Time & space complexity

- TC != time taken
- It depends on system

⇒ Then what is time complexity
↳ Rate at which the time taken increases with respect to the input size

- TC → Big-Oh notation - O(n)
  n → input size

ex: old Window vs New Macbook

- TC, worst case scenario
  avoid constants
  avoid lower values
- Best case : omega notation (Ω)
  Average case : Theta notation (Θ)
  worst case : Big-oh (O) (upper Bond)

Space complexity

↳ memory space
↳ Big-O notation
↳ Auxiliary space + input space

- Auxiliary space: space that we take to solve the problem
- input space: space that we take to store the input

Ex:

```
for(int i=0 ; i<N ; i++)
  { for (int j=0 ; j<N; j++)
    {
      ---
    }
  }
```

⇒ run N^2 times $O(n^2)$

Ex:

```
for(i=0 ; i<n ; i++)
  { for (j=0 ; j<=i ; j++)
    {
      ---
    }
  }
```

3 3

i=0 (j=0)
i=1 (j=0,1)
i=2 (j=0,1,2)
⋮
i=n-1 (j=0,1,2.. n-1)

$\therefore 1+2+3+\cdots n = \frac{n(n+1)}{2}$
$\simeq O(n^2)$

Build-up Logical Thinking

⇒ Learning important Pattern Problem

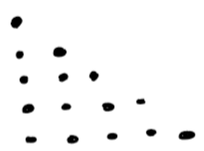
Pattern-1

```
****
****
****
****
```

no of row = 4
no of column = 4

```
for(i=0;i<4;i++){
   for(j=0;j<4;j++){
      cout<<"*"<<
   }
   cout<<endl;// move to new line
}
```

Pattern-2



no. of row = 5
no. of colun = 5
connection b/w row and column is

$r_0 \to 1$
$r_1 \to 2$
$r_2 \to 3$
$r_3 \to 4$
$r_4 \to 5$

```
for(i=0;i<n;i++){
 for(int j=0;j<=i;j++){
   cout<<"*"<<;
 }
 cout<<endl;
}
```

Pattern-3

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

```
for(int i=1;i<=n;i++){
 for(int j=1;j<=i;j++){
   cout<<j<<" ";
 }
 cout<<endl;
}
```

Soln
$R_1 \to 1$
$R_2 \to 1\ 2$
$R_3 \to 1\ 2\ 3$
$R_4 \to 1\ 2\ 3\ 4$
$R_5 \to 1\ 2\ 3\ 4\ 5$

For every row, column is going to row itself

Pattern-4

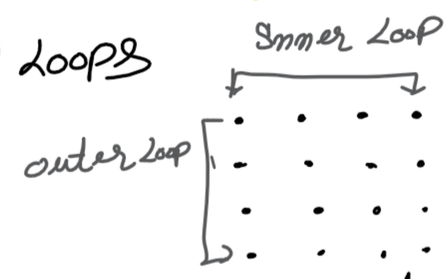
```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

Soln. same as Pattern-3 But, we are Printing the row number

```
for(int i=1;i<=n;i++){
 for(int j=1;j<=i;j++){
   cout<<i<<" ";
 }
 cout<<endl;
}
```

⇒ Trick for solving any Pattern

- nested loops



Rule:

(1) for the outer loop, count the no. of row
(2) for the inner loop, focus on the columns & connect them somehow to the row
(3) print them '*' inside the inner for loop
(4) observe symmetry (optional)

Pattern-5

```
* * * * *
* * * *
* * *
* *
*
```

Soln $R_1 \to 5$, $R_2 \to 4$, $R_3 \to 3$, $R_4 \to 2$, $R_5 \to 1$

n - row no. + 1

```
for(int i=1;i<=n;i++){
for(int j=0;j<n-i+1;j++){
  cout<<"*"<<j;
}
cout<<endl;
}
```

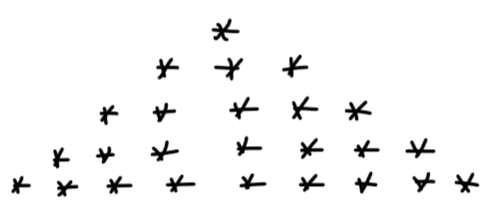
Pattern-6

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

Soln:

```
for(i=1;i<=n;i++){
for(int j=1;j<=n-i+1;j++){
  cout<<j<<" ";
}
cout<<endl;
}
```

Pattern-7



Soln we are Printing space, star, space

	space	star	space
r_0	4	1	4
r_1	3	3	3
r_2	2	5	2
r_3	1	7	1
r_4	0	9	0
	$n-i-1$		$2i+1$

```
for(int i=0;i<n;i++){
 for(int j=0;j<=n-i-1;j++){
   cout<<" ";
 }
 for(int j=0;j<2i+1;j++){
   cout<<"*";
 }
 for(int j=0;j<=n-i-1;j++){
   cout<<" ";
 }
 cout<<endli
}
```

Pattern-8

```
*********
 *******
  *****
   ***
    *
```

Soln:- [space, star, space]

$R_0 \rightarrow$ 0, 9, 0
$R_1 \rightarrow$ 1, 7, 1
$R_2 \rightarrow$ 2, 5, 2
$R_3 \rightarrow$ 3, 3, 3
$R_4 \rightarrow$ 4, 1, 4

i 2n-(2i+1) i

```
for(int i=0; i<n; i++){
  for(int j=0; j<i; j++){
    cout<<" "<<;
  for(int j=0; j<2n-(2i+1); j++){
    cout<<"*"<<;
  for(int j=0; j<i; j++){
    cout<<" "<<;
  }
  cout<<endl;
}
```

Pattern-9

```
    *
   ***
  *****
 *******
*********
*********
 *******
  *****
   ***
    *
```

Here, we observe symmetry in the Pattern:
This Pattern is just a mixture of the last two Pattern (erect Pyramid and inverted Pyramid). So, Firstly we will Print the erect Pyramid and then an inverted Pramid below it

Pattern-10

```
*
**
***
****
*****
****
***
**
*
```

```
for(int i=1; i<=2n-1; i++){
  int star=i;
  if(i>n) star=2n-i;
  for(int j=1; j<=star; j++){
    cout<<"*"<<;
  }
  cout<<endl;
}
```

Soln: no. of row = 9 (odd) = (2n-1) times

∴ outer loop runs (2n-1) times

$R_1 \rightarrow 1$
$R_2 \rightarrow 2$
$R_3 \rightarrow 3$
$R_4 \rightarrow 4$
$R_5 \rightarrow 5$
← till here no. of stars = no. of row

$R_6 \rightarrow 4$
$R_7 \rightarrow 3$
$R_8 \rightarrow 2$
$R_9 \rightarrow 1$
→ (2n-i) → row

Pattern-11

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

$1 \rightarrow R_0$
$0\ 1 \rightarrow R_1$
$1\ 0\ 1 \rightarrow R_2$
$0\ 1\ 0\ 1 \rightarrow R_3$
$1\ 0\ 1\ 0\ 1 \rightarrow R_4$

📌 Pattern Logic:
For even rows (0, 2, 4...), start with 1
For odd rows, start with 0
Alternate the digits using 1 - start

no. of row = 5

[for all the even row start with 1 otherwise 0.

