20C**5**= = 15504.

**Permutation(P):**

* If, **Order** **Matter** and **Repetition** **Allowed** then, Possibilities = nr
* If, **Order** **Matter** and **Repetition Not** **Allowed** then, Possibilities =
* **Properties**: nP0 = 1, nP1 = n, nP(n-1) = n!, nPr**/**nP(r-1) = n – r + 1.
* **Short Technique: nPr = n\*(n-1)\*(n-2)\*… r’th times.**

Ex: 10P3 = 10 \* (10-1) \* (10-2) = 720, 15P4 = 15 \* 14 \* 13 \* 12 = 32760.

**Find Combination(nCr):**  **=> O(r\*log(n))**

Ex: 5C2 = 10, 13C5= 1287;

void nCr( ll n, ll r)

{

ll p= 1, k=1, m;

if (n – r < r) r = n - r;

if (r != 0)

{

while(r)

{

p\*=n, k\*=r;

m=\_\_gcd(p, k);

p/=m, k/=m;

n--, r--;

}

}

else p=1;

cout<<p<<endl;

}

**Find Permutation (nPr):**  **=> O( n)**

Ex: 5P2= 20, 6P3= 120;

ll fact(ll n)

{

if(n <= 1) return 1;

return n \* fact(n - 1);

}

ll nPr(ll n, ll r)

{

return fact(n) / fact(n - r);

}

int main()

{

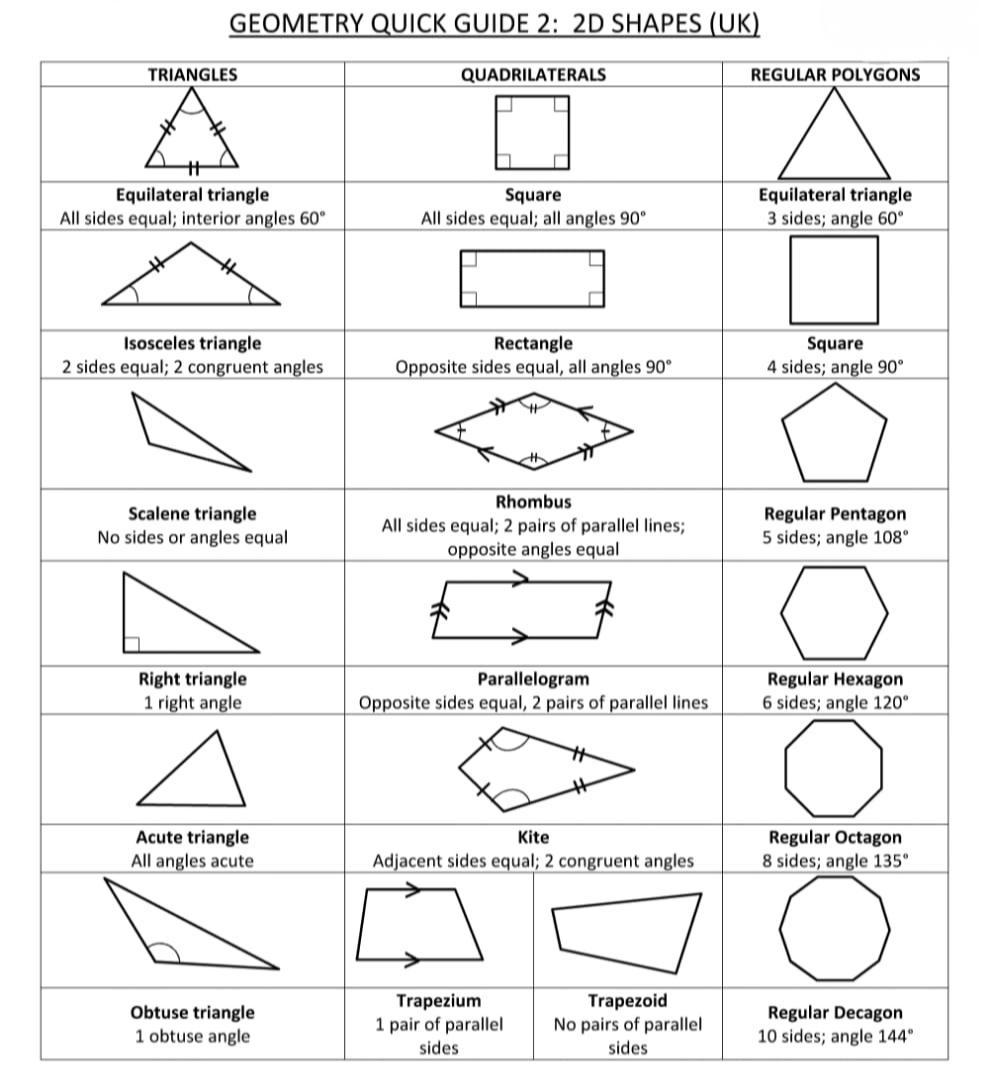
ll n, r;

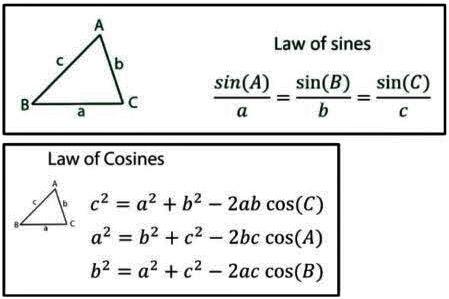
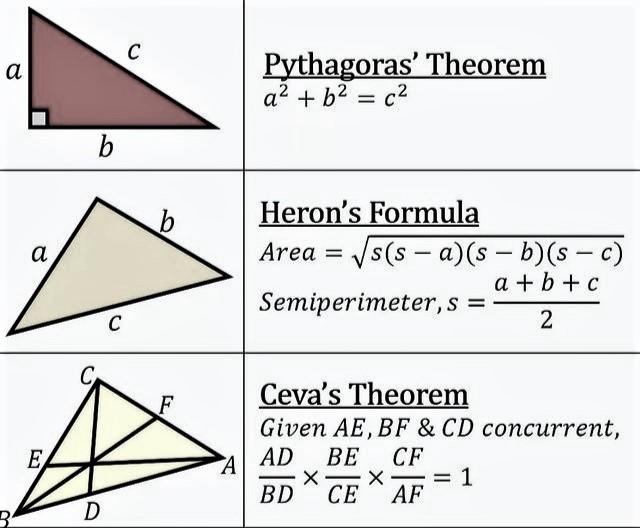
cin>>n>>r;

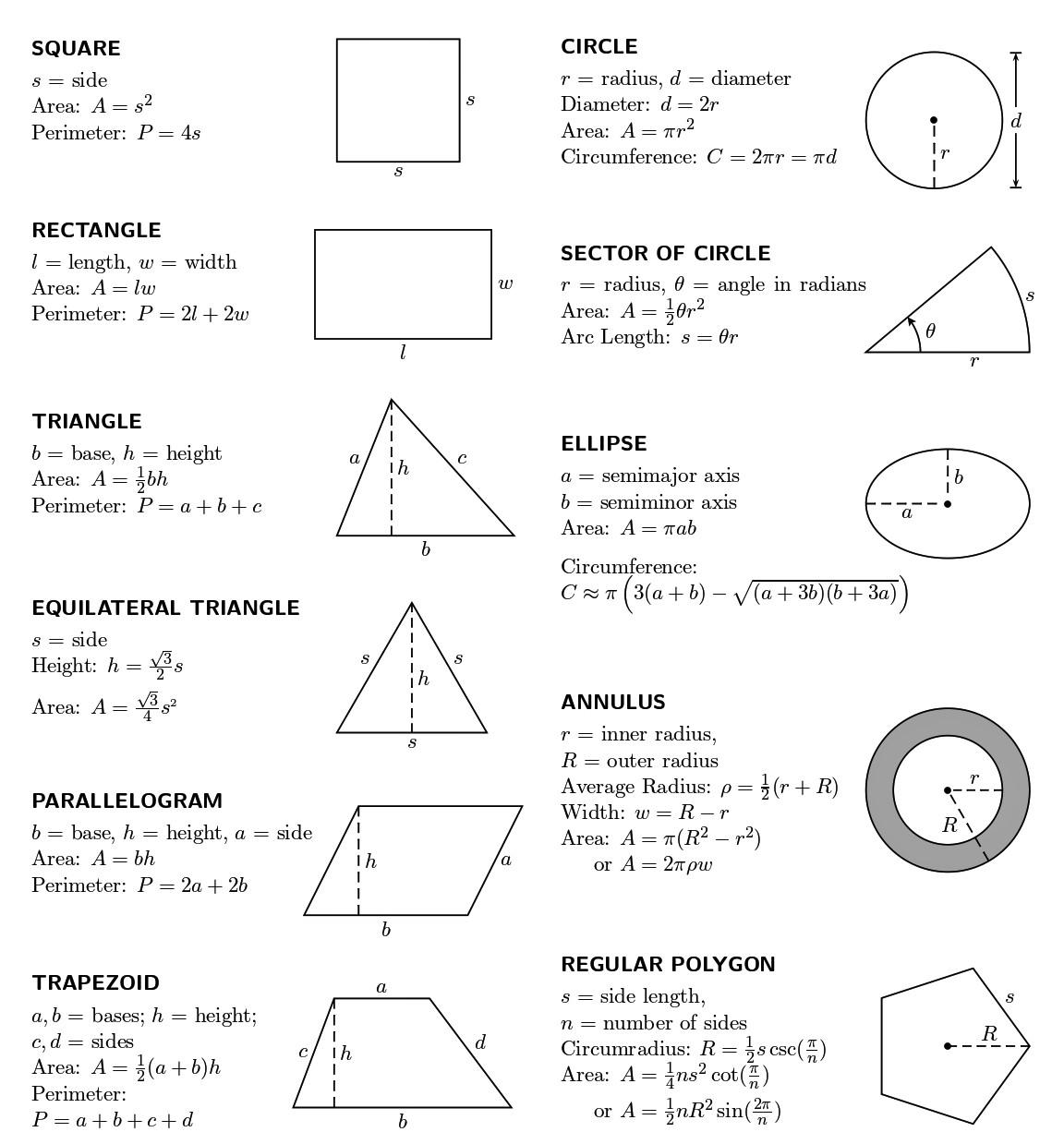
cout<<nPr(n, r);