```
int start=1;
for(int i=0; i<n; i++){
  if(i%2==0) start=1;
  else start=0;
  for(int j=0; j<=i; j++){
    cout<<start;
    start=1-start;
  }
  cout<<endl;
}
```

Pattern-12

```
1      1
12    21
123  321
12344321
```

no. of row = 4, outer loop runs 4 times

number, space, number.

$R_1 \rightarrow$ 1, 6 (-2), 1
$R_2 \rightarrow$ 2, 4 (-2), 2
$R_3 \rightarrow$ 3, 2 (-2), 3
$R_4 \rightarrow$ 4, 0 (-2), 4

Same as row no. — This is even number and reduce by 2 ∴ we can say it is 2n-2 times — Same as row no. — But we have to Print it recurr (j--) — This is our 1st number

int space = 2n-2

```
for(int i=1; i<=n; i++){
  //numbers
  for(int j=1; j<=i; j++){
    cout<<j;
  }
  // space
  for(int j=1; j<=space; j++){
    cout<<" ";
  }
  // number
  for(int j=i; j>=1; j--){
    cout<<j;
  }
  cout<<endl;
  space-=2;
}
```

Pattern-13

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

no. of row = 5

```
int num=1
for(int i=1; i<=n; i++)
  for(int j=1; j<=i; j++)
    cout<<num<<" ";
    num=num+1;
  }
  cout<<endl;
}
```

Pattern-14 :-

```
A
AB
ABC
ABCD
ABCDE
```

```
for (int i; i<=n; i++){
for(char ch='A'; ch<='A'+i ;ch++){
 cout << ch << " ";
 }
cout << endl;
```

Pattern-15 :-

```
A B C DE
A B C D
A B C
A B
A
```

```
for(int i=1 ; i<= n; i++){
for(char ch='A'; ch<='A'+(n-i-1);ch++){
  cout << ch << " ";
  }
  cout endl;
}
```

NOTE:

A+B+C+D+E

A + 4

Pattern-16

```
A
B B
C C C
D D D D
E E E E E
```

```
for(int i=1; i<=n; i++){
   char ch='A'+i;
 for(int j=1; j<=i; j++){
   cout << ch << " ";
  }
 cout endl;
 }
```

Pattern-17

```
    A
   ABA
  ABCBA
 ABCDCBA
```

observe symmetry

A B C B A (C: --)

A B C D C B A (D: --)

∴ Break it from middle

∴ (2i+1)/2

[spaces, Alpha, spaces]

n-i-1 , (2i+1) , n-i-1

```
// space
for(int i=0; i<n; i++){
   cout << " ";
// character
  char ch='A';
  int breakpoint =(2i+1)/2
for(int j=1 ; j<=2i+1; j++){
   cout << ch;
  if(j<= breakpoint) ch++;
   else ch--;
 }
// space
for(int j=0; j<n-i-1 ;j++){
   cout << " ";
 }
 cout << endl;
}
}
```

#Pattern-18

```
E
D E
C D E
B C D E
A B C D E
```

```
for int i=1; i<=n; i++){
 for(char ch='E'-i ; ch<='E', ch++)
   cout << ch << " ";
 }
 cout endl;
}
```

NOTE:
The letter will be Printed from 'E'-i to E where i= row index

#Pattern-19

```
**********
****  ****
***    ***
**      **
*        *
*        *
**      **
***    ***
****  ****
**********
```

∴ There is a symmetry.

∴ First solve up side Pattern

stars, space, stars

	stars	space	stars
R_0 →	5	0 (+2)	5
R_1 →	4	2 (+2)	4
R_2 →	3	4	3
R_3 →	2	6	2
R_4 →	1	8	1
	n-i	2n	n-i

```
int space=0
for(int i=0; i<n; i++){
 //star
  for(int j=1; j<=n-i; j++){
    cout << "*";
  }
 //space
  for(int j=0; j<space; j++){
    cout << " ";
  }
  3
 // star
  for(int j=1; j<=n-i; j++){
    cout << "*";
  }
  3
    space+=2
    cout << end;
  3
 int space = 2n-2
for(int i=1; i<=n; i++){
//star
  for(int j=1; j<=i ;j++){
    cout << '*';
  3
// space
  for(int j=1 ; j<=space; j++){
    cout << " ";
  3
// star
  for(int j=1 ; j<=i ;j++){
    cout << '*';
  3
    space-=2;
    cout << endl;
  3
3
```

// solving down Pattern

	star	space	star
R_0	1	8 (-2)	1
R_1	2	6	2
R_2	3	4 (-2)	3
R_3	4	2	4
R_4	5	0	5
	i	2n-2	i

Pattern-20

star, space, star

R1	1	8	1
R2	2	6	2
R3	3	4	3
R4	4	2	4
R5	5	0	5

Till here no. of star = row no.

R1	4	2	4
R7	3	4	3
R8	2	6	2
R9	1	8	1

no. of star = 2n-i

Before 5th row, space is decreasing and after 5th row space is increasing.

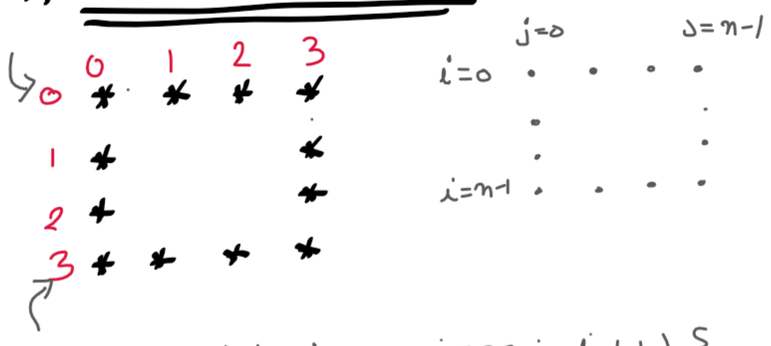
no. of row = 9 (odd)

∴ (2n-1) times it will run

```
int space = 2*n-2;
for(int i=1; i<=2*n-1; i++){
  // stars
   int stars = i
    if(i>n) stars = 2*n-i;
   for(int j=1; j<= stars ;j++){
       cout << "*";

  // spaces:
   for(int j=1; j<= spaces ;j++){
        cout << " ";
    }
  //stars
  for(int j=1;j<= stars;j++){
      cout << "*";
  }
  cout << endl;
  if(i<n) space -=2;
   else space +=2
}
}
```

Pattern-21