}

# **Geometry:**



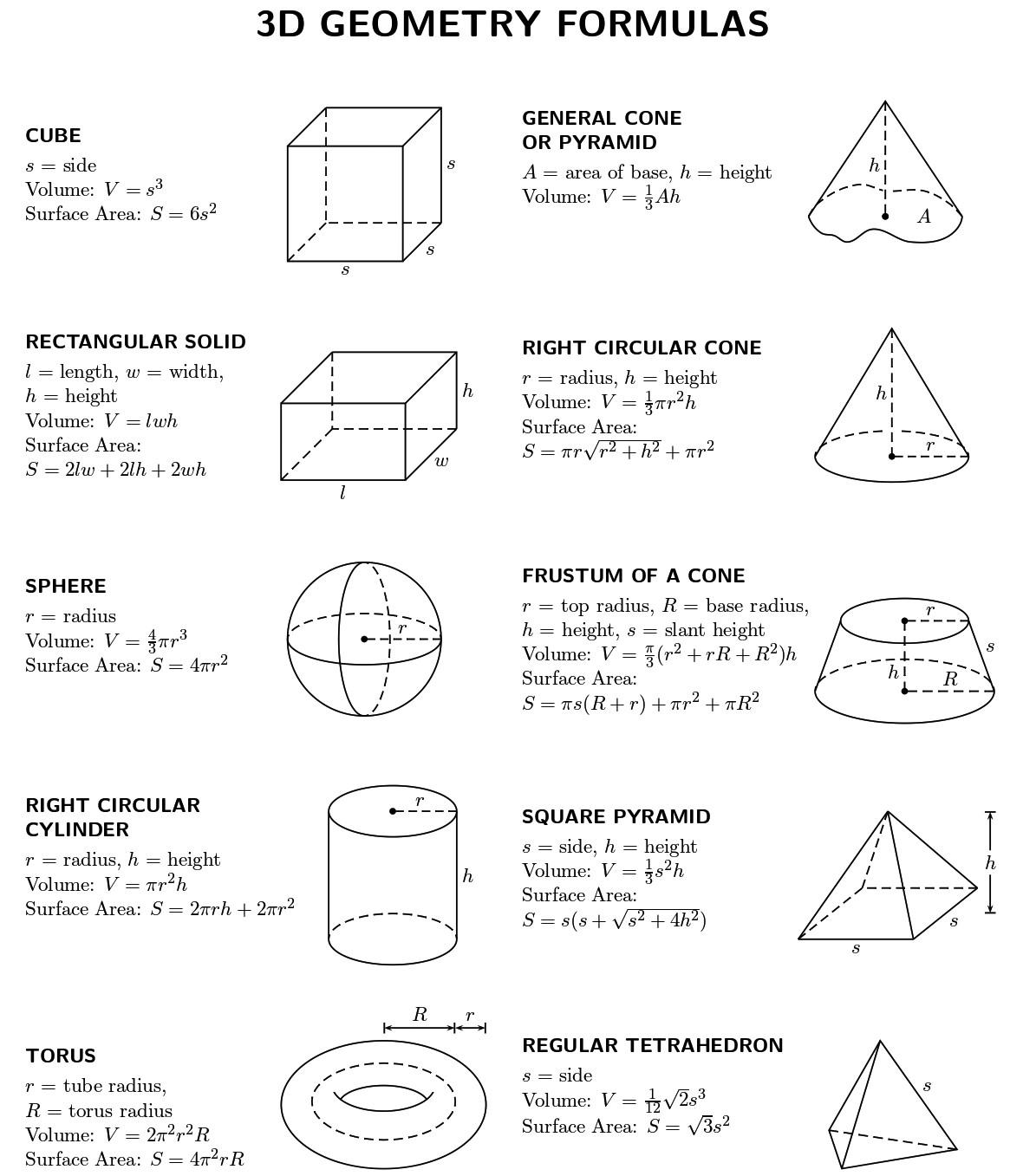
****



**Rhombus: Area =** (d1 \* d2) / 2 **=** s2 \* sin(C)**; Perimeter =** 4\*s ;

**Kite**: **Area** = (d1 \* d2) / 2; **Perimeter =** 2(s1 + s2);

**[** d1 and d2= lengths of the diagonals,s = s1= s2 = length of side, C = interior angle;**]**



* **pi** = 2 \* acos(0.0);
* Convert **Radian** to **Degree**: **sin(**val **\* (pi / 180.0)); asin(**val**) \* (180.0 / pi);**

# **Algorithms**

## **Binary Search:** **=> O(log(n))**

void BinarySearch(vector<ll> &v, int n, int target)

{

int low = 0, high = v.size() - 1, c = 0, mid = 0;

**while (low** **<= high)** // or, high - low > 1

**{**

c++;

**mid = (low + high) >> 1;** // or, mid= low+(high - low)/2; **or**, (low + high)/2;

**if (v[mid] < target) low = mid + 1;**

**else high = mid;**

**}**

if (v[low] == target) cout << low << "Found\n";

else if (v[high] == target) cout << high << "Found\n";

else cout << "Not Found\n";

}

# **Number Theory**

## **Extended Euclid:** //=> **O(log(min(a, b)))**

For this Eq. **(a\*x) + (b\*y) = gcd(a, b);**

ll extended\_euclid(ll a, ll b, ll &x, ll &y)

{

if (b == 0)

{

x = 1, y = 0;

return a;

}

ll x1, y1;

ll gcd = extended\_euclid(b, a % b, x1, y1);

x = y1;

y = x1 - y1 \* (a / b);

return gcd;

}

## **Sum and Count of Divisor: =>O(sqrt(n))**

Ex(sum): 20 => 22 (1+2+4+5+10+20). Ex(count): 20 => 6 (1,2,4,5,10,20).

void divisor()

{

ll n, sum = 0, i, c = 0;

cin >> n;

vector<ll> divisors;

for (i = 1; i \* i <= n; i++)

{

if (n % i == 0)

{

sum += i, ++c;

divisors.push\_back(i);

if (i != n / i)

sum += n / i, ++c, divisors.push\_back(n / i);

}

}

cout<<"Count = "<< c <<", Sum = "<< sum <<endl;

**sort**(divisors.begin(), divisors.end());

for(auto &i: divisors) cout << i << " ";

}

**Number of divisors: => O( nlog(n))**

Ex: 32 => 2 4 8 16 32

const int N = 1e5 + 10;

vector<int> divisor[N];

int main()

{

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

{

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

divisor[j].push\_back(i);

}

int n; cin >> n;

for (auto &it : divisor[n]) cout << it << " ";

cout << endl;

}

## **Bitwise Sieve Algorithm (find prime number):** **=>O(nloglogn)**

bool **CheckBit**(int N, int pos) { return (bool)(N & (1 << pos)); }

void **SetBit**(int &N, int pos) { N = N | (1 << pos); }

const int MAX = **1e8** + 5;

int **isPrime**[(MAX >> 5) + 2];

vector<int> **primes**;

void **sieve**()

{

int lim = sqrtl(MAX);

for (int i = 3; i <= lim; i += 2)

{

if (!CheckBit(isPrime[i >> 5], i & 31)) // **i>>5 = i/32**; **i&31 = i%32;**

{

for (int j = i \* i; j <= MAX; j += (i << 1))

{

SetBit(isPrime[j >> 5], j & 31);

}

}

}

primes.push\_back(2);

for (int i = 3; i <= MAX; i += 2)

{

if (!CheckBit(isPrime[i >> 5], i & 31))

primes.push\_back(i);

}

}

## **Sieve Algorithm (find prime number):** **=>O(nloglogn)**

const int N = 1e7 + 10;

vector<bool> isPrime(N, 1);

void sieve()

{

isPrime[0] = isPrime[1] = false;

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

{

if (isPrime[i] == true)

{

for (int j = 2 \* i; j < N; j += i)

{

isPrime[j] = false;

}

}

}

}

## **Prime Factorization (Integer factorization):** **=> O(sqrt(n))**

Ex: 36 => 2 2 3 3

int main()

{

int n;

cin >> n;

vector<int> prime\_factors;

for (int i = 2; i \* i <= n; i++)

{

while (n % i == 0)

{

prime\_factors.push\_back(i);

n /= i;

}

}

if (n > 1) prime\_factors.push\_back(n);

for (auto &prime : prime\_factors)

cout << prime << " ";

}

## **Prime Factorization using Sieve algorithm: => O(log(n))**

Ex: 50 => 2 5 5

vector<int> spf(N); // SPF : smallest prime factor

void sieve() // => O( nloglogn)

{

for (int i = 1; i < N; i++) spf[i] = i;

for (int i = 2; i \* i < N; i++)

{

if (spf[i] == i)

{

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

if (spf[j] == j) spf[j] = i;

}

}

}

int main()

{

sieve();

int n;

cin >> n;

while (n != 1)

{

cout << spf[n] << " ";

n /= spf[n];

}

}

## **Binary Exponentiation using Iterative method:** **=> O( log(b)).**

Ex: 313 => 3(8+4+0+1) => 38 \* 34 \* 30 \* 31=> 1594323; →**(ab)**

const int Mod = 1e9 + 7;

long long BinExpIter(long long a, long long b)

{

long long ans = 1;

while (b)

{

if (b & 1)

{

ans = ans \* a;

// ans=(ans\*a) % Mod;

}

a = a \* a;

// a=(a\*a) % Mod;

b >>= 1;

}

return ans;

}

**Binary Exponentiation for :** **=> O(x\*log(N\*10d))**

**=** 1.2457312346;

double **eps** = 1e-6; // eps=1e-**d**; =>with **d** **decimal accuracy**

double **BinExpPow** (double n, int x)

{

double l = 0, r = n, m = (l + r) / 2;

while (r - l > eps)

{

if (**pow**(m, x) > n) r = m;

else l = m;

m = (l + r) / 2;

}

return m;

}

# **Graph:**

const int fx[] = {+0,+0,+1,-1,-1,+1,-1,+1}; // **king's** move (0 to 3 index => Side Moves)

const int fy[] = {-1,+1,+0,+0,+1,+1,-1,-1}; // **king's** move (4 to 7 index => Diagonal Moves)

const int kx[] = {-2,-2,-1,-1,+1,+1,+2,+2}; // **knight's** move

const int ky[] = {-1,+1,-2,+2,-2,+2,-1,+1}; // **knight's** move

## **Depth First Search(DFS):** => **O(V+E)**

const ll N = 1e5 + 10;

vector<ll> g[N], height(N), depth(N);

bool vis[N];

int Par[N];

void dfs(ll vertex, ll par = -1)

{

**/\*\*\*** Take action on vertex after entering the vertex. **\*\*\*/**

**vis**[vertex] = true**;**

// bool isLoopExists = false; //<= Use For Finding Cycle

**Par**[vertex] = par;

**for (auto &child : g[vertex])**

**{**

**/\*\*** Take action on child before entering the child node. **\*\*/**

**if (vis[child]) continue;**

**dfs(child, vertex);**

**/\*** => **Use for Finding Cycle:**

if(vis[child] && child == par) continue;

if(vis[child]) return true;

isLoopExists |= dfs(child, vertex);

**\*/**

**/\*** => **Use for Tree (No need Visited array):**

if(child == par) continue;

depth[child] = depth[vertex]+1;

dfs(child, vertex);

height[vertex] = max(height[vertex], height[child]+1);

**\*/**

**/\*\*\*** Take action on child after exiting child node.  **\*\*\*/**

**}**

**/\*\*\*** Take action on vertex before exiting the vertex. **\*\*\*/**

**}**

## **Breadth-first search (BFS) And 0/1 BFS:** => **O(V+E)**

const int N=1e5+10;

vector<int>g[N]; //vector<**pair<int, int>**> g[N]; =>For 0/1 BFS

bool vis[N]; // **No need** vis array for 0/1 BFS.

vector<int> level(N); //vector<int> level(N, INT\_MAX); =>For 0/1 BFS

void **bfs**(int **source**)