```
for(int i=0; i<n; i++){
   for(int j=0; j<n; j++){
   if(i==0||j==0||i==n-1||j==n-1){
       cout << '*';
    }
   else cout << " ";
  }
  cout<<endl;
}
```

Pattern-22

```
4 4 4 4 4 4 4
4 3 3 3 3 3 4
4 3 2 2 2 3 4
4 3 2 1 2 3 4
4 3 2 2 2 3 4
4 3 3 3 3 3 4
4 4 4 4 4 4 4
```

Logic:

- We observe that on the Perimeter of the square, an integer N, which decreases by 1 as we are moving inside the square.
- Subtract the whole Pattern by N.

The outer & inner loop will run for the same number of time i.e (2N-1) time, since we have to print square

⇒ we take the minimum distance of the current cell from each of the end of the square (min(1,3,3,5)) & make the current cell value equal to that number.

for example Let current cell be (I)

∴ The distance of current cell from

top = i
left = j
right = (2n-1)-1-j = (2n-2)-j
bottom = (2n-1)-i = (2n-2)-i

```
for(int i=0; i<2*n-1; i++){
   for(int j=0; j<2*n-1; j++)
       int top=i
       int left=j;
       int right = (2*n-2)-j;
       int bottom =(2*n-2)-i;
       cout << (n-min(min(top,bottom),min(left,right)));
   }
   cout << endl;
}
}
```

STL in C++

- Containers
- Algorithm
- Iterator
- Functions

⇒ `#include <bits/stdc++.h>`

↳ This header files contains all STL

Storing inte using Pair

```
# Void explainPair()
Pair<int,int> P={1,3};
```

To accers

```
P.first = 1
P.second = 3;
cout<<P.first<<" "<<P.second;
```

⇒ When more than two variable we use nested Property of Pair

```
Pair<int, Pair<int,int>> P={1,{3,4}};
```

```
P.first = 1
P.second.first = 3
P.second.second = 4
```

⇒ Pair Array

```
Pair<int,int> arr[] ={{1,2},{3,5},{5,1}};
cout << arr[1].second;
```

Vector

```
Vector<int> V;
```

Vector

v.push_back(val)	Adds element to end
v.pop_back()	Removes last element
v.front()	Returns first element
v.back()	Returns last element
v.size()	Returns number of elements
v.clear()	Clears the vector
v.empty()	Checks if empty
v[i], v.at(i)	Access element at index
v.begin(), v.end()	Iterators to range
v.insert(pos, val)	Insert at position
v.erase(pos)	Erase at position

List

lst.push_front(val)	Insert at front
lst.push_back(val)	Insert at back
lst.pop_front()	Remove from front
lst.pop_back()	Remove from back
lst.insert(it, val)	Insert before iterator
lst.erase(it)	Erase element at iterator
lst.remove(val)	Remove all val
lst.sort()	Sort the list
lst.reverse()	Reverse the list
lst.unique()	Remove consecutive duplicates

Deque

dq.push_front(val)	Insert at beginning
dq.push_back(val)	Insert at end
dq.pop_front()	Remove from front
dq.pop_back()	Remove from back
dq.front()	First element
dq.back()	Last element
dq.insert(pos, val)	Insert at position
dq.erase(pos)	Erase from position
dq.clear()	Clear all elements
dq.empty()	Check if empty
dq.size()	Number of elements

Stack

st.push(val)	Add to top
st.pop()	Remove top element
st.top()	Access top element
st.empty()	Check if empty
st.size()	Number of elements

STL Algorithms

sort(start, end)	Sorts range
reverse(start, end)	Reverses range
count(start, end, val)	Count val in range
find(start, end, val)	Finds val
accumulate(start, end, init)	Sum of range
binary_search(start, end, val)	Check if val exists (sorted)
lower_bound(start, end, val)	First position >= val
upper_bound(start, end, val)	First position > val
next_permutation(start, end)	Next lexicographic permutation
all_of(start, end, cond)	True if all satisfy cond
any_of(start, end, cond)	True if any satisfy cond
none_of(start, end, cond)	True if none satisfy cond

Queue

q.push(val)	Add to back
q.pop()	Remove front element
q.front()	Access front
q.back()	Access back
q.empty()	Check if empty
q.size()	Number of elements

Priority Queue

pq.push(val)	Add element
pq.pop()	Remove top element
pq.top()	Top (max by default)
pq.empty()	Check if empty
pq.size()	Number of elements

Set

s.insert(val)	Insert value
s.erase(val)	Remove value
s.find(val)	Find value
s.count(val)	Return 1 if found
s.begin()	Iterator to begin
s.end()	Iterator to end
s.empty()	Check if empty
s.size()	Number of elements

Map

mp[key] = val	Assign value to key
mp.at(key)	Access value at key
mp.insert({k,v})	Insert key-value
mp.erase(key)	Remove key
mp.find(key)	Find key
mp.count(key)	Count of key
mp.begin()	Iterator to begin
mp.end()	Iterator to end

Basic Math for DSA

#Digit Concept / Extraction of Digit

N = 7789 % 10 = 9
/10 → 778 % 10 = 8
/10 → 77 % 10 = 7
/10 → 7 % 10 = 7
0 % 10

Pseudo code:

```
while(N>0){
  lastdigit = N%10
  N = N/10;
}
```

TC = O(log N/10)

(a) Given the number 'n' find out & return the number of digit present in a number

```
int count=0
while(n>0){
int lastdigit = n%10
    count = count+1;
     n = n/10;
}
return count
```

#Reverse Concept

N = 7789
output = 9877

To reverse the number:

Reverse = (revNum × 10) + Lastdigit

Ex: revNum = 0
∴ (0 × 10) + 9 = 9
(9 × 10) + 8 = 98
(98 × 10) + 7 = 987
(987 × 10) + 7 = 9877

#Palindrome Number

Palindrome number is a number which reads the same both left to right and right to left.

Ex: input = 121 → Palindrom
output = 121

input = 123 → No
output = 321

My thinking:
(1) Given N
(2) reverseNum
(3) if(N = reverseNum) cout << True; else false.

Edge Case:
(1) negative no. like -121 (x<0) are never Palindromes
(2) Number ending with 0 but not 0 itself (like 10) are not Palindroms (because 01 ≠ 10) — x%10 == 0 && x != 0

```
if (x<0 || (x%10==0 && x!=0))
    return false;
```

#Armstrong Number

- An armstrong number is a number which is equal to the sum of the digit of the number, raised to the power of the number of digit.

Ex: input n = 153
output = true.
Explanation: Number of digit = 3
∴ $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$
∴ Therefore, it is an Armstrong no.

⇒ My thinking:
(1) find no. of digit
(2) sum of digit of no. raised to Power of digit

sum = sum + power(digit, count)

Use Power(a,b)

```
class Solution {
public:
    bool isArmstrong(int n) {
        int original = n;
        int count = 0;
        int temp = n;

        // Count the number of digits
        while (temp > 0) {
            temp /= 10;
            count++;
        }

        temp = n;
        int sum = 0;

        // Sum the digits raised to the power of count
        while (temp > 0) {
            int digit = temp % 10;
            sum += pow(digit, count);
            temp /= 10;
        }

        return sum == original;
    }
};
```

count

(digit)^count

#Print All Divisors

- All the divisors between [1 to N]

```
for (i=1; i<=n; i++){
  if (N%i==0)
     Print i;
}
```

(1) Run Loop [1 to n]
(2) n%i==0
(3) Print i;

If is i completly divid N
i.e N%i = 0

(1) $O(n)$.

- 2nd APP

```
for(i=1; i<=sqrt(n); i++){
  if(n%i==0){
    Print(i)
    if((n/i)!=i)
      Print(n/i);
}
```

Ex: N = 36

1 × 36
2 × 18
3 × 12
4 × 9
6 × 6 ⇐ √N
9 × 4
12 × 3
18 × 2
36 × 1

- store in vector and sort it.

#Prime Number Check

- exactly 2 factors 1 & itself

Brute force ⇒ O(n)

```
⇒ count=0
  for(i=1; i<=N; i++){
    if (n%i==0)
      count ++;
  }
  if (count==2) ✓
  else X
```

Pseudo code
(1) for 1 ← n
(2) Print divisor
(3) if no. of divisor == 2, then Prime, otherwise not.

= optimal app

```
count=0
for(i=1; i<=√n; i++){
  if (n%i==0){
    count++;
    if (n/i != i)
      count++
  }
if (count==2) ✓
else X
```

⇒ O(√n) times
O(1) space.

Edge-case:
- Number less than or equal to 1 are not Prime
if (n<=1) return 0;

#GCD/HCF

- Greatest common Divisior
- Highest common factor

Ex: $N_1 = 9$, $N_2 = 12$

1, 3, 9 — 1, 2, 6, 12, 3

Common factors = 1, 3

GCD = 3

Ex: $N_1 = 11$, $N_2 = 13$

GCD = 1

Ex: 20, 30

1, 2, 4, 5, 10, 20 — 1, 2, 3, 5, 6, 10, 15, 30

∴ GCD = 10

⇒ Pseudo code / Brute force

- N_1, N_2
- for $1 \leftarrow N_2$
- find common factor($n_1 \% i == 0 \&\& n_2 \% i == 0$)

```
for(i=1 ; i<=min(n1,n2) ; i++)
    if(n1%i==0 && n2%i==0)
        gcd = i;
}
```

⇒ TC = $O(\min(n_1, n_2))$

Euclidean Algorithm:

n_1, n_2

$$gcd(n_1, n_2) = gcd(n_1 - n_2, n_2)$$

where, $n_1 > n_2$

Algorithm:

$$gcd(a,b) \rightarrow gcd(a-b, b) \quad \cdots\cdots 0$$

But better way is

$$gcd(a,b) = gcd(a \% b, b)$$

Logic: greater % smaller → till one of this is zero, then other is gcd

```
while(a>0 && b>0){
    if(a>b)  a = a%b;
    else     b = b%a;
}
if(a==0)  gcd(b)
else      gcd(a)
```

⇒ TC: $O(\log_\phi(\min(a,b)))$

Binary Number System

- Two digit (0/1)
- Binary no. (Base-2)

Decimal to Binary

- repeated division with 2

2 | 42	0	↑
2 | 21	1	
2 | 10	0	
2 | 5	1	
2 | 2	0	
2 | 1	1	
0		

101010

$(42)_{10} = (101010)_2$

Another APP using string or Vector

⇒ Logic

- use modulus (%2) to get Binary digit
- store digit in a string or vector
- Revrse the string/Vector at the end to get the correct binary.

Pseudocode:

```
if n==0 return 0;
binary = empty string/vector
while(n>0):{
    rem = n%2
    binary = to string(rem) + binary
    n = n/2
}
return binary.
```

My thinking:

- Keep dividing number by 2
- Record the remainder at each step
- Stop when the number becomes Zero(0)
- Reverse the collect remainder → that is our Binary number.

```
ans = 0
Power = 1
while(decNum>0) {
    rem = decNum % 2
    decNum = decNum/2
    ans += (rem x pow);
    Pow = Pow x 10;
}
```

Binary to Decimal:

Ex:

1 0 1 0 1 0 ← 2^{pow}

$2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$

```
binNum = 0
ans = 0
Pow = 1 // 2^0
while(binNum>0){
    rem = binNum % 10
    ans += binNum x pow
    binNum = binNum // 10
    Power = Power x 2
}
```

Basic Recursion

- **Recursion:** When a function called itself until a specified condition is met.

- **Base condition:**

It is the condition that is written in a recursive function in order for it to get completed and not to run infinitely (∞).

Ex:
```
int count=0;
Void f(){                          → Base condition
  if(count==3) return;
    Print(count);
    count++;
     f();
  }
main()
{
  Print();
}
```

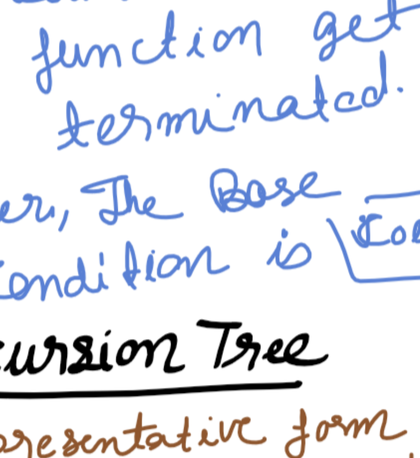
output: ϕ 1 2

count==3, then function get terminated.

∴ Here, The Base condition is `count==3`

- **Recursion Tree**

↳ representative form of recursion which depicts how function are called as returned are called as a series of events as a series of events happening consecutively.



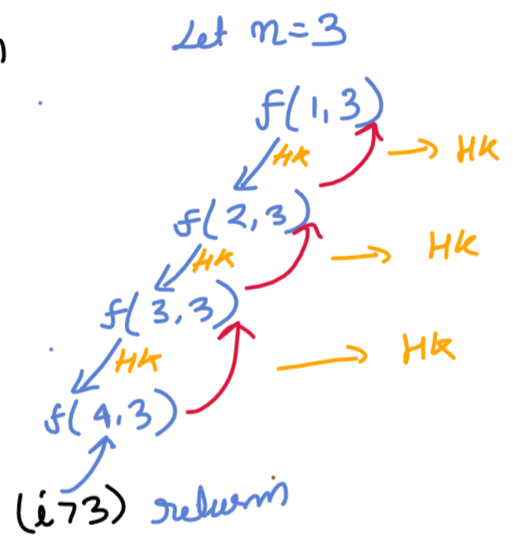
Basic Recursion Problems

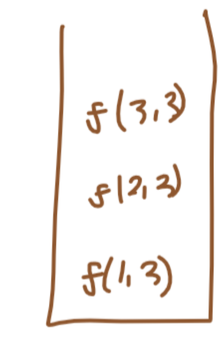
Q(1) Print Name N times using recursion

Pseudo code:

```
Void f(i,n){                       ← Base condition
 if(i>n) return;
   Print('HK');
    f(i+1,n);
 }
main(){
   int n;
   cin>>n;
    f(1,n);
```

⇒ TC = O(n)
SC = O(n)





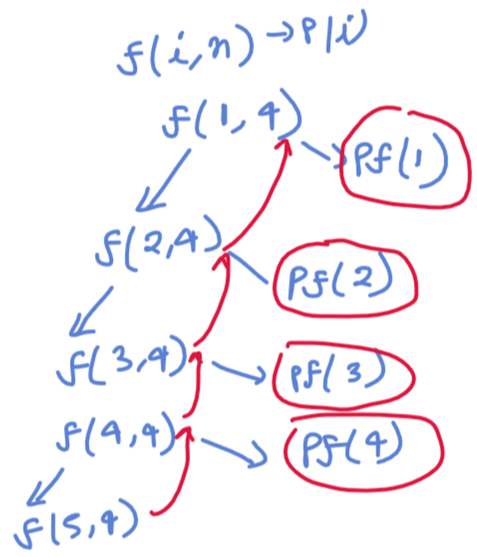
stack space

Q(2) Print from 1 to N

Ex: N=4
Print = 1, 2, 3, 4.

Pseudo code

```
f(i,n){
 if(i>n) return;
  Print(i)
   f(i+1,n);
}
main(){
  input n
   f(1,n)
}
```



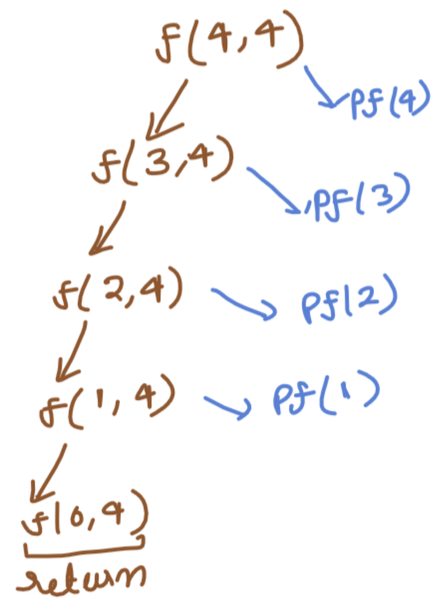
∴ output = 1234

Q(3) Print in term of N → 1

Soln: Ex: n=4
output: 4 3 2 1 (n-1) decrease by 1

Pseudo code:

```
f(i,n){
 if (i<1) return;
   Print(i)
   f(i-1,n);
}
main(){
input (n)
  f(n,n);
}
```



∴ output = 4321

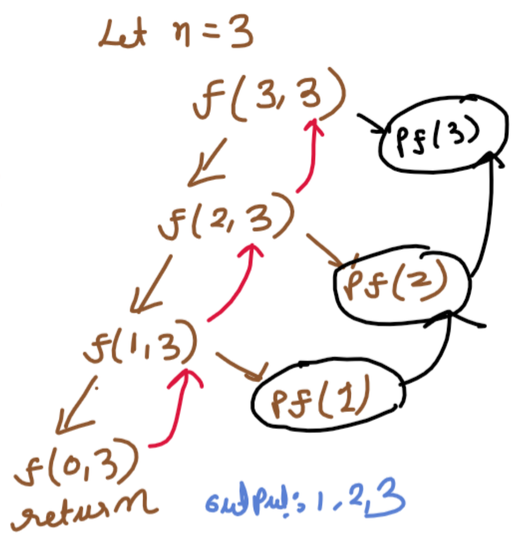
Q(4) Print from 1 to N. using Backtracking

Pseudo code:

```
f(i, n) {
if (i < 1) return;
f(i-1, N);   ⇒ recursive call
  Print (i); ⇒ statement.

}
main( ){
 input (n)
  f(N, N);
}
```

Let n = 3



output: 1, 2, 3

NOTE:

Statement written after recursive call execute in opposite order of function call

f(4) → f(3) → f(2) → f(1)
pf(4) ← pf(3) ← pf(2) ← pf(1)

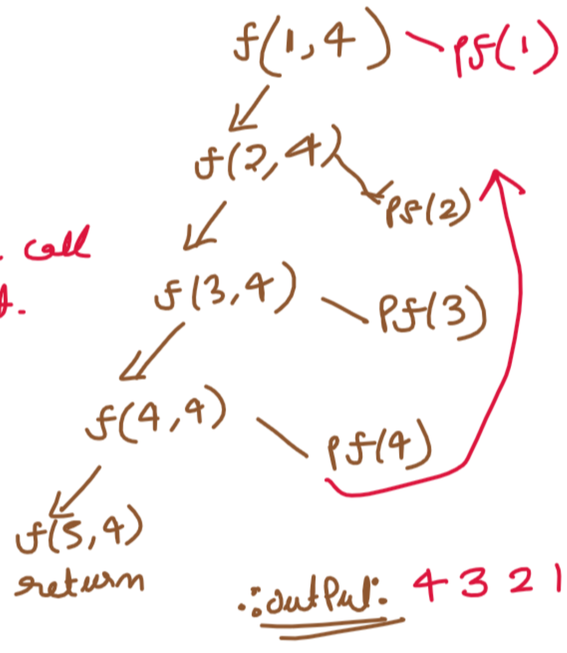
Q(5) Print from N → 1. using Backtracking

Ex: N = 4
output = 4 3 2 1

Pseudo code:

```
f(i, n) {
if (i > n) return;
f(i+1, N);   ⇒ recursive call
  Print (i); ⇒ statement.