{

**queue<int>q;** //deque<int>q; =>For 0/1 BFS

**q.push(source);** //q.push\_fornt(source); =>For 0/1 BFS

**vis[source]=1;** //level[source] = 0; =>For 0/1 BFS

**while(!q.empty())**

**{**

**int par=q.front();**

**q.pop();** //q.pop\_front(); =>For 0/1 BFS

**for(auto &child: g[par])**

**{**

**if(vis[child]) continue;**

**q.push(child);**

**vis[child]=1;**

**level[child]=level[par]+1;**

**/\***=> **For 0/1 BFS:**

int u = child.first, w = child.second ;

if(level[par] + w **<** level[u])

{

level[u] = level[par] + w;

if(w==0) q.push\_front(u);

else q.push\_back(u);

}

**/\***

}

}

}

## **Dijkstra’s Shortest Path Algorithm(Single Source Shortest Path):**

const int N = 1e5 + 10, INF = 1e9 + 7; => O((V+ E)\*log(V))

vector<pair<int, int>> g[N]; //g[u].pb({v,w});

vector<int> dist(N, INF); //store minimum distance;

vector<bool> vis(N);

void dijkstra(int s)

{

multiset<pair<int, int>> st;

st.insert({0, s});

dist[s] = 0;

while (st.size())

{

int u = (st.begin())->second;

// int u\_w=(st.begin())->first;

st.erase(st.begin());

if (vis[u]) continue;

vis[u] = 1;

for (auto &child : g[u])

{

int v = child.first;

int v\_w = child.second;

if (dist[u] + v\_w < dist[v])

{

dist[v] = dist[u] + v\_w;

st.insert({dist[v], v});

}

}

}

}

## **Floyd-Warshall Algorithm(All Pair Shortest Path):** =>O(n^3)

=> finding the shortest paths in a weighted graph with positive or negative edge weights (but with **no negative cycles**);

const int N=510, INF=1e9+10;

int dp[N][N];

int n, m;

void floyd\_warshall()

{

for (int k = 1; k <= n; ++k)

{

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

{

for (int j = 1; j <= n; ++j)

{

if (dp[i][k] < INF && dp[k][j] < INF)

dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);

}

}

}

}

int main()

{

cin>>n>>m;

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

{

for(int j=1;j<=n; ++j)

{

if(i==j) dp[i][j]=0;

else dp[i][j]=INF;

}

}

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

{

int x, y, wt; cin>>x>>y>>wt;

dp[x][y]=wt;

}

**floyd\_warshall();**

return 0;

}

# **Data Structures:**

## **Segment tree:**

const int N = 3e5 + 9;

ll a[N];

ll t[4 \* N];

void **build**(int node, int st, int en) //=> O(N)

{

if (st == en)

{

t[node] = a[st];

return;

}

int mid = (st + en) / 2;

build(2 \* node, st, mid); // divide left side

build(2 \* node + 1, mid + 1, en); // divide right side

t[node] = t[2 \* node] + t[2 \* node + 1]; // merging left and right portion

}

ll **query**(int node, int st, int en, int l, int r) //=> O(log n)

{

if (st > r || en < l) // No overlapping and out of range

{

return 0;

}

if (l <= st && en <= r) // Complete overlapped (l-r in range)

{

return t[node];

}

// Partial overlapping

int mid = (st + en) / 2;

ll q1 = query(2 \* node, st, mid, l, r);

ll q2 = query(2 \* node + 1, mid + 1, en, l, r);

return q1 + q2;

}

void **update**(int node, int st, int en, int idx, ll val) //=> O(log n)

{

if (st == en)

{

a[st] = val;

t[node] = val;

return;

}

int mid = (st + en) / 2;

int left = 2 \* node, right = 2 \* node + 1;

if (idx <= mid) update(left, st, mid, idx, val);

else update(right, mid + 1, en, idx, val);

t[node] = t[left] + t[right];

}

## **Segment tree Lazy:**

const int N = 5e5 + 9;

int a[N];

struct ST

{

#define lc (n << 1)

#define rc ((n << 1) | 1)

ll t[4 \* N], lazy[4 \* N];

ST()

{

for (int i = 0; i < 4 \* N; i++)

t[i] = lazy[i] = 0;

}

inline void **push**(int n, int st, int en)

{

if (lazy[n] == 0) return;

t[n] = t[n] + lazy[n] \* (en - st + 1);

if (st != en)

{

lazy[lc] = lazy[lc] + lazy[n];

lazy[rc] = lazy[rc] + lazy[n];

}

lazy[n] = 0;

}

inline void **pull**(int n)

{

t[n] = t[lc] + t[rc];

}

void **build**(int n, int st, int en)

{

lazy[n] = 0;

if (st == en)

{

t[n] = a[st];

return;

}

int mid = (st + en) >> 1;

build(lc, st, mid);

build(rc, mid + 1, en);

pull(n);

}

void **update**(int n, int st, int en, int l, int r, ll v)

{

push(n, st, en); // push the value left and right child

if (r < st || en < l) return;

if (l <= st && en <= r)

{

lazy[n] = v; // set lazy

push(n, st, en);

return;

}

int mid = (st + en) >> 1;

update(lc, st, mid, l, r, v);

update(rc, mid + 1, en, l, r, v);

pull(n);

}

inline ll **combine**(ll a, ll b)

{

return a + b;

}

ll **query**(int n, int st, int en, int l, int r)

{

push(n, st, en);

if (l > en || st > r) return 0; // return null

if (l <= st && en <= r) return t[n];

int mid = (st + en) >> 1;

return combine(query(lc, st, mid, l, r), query(rc, mid + 1, en, l, r));

}

} **st**;

## **Binary Indexed tree(BIT):**

/\* **1'base indexing** \*/

const int N = 3e5 + 9;

ll bit1[N];

ll bit2[N];

ll n;

void update(ll i, ll x, ll \*bit) // O(logn)

{

while (i < N)

{

bit[i] += x;

i += (i & (-i));

}

}

ll query(ll i, ll \*bit) // O(logn)

{

ll sum = 0;

while (i > 0)

{

sum += bit[i];

i -= (i & (-i));

}

return sum;

}

void **rupdate**(ll l, ll r, ll val)

{

update(l, val, bit1);

update(r + 1, -val, bit1);

update(l, val \* (l - 1), bit2);

update(r + 1, -val \* r, bit2);

}

ll **rquery**(ll l, ll r)

{

ll sum1 = query(r, bit1) \* r - query(r, bit2);

ll sum2 = query(l-1, bit1) \* (l-1) - query(l-1, bit2);

return sum1 - sum2;

}

**Mo’s offline Query:** //=> O((N+Q)\*sqrt(N))

const int N = 1e6 + 10;

int rootN;

struct Q

{

int l, r, idx;

};

Q q[N];

bool **comp**(Q q1, Q q2)

{

if (q1.l / rootN == q2.l / rootN) return q1.r > q2.r;

return q1.l / rootN < q2.l / rootN;

}

int main()

{

int n;

cin >> n;

int a[n];

for (int i = 0; i < n; ++i) cin >> a[i];

int query;

cin >> query;

rootN = sqrtl(n) + 1;

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

{

int l, r;

cin >> l >> r;

q[i].l = l;

q[i].r = r;

q[i].idx = i;

}

sort(q, q + query, **comp**);

int curr\_l = 0, curr\_r = -1, l, r;

ll curr\_ans = 0;

vector<ll> ans(query);

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

{

l = q[i].l, r = q[i].r;

--l, --r;

while (curr\_r < r)

{

++curr\_r;

curr\_ans += a[curr\_r];

}

while (curr\_l > l)

{

--curr\_l;

curr\_ans += a[curr\_l];

}

while (curr\_l < l)

{

++curr\_l;

curr\_ans -= a[curr\_l];

}

while (curr\_r > r)

{

--curr\_r;

curr\_ans -= a[curr\_r];

}

ans[q[i].idx] = curr\_ans;

}

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

{

cout << ans[i] << endl;

}

return 0;

}

**Disjoint Set Union(DSU):**  //=>O(1)

**Applications**: 1) Cycle detection. 2) Connected Components in graph. 3) MST(Minimum Spanning Tree).

const int N = 1e5 + 10;

int par[N];

int Size[N];

multiset<int> grpSize;

int **Find**(int v) // returns the representative of the set that contains the element v

{

if (par[v] == v) return v;

return par[v] = Find(par[v]); // Path Compression

}

void **margeGrpSize**(int repU, int repV)

{

grpSize.erase(grpSize.find(Size[repU]));

grpSize.erase(grpSize.find(Size[repV]));

grpSize.insert(Size[repU] + Size[repV]);

}

void **Union**(int u, int v) // merges the two specified sets(u & v)

{

int repU = Find(u);

int repV = Find(v);

if (repU != repV)

{

if (Size[repU] < Size[repV]) swap(repU, repV); // Union by size

par[repV] = repU;

//margeGrpSize(repU, repV);

Size[repU] += Size[repV];

}

}

int main()

{

int u, v, tc, n, k;

cin >> n >> k;

for (int i = 0; i <= n; i++) // Create a new set

{

par[i] = i;

Size[i] = 1;

//grpSize.insert(1);

}

bool cycle = 0;

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

{

cin >> u >> v;

**/\*** // **Finding Cycle**

if(Find(u)==Find(v)) cycle=1; //Cycle is Found;

else Union(u, v); **\*/**

Union(u, v);

}

// if(cycle) cout<<"Found Cycle";

**/\*** //**Count Connected Components**

int ct = 0;

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

{

if (Find(i) == i) ++ct;

}

cout << ct << endl; **\*/**

return 0;

}

## **Lowest Common Ancistor(LCA):**

const int N = 1e5 + 10;

vector<int> g[N];

int table[N + 1][22];

int level[N];

int tin[N], tout[N];

int minLen[N + 1][22], maxLen[N + 1][22]; // maximum ans minimun weight of a tree

int n, lg, Time = 0, INF = 1e9 + 10;

void dfs(int v, int par = -1, int dep = 0, int mn=INF, int mx=-1)

{

tin[v] = ++Time; //for find is\_ancestor

table[v][0] = par;

level[v] = dep;

minLen[v][0] = mn, maxLen[v][0] = mx;

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

{

if (table[v][i - 1] != -1)

{

table[v][i] = table[table[v][i - 1]][i - 1];

//minLen[v][i] = min(minLen[v][i - 1], minLen[table[v][i - 1]][i - 1]);

//maxLen[v][i] = max(maxLen[v][i - 1], maxLen[table[v][i - 1]][i - 1]);

}

}

for (auto &child : g[v])

{

if (child == par) continue;

dfs(child, v, dep + 1);

//dfs(child.first, v, dep + 1, child.second, child.second); //for max & min

}

tout[v] = ++Time;

}

void **lca\_build()** //=> O(n\*logn)

{

dfs(1);

}

int **lca\_query**(int a, int b) //=> O(logn)

{

if (level[a] < level[b]) swap(a, b);

// int dis = level[a] - level[b];

// while (dis) //a and b come to the same level

// {

// int i = log2(dis);

// a = table[a][i], dis -= (1 << i);

// }

for (int i = lg; i >= 0; i--) //a and b come to the same level

{

if (table[a][i] != -1 && level[table[a][i]] >= level[b])

a = table[a][i];

}

if (a == b) return a;

for (int i = lg; i >= 0; i--)

{

if (table[a][i] != -1 && table[a][i] != table[b][i])

a = table[a][i], b = table[b][i];