}
main( ){
 input (n)
  f(1, N);
}
```



∴ output: 4 3 2 1

Q6 Sum of first N number using recursion

Method-1: Parameterised

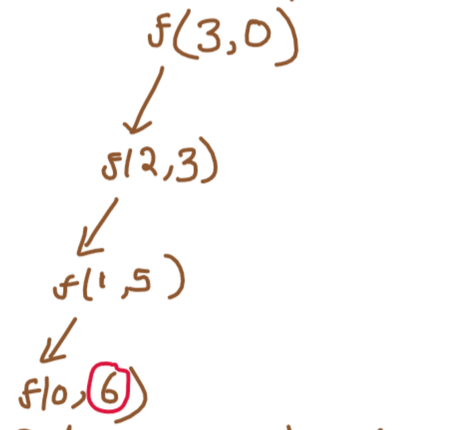
Pseudo code:

```
f(i, sum) {
 if (i < 1)
    Print (Sum)
    return;
 f(i-1, sum+i)
}
main( ){
 input n
   f(n, 0)
}
```

initial sum = 0

N = 3 (1, 2, 3)
output = 1 + 2 + 3 = 6

Recursion Tree



return sum i.e 6 Ans

TC = O(N)
SC = O(N)

Method-2 → Functional recursion

Pseudo code:

```
f(n) {
 if (n == 0) return 0;
 return n + f(n-1);
}
main ( ){
  input n
    f(n);
}
```

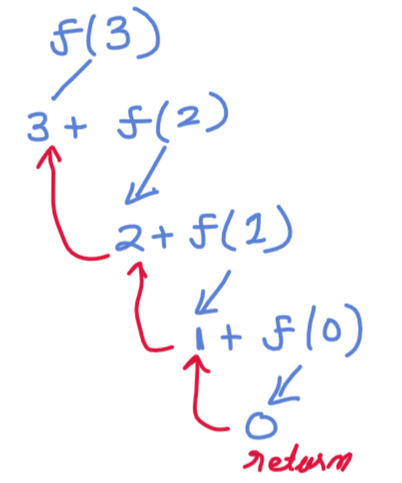
Think like

N = 3 ? 1, 2, 3

3 + f(2)
2 + f(1)
1 + f(0)

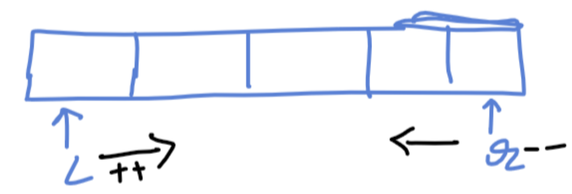
means: n + f(n-1)

Recursion Tree:

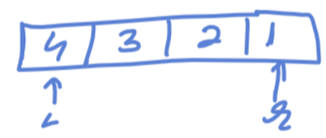


∴ output = 3 + 2 + 1 = 6

Q7. Reverse an Array, using recursion



swap l & r, again
swap l++ & r--



Pseudo code

```
f(l, r) {
  if (l >= r) return;
swap(a[l], a[r]);
   f(l+1, r-1)
}
main ( ){
  input array
   f(0, n-1);
}
```

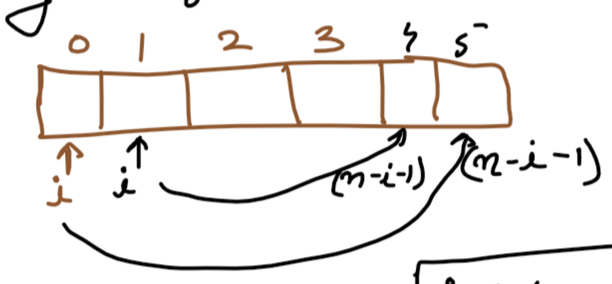
first indx last indx.

1 2 3 4
reverse array.

Ex:

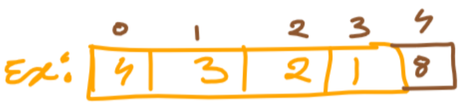
Recursion Tree:

⇒ Using single variable to reverse

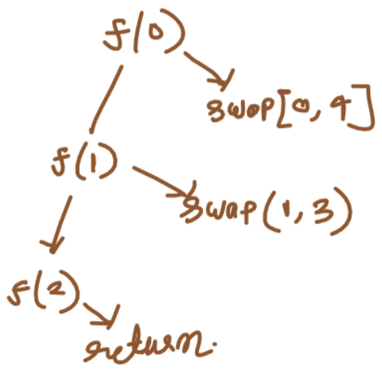


Means, we have to swap **i & (n-i-1)**

Ex:




```
f(i) {
 if (i >= n/2) return;
 swap(a[i], a[n-i-1]);
   f(i+1);
}
main( ){
  input arr;
   f(0)
}
```



Q8 Check if a string is Palindrome.

Palindrome: A string on reversal read the same.

Ex: 'MADAM' = MADAM

Equal, then return true,
Otherwise false.
we will compare till $i >= \frac{n}{2}$ (middle of string)

```
f(i){
  if (i >= n/2) return true;
  if (s[i] != s[n-i-1]) return false;
  return f(i+1, s);
}
main(){
    input string;
    Print f(0, s);
}
```

n = size of string = s.size.

Multiple Recursion calls

Example: Fibonacci Number

0, 1, 2, 3, 5, 8, 13, 21, ...

0 → 1st, 1 → 2nd, 2 → 3rd, ... → f(5)

$\therefore f(5) = f(3^{rd}) + f(2)$

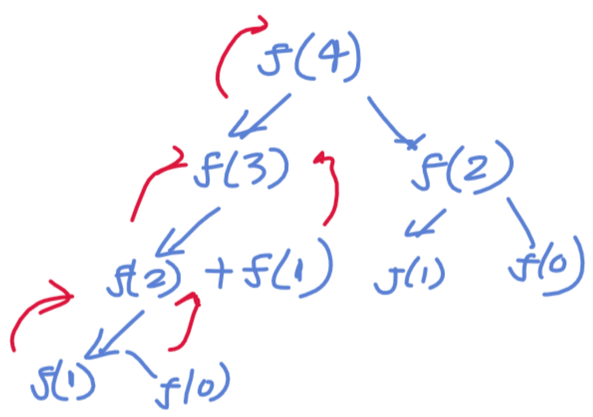
$f(n) \Rightarrow f(n-1) + f(n-2)$