}

return table[a][0];

}

int **dist**(int u, int v) // distance between two node

{

int l = lca\_query(u, v);

return level[u] + level[v] - (level[l] << 1); //level[l]\*2

}

int kth(int u, int k)

{

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

if (k & (1 << i)) u = table[u][i];

return u;

}

int **findKth**(int u, int v, int k) // kth node from u to v, 0th node is u

{

int l = lca\_query(u, v);

int d = level[u] + level[v] - (level[l] << 1);

if (level[l] + k <= level[u]) return kth(u, k);

k -= level[u] - level[l];

return kth(v, level[v] - level[l] - k);

}

bool **is\_ancestor**(int u, int v) //u is an ancestor of v

{

return tin[u] <= tin[v] && tout[u] >= tout[v];

}

void reset()

{

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

{

g[i].clear();

level[i] = 0;

for (int j = 0; j <= lg; j++) table[i][j] = -1;

}

}

int main()

{

cin >> n;

lg = log2(n) + 1;

reset();

// Input …. Query …

}

**Trie:** //=> O(length)

struct node

{

node \*next[26];

bool completedWord;

node()

{

completedWord = false;

for (int i = 0; i < 26; i++)

next[i] = NULL;

}

} **\*root**;

void **trieInsert**(string s)

{

node \*cur = root;

for (int i = 0; i < s.size(); i++)

{

int x = s[i] - 'a';

if (cur->next[x] == NULL)

{

cur->next[x] = new node();

}

cur = cur->next[x];

}

cur->completedWord = true;

}

bool **trieSearch**(string s)

{

node \*cur = root;

for (int i = 0; i < s.size(); i++)

{

int x = s[i] - 'a';

if (cur->next[x] == NULL)

{

return false;

}

cur = cur->next[x];

}

return cur->completedWord;

}

void **reset**(node\* cur) //Delete the tree after every test case for memory efficiency

{

for(int i=0; i<26; i++)

if(cur->next[i])

reset(cur->next[i]);

delete(cur);

}

# **String:**

## **Double Hashing:**

const int N = 1e6 + 5;

const int Base1 = 137, Base2 = 277;

const int mod1 = 127657753, mod2 = 987654319;

bool isCalPow = 0;

pair<ll, ll> po[N];

void generatePower() // Storing the power of the Base.

{

po[0].first = 1, po[0].second = 1;

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

{

po[i].first = (po[i - 1].first \* Base1) % mod1;

po[i].second = (po[i - 1].second \* Base2) % mod2;

}

}

struct Hashing

{

vector<pair<ll, ll>> prefix, suffix;

int n;

void **generatePrefixHash**(string &s)

{

prefix[0].first = s[0], prefix[0].second = s[0];

for (int i = 1; i < s.size(); i++)

{

prefix[i].first = ((prefix[i - 1].first \* Base1) + s[i]) % mod1;

prefix[i].second = ((prefix[i - 1].second \* Base2) + s[i]) % mod2;

}

}

void **generateSuffixHash**(string &s)

{

suffix[n - 1].first = s[n - 1], suffix[n - 1].second = s[n - 1];

for (int i = n - 2; i >= 0; i--)

{

suffix[i].first = ((suffix[i + 1].first \* Base1) + s[i]) % mod1;

suffix[i].second = ((suffix[i + 1].second \* Base2) + s[i]) % mod2;

}

}

pair<ll, ll> **generateHash**(string &s) // return hash value of a string

{

pair<ll, ll> H = {0, 0};

for (auto &c : s)

{

H.first = ((H.first \* Base1) + c) % mod1;

H.second = ((H.second \* Base2) + c) % mod2;

}

return H;

}

pair<ll, ll> **getPrefixRangeHash**(int l, int r) // return hash value of a range

{

if (l == 0) return prefix[r];

pair<ll, ll> Hs;

Hs.first = (prefix[r].first - (prefix[l - 1].first \* po[r - l + 1].first % mod1) + mod1) % mod1;

Hs.second = (prefix[r].second - (prefix[l - 1].second \* po[r - l + 1].second % mod2) + mod2) % mod2;

return Hs;

}

pair<ll, ll> **getSuffixRangeHash**(int l, int r) // return hash value of a range

{

if (r == n - 1) return suffix[l];

pair<ll, ll> Hs;

Hs.first = (suffix[l].first - (suffix[r + 1].first \* po[r - l + 1].first % mod1) + mod1) % mod1;

Hs.second = (suffix[l].second - (suffix[r + 1].second \* po[r - l + 1].second % mod2) + mod2) % mod2;

return Hs;

}

pair<ll, ll> **concat**(pair<ll, ll> &hash1, pair<ll, ll> &hash2, int len) //len = 2nd string size

{

return {((hash1.first \* po[len].first) + hash2.first) % mod1, ((hash1.second \* po[len].second) + hash2.second) % mod2};

}

void **build**(string &s)

{

n = s.size();

prefix.resize(n), suffix.resize(n);

generatePrefixHash(s);

// generateSuffixHash(s);

if (!isCalPow) generatePower(), isCalPow = 1;

}

} **Hash**;

void solve()

{

int n, m;

string s1, s2;

s1 = "abcabababc", s2 = "abc";

// cin >> s1 >> s2;

n = s1.size();

Hash.build(s1);

pair<ll, ll> hashOfS2 = Hash.generateHash(s2);

for (int i = 0; i + s2.size() <= s1.size(); i++)

{

if (Hash.getPrefixRangeHash(i, i + s2.size() - 1) == hashOfS2)

{

cout << i << "\n";

}

}

return;