```
f(n){
  if (n<=1) return n;
  return f(n-1) + f(n-2);
}
main(){
  input n
  f(n)
}
```

(f(n-1): last; f(n-2): 2nd last.)

Recursion Tree

Let n = 5



TC: ≈ $O(2^n)$

Print all Subsequences:

A contiguous/non-contiguous sequence, which follows the order.

Common Mistake

Subset: Set (no order)

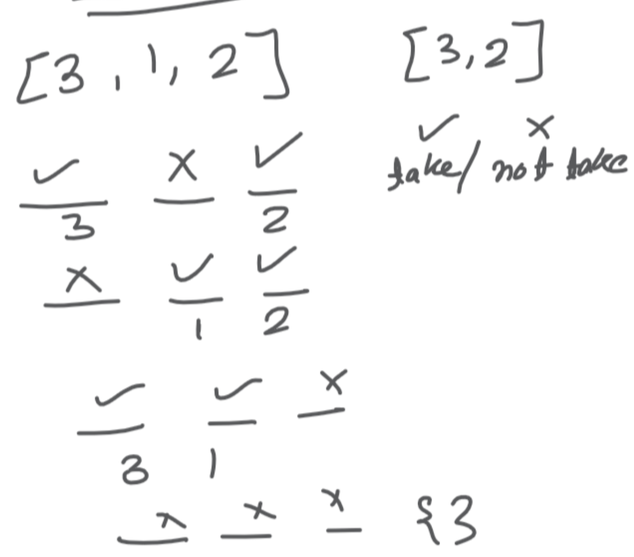
Subsequence: Sequence (order)

Ex: arr[3,1,2] ⇒ Subsequence

3
1
2
3 1
1 2
3 2
3 1 2
{} → null

→ 8 element.

Pattern:



Ex: {3, 1, 2}

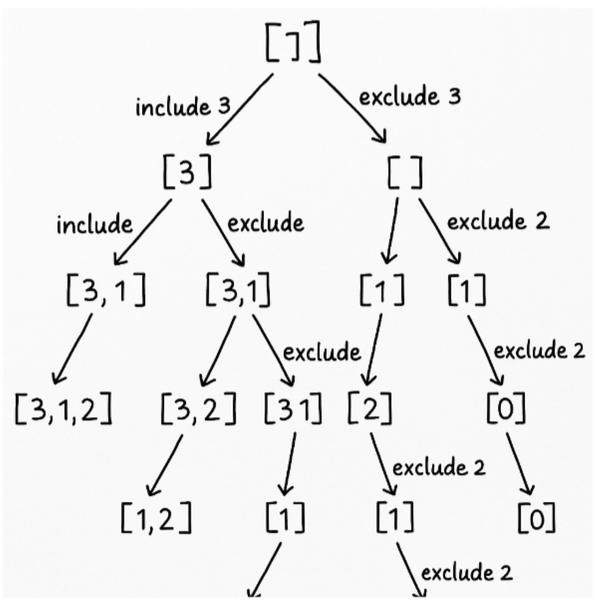
Recursive way

arr [3, 1, 2]
0 1 2 ← n

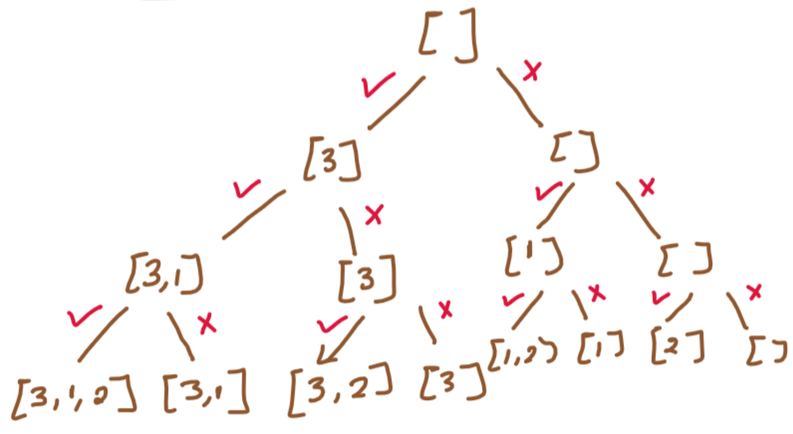
Pseudo code

```
f(index, a[])          // a[] : Empty
if (index >= n)
    Print a[]
    return;
// include current element
a[].Pushback(arr[i]);
f(indx+1, a[]);        => // taken
a[].remove(arr[i]);
// exclude current element.
f(index+1, a[]);       => // not taken
}
```

NOTE: Total subsequence = 2^n, including empty.



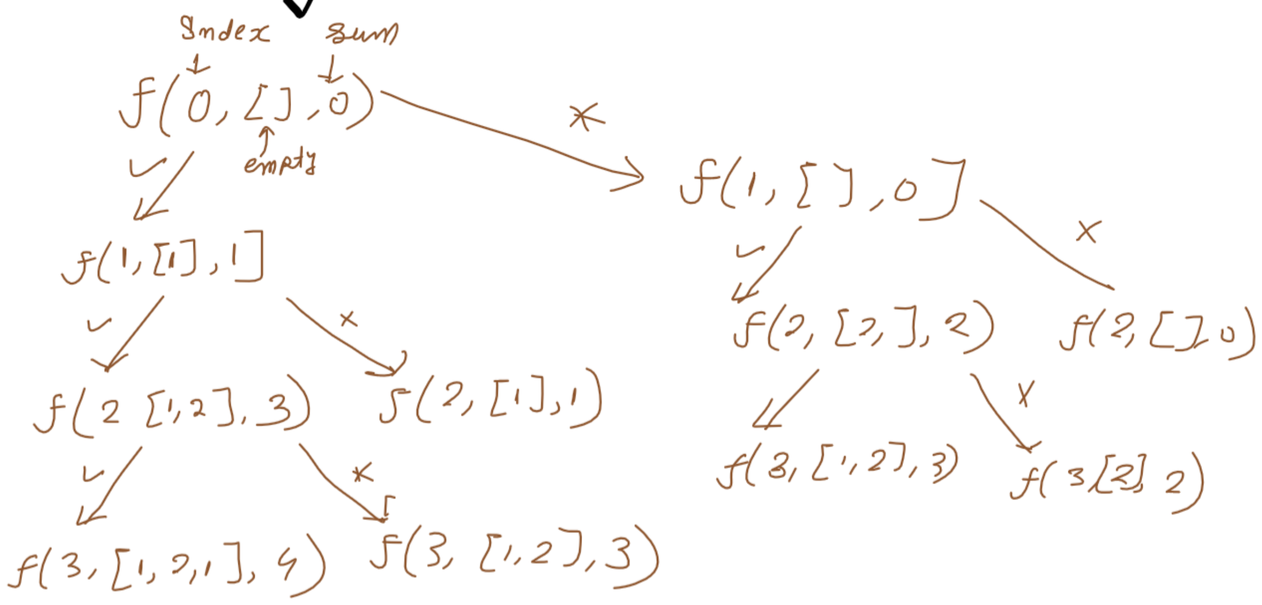
Recursion Tree



NOTE: Kaise Yaad Rakhe:

- Har level ek index ko represent krta hai
- Har level per include(✓) ya exclude(✗) dono call jaate hai
- Jab index == arr.size(), Print ya Store karo a[].

Printing Subsequences whose sum is k.