}

# **Others:**

## **Sorting pair Using Compare Function:** =>O(n\*log(n))

If vector<pair<ll, ll>> **vec**{{3, 4}, {1, 2}, {3, 5}, {3, 2}, {6, 1}};

bool **cmp**(pair<ll, ll> a, pair<ll, ll> b)

{

if (a.first != b.first) return a.first < b.first; //=>**first** value **increasing** order;

return a.second > b.second; //=> **second** value **descending** order;

}

sort(vec.begin(), vec.end(), **cmp**); //=> **vec**={{1,2}, {3,5}, {3,4}, {3,2}, {6,1}};

## **Minimum fraction:**

If **a/b = c/d** => ex: 12/18 = 2/3

c = **a /\_\_gcd(a,b)**; d = **b /\_gcd(a,b);**

## **Find N’th Fibonacci number using Binet's Formula: => O(1)**

int fib(int n){

double **phi = (sqrt(5) + 1) / 2;**

return **round(pow(phi, n) / sqrt(5));**

}

## **Count words in a string using stringstream:**

#include<sstream>

#include<string>

int countWords(string str)

{

stringstream sf(str);

string word;

int count = 0;

while (sf >> word)

{

count++; // <= you can change statement

}

return count;

}

**Find SubString of a stirng: => O(n^2)**

str = ”abcd” **=>** a, ab, abc, abcd, b, bc, bcd, c, cd, d.

for (int i = 0; i < str.length(); i++)

{

string subStr;

for (int j = i; j < str.length(); j++)

{

subStr += str[j];

cout << subStr << endl;

}

}

## **Find SubSequences / SubSet using Iterative:** **=> O(n\* 2^n)**

{1, 2, 3} => {1}, {2}, {1, 2}, {3}, {2, 3}, {1, 2, 3}, {4}, {1, 4}, {2, 4}, …

vector<vector<int>> subsets(vector<int> &nums)

{

vector<vector<int>> allSubSets;

int n = nums.size();

///=> In **Bits SubSets**, the nums array is which Bit position you want for SubSets;

for (int i = 0; i < (1 << n); i++) //for 2^n possible solution

{

vector<int> subset;

///int tempA=a, tempB=b; ///=> for Bits SubSets

for (int j = 0; j < n; j++) //for nums array

{

if (i & (1 << j))

{

subset.push\_back(nums[j]);

///tempA |= (1LL << nums[j]); ///ON the nums[j] position Bit in tampA

}

///else tempB |= (1LL << nums[j]); ///ON the nums[j] position Bit in tampB

}

allSubSets.push\_back(subset);

///ans=max(ans, temA\*tempB); //qn needed operation

}

return allSubSets;

}

## **Find SubSequences / SubSet using Recursion:** **=> O(n\* 2^n)**

s =”abc” **=>** **subsequences** = { “a”,” b”, “c”, “ab”, “bc”, “ac”, “abc”};

vector<string> **subsequences**;

void **AllSubsequences**(string &s, string subseq="", int index=0)

{

if (index == s.length())

{

subsequences.push\_back(subseq);

return;

}

**AllSubsequences**(s, subseq, index + 1);

**AllSubsequences**(s, subseq + s[index], index + 1);

}

## **\_\_int128 Data-type:**

\_\_int128 **read**()

{

\_\_int128 x = 0, f = 1;

char ch = getchar();

while (ch < '0' || ch > '9')

{

if (ch == '-') f = -1;

ch = getchar();

}

while (ch >= '0' && ch <= '9')

{

x = x \* 10 + ch - '0';

ch = getchar();

}

return x \* f;

}

void **print**(\_\_int128 x)

{

if (x < 0) putchar('-'), x = -x;

if (x > 9) print(x / 10);

putchar(x % 10 + '0');

}

string **\_\_int128toString**(\_\_int128 num)

{

auto tenPow18 = 1000000000000000000;

string str;

do

{

long long digits = num % tenPow18;

auto digitsStr = to\_string(digits);

auto leading0s = (digits != num) ? string(18 - digitsStr.length(), '0') : "";

str = leading0s + digitsStr + str;

num = (num - digits) / tenPow18;

} while (num != 0);

return str;

}

bool **cmp**(\_\_int128 x, \_\_int128 y) { return x > y; }

* **Subarrays/Substring**: A subarray is a contiguous part of array and maintains relative ordering of elements. For an array/string of size n, there are **n\*(n+1)/2** non-empty subarrays/substrings. **[“1234” => {1,2}, {1,2,3}, {2,3,4} etc.]**
* **Subsequence**: A subsequence maintain relative ordering of elements but may or may not be a contiguous part of an array. For a sequence of size n, we can have (**2^n)-1** non-empty sub-sequences in total. **[“1234” => {1,2,4}, {2,4} etc.]**
* **Subset**: A subset **MAY NOT** maintain relative ordering of elements and can or cannot be a contiguous part of an array. For a set of size n, we can have **(2^n)** sub-sets in total. **[“1234” => {1,3,2}, {4,2,3} etc.]**
* **Co-Prime:** That means a pair of numbers are said to be co-prime when they have their highest common factor as 1. [i.e**: gcd(A, B)=1**; ]
* **Lexicographic or Lexicographically:** means sorting in the natural order / dictionary order. [ Ex: “a” **<** ”b”**;** “aa” **<** “ab”**;** “aaab” **<** ”ab”**;** “abcd” **<** ”baa”**;** ]
* **Parity**: is a term used to refer to the property of being even or odd.
* **Permutations**: are often used to count the **number of ways to arrange** a certain number of objects. The number of permutations of a set of n objects is given by **n!**.
* **MEX:** usually refers to the "minimum excluded value" of a set. Given a set of non-negative integers, the MEX is the smallest non-negative integer that is not present in the set. Ex: {0, 1, 3, 4, 7} => MEX is 2;