```
f(i, [], s)
{ if (i==n){
   if (s==sum)
     Print(ds)
   return;
}
   ds.add(arr[i]);
   s+ =arr[i]
   f(i+1, ds, s)
   ds.remove(arr[i])
   s-= arr[i];
   f(i+1, ds, s)
}
```

⇒ taken

⇒ not taken

Print any 1 subsequence whose sum is sum.

⇒ Technique to print only one answer

Logic:

```
f(){
Base condition
satisfied
return true;
else false.
if (f()==true)
    return;
  f()
}
```

Q Count the subsequences with sum = k

```
f( )
{
 Base case
return 1 → condition satisfied
return 0 → condition not sat

left = f()
right = f()
return left+right;
}
```

Hashing

Applications of Hashing using `map` / `unordered_map` in C++:

1. Counting frequency of elements
2. Detecting duplicates in array or string
3. Finding first non-repeating character
4. Checking if two strings are anagrams
5. Storing prefix sums
6. Memoization in dynamic programming
7. Tracking visited nodes/states
8. Storing frequency of characters (`map<char, int>`)
9. Finding number of subarrays with given sum
10. Finding pairs with a given difference

Ex: 1 | 2 | 1 | 3 | 2

1 → 2 times
3 → 1 times
4 → 0 times
2 → 2 times
10 → 0 times

Brute force app

- Run for Loop to count number

```
f(number, a[])
    count = 0
    for (i = 0; i < n; i++) {
        if a[i] == number)
            count += 1;
    }
    return count;
}
```

Tc: O(n)

NOTE: We may encounter a Problem where the maximum array element may be very large like 10^9. The maximum size of an array can be the following.

Array Dclc	Max size (int)	Max size (Boolean)
inside main	10^6	10^7
Globally	10^7	10^8

Optimized approach using Hashing:

Hashing: Pre storing / fetching

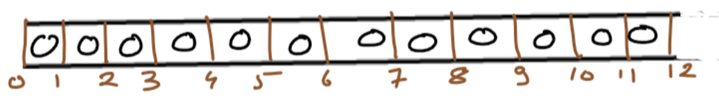
Example:

Assume: The maximum element in the given array can be 12.

Step-1: Pre-storing: we will create an array (named hash array) of size 13, so that we can get the index 12.

n = 13

Hash array



. initially 0 in Hash array

Pseudo code

```
input n //size of array
Declare arr of size n
for i = 0 to n-1
    input arr[i]
// Precompute fregeuncy
Declare has[13] & initialize all to 0
for i = 0 to n-1
    has[arr[i]] += 1
input q // number of queries
while q > 0
    input number
    // fetching:
    output has[number]
    q = q - 1
```

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

int main() {

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

    //precompute:
    int hash[13] = {0};
    for (int i = 0; i < n; i++) {
        hash[arr[i]] += 1;
    }

    int q;
    cin >> q;
    while (q--) {
        int number;
        cin >> number;
        // fetching:
        cout << hash[number] << endl;
    }
    return 0;
}
```

Char hashing

Ex: s = "abcdabefc" ⇒ Optimized approach using Hashing:

queries { a → 2, c → 2, 2 → 0 }

To map char to integer, we use the ASCII value of respective char.

∴ a = 97 ⇒ int x = 'a'
A = 65 ⇒ int x = 'A'

Brute force:

```
f(char c, S[]) {
    count = 0
    for (i = 0; i < n; i++)
    {
        if (S[i] == c)
            count++;
    }
    return count
```

Tc: O(n)

Case-1: only Lower case:

Value = char - 'a'
n = 25
char has has size = 26
∴ Pre-Storing = has[S[i] - 'a'] += 1
Fetching = has[char - 'a']

Case-2: only upper case:
same as case 1

***

Case-3: both upper case & Lower case:

Total characters = 256
∴ has array size = 256
We will not subtract anything from the given character.
⇒ Pre-storing = has[S[i]] += 1
Fetching = has[character]

The case 3 method also applies to case 1 & case 2.

In character hashing limit will not cross 256. so we will always use this method.

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

int main() {

    string s;
    cin >> s;

    //precompute:
    int hash[256] = {0};
    for (int i = 0; i < s.size(); i++) {
        hash[s[i]]++;
    }

    int q;
    cin >> q;
    while (q--) {
        char c;
        cin >> c;
        // fetch:
        cout << hash[c] << endl;
    }
    return 0;
}
```

STL Map & unordered map in C++

arr = [1, 2, 3, 1, 3, 2]

number is key

map<key, value>

number — frequency

3 → 1 2
2 → 1 2
(1) → 1 2

arr = | 1 | 2 | 3 | 1 | 3 | 2 |

mpp[1]++

mpp{arr[i]}++

∴ Pre-store:

```
map<int, int> mp;
for(int i=0; i<n; i++){
  map[arr[i]++;
```

fetches:

mapp[number]

TC → map ⇒ O(N)

NOTE: Map store all the value in sorted ordered. / unordered_map does not follow any specific order

For char Hashing using map

```
key = char
map<char, int>
mpp[s[i]]++;
```

NOTE: Most of the time we will use unorder_map, if it show time limit exceed, then we go for map

Collision :-

Hashing is done using these method

(1) Division Method → important.
(2) Folding Method
(3) Mid-square method

} Not important for interview or coding round

Division Method

It computes the hash index using formula

hash(key) = key % m

m — size of hash table

NOTE: whatever method the map is using, if all the element go to the same hash index, we will call it a case of collision.

Q Counting Frequencies of Array element

Pseudo code:

```
function count(arr):
  create empty map freq
  for each element in arr:
    freq[element] += 1
  for each key in freq:
    Print key → freq[key]
```

Q Find the highest/Lowest frequency element.

Pseudo code:

(1) input array A
(2) create an empty hash map (freq_map)
(3) for each element in A:
  freq_map[element] += 1
(4) Initialize:
  max_freq = 0, min_freq = n+1
  max_elem = -1, min_elem = -1
(4) for each (element, count) in freq_map
  if count > max_freq:
    max_freq = count
    max_elem = elem
  if count < min_freq:
    min_freq = count
    min_elem = elem
(5) Print High & Low frequency

Important Sorting Technique

Selection Sort:-

0	1	2	3	4	5
13	46	24	52	20	9

no. of element = 6

(1) Select the minimum i.e 9
(2) Swap

9 46 24 52 20 13 → Step-1
9 13 24 52 20 46 → Step-2
9 13 20 52 24 46 → Step-3
9 13 20 24 52 46 → Step-4
⇒ 9 13 20 24 46 52 → Step-5

sorted

Observation:

swap at index 0, & min index [0 → n-1]
swap at index 1, & min index [1 → n-1]
swap at index 2, & min index [2 → n-1]
⋮
n-2 . & min index [n-2 → n-1]

Pseudo code:

```
for(i=0; i<=n-2; i++)
  {
    min=i
  for(j=i; j<n-1; j++)
    {
      if([arr[j] < arr[min]) min=j
    }
    swap(arr[min], arr[i]);
```

Dry Run Example:-

5	3	6	2	1
0	1	2	3	4

i → 0 1 2 3 4

i=0 → min = 4 (Value 1) → Swap A[0] & A[4] → [1, 3, 6, 2, 5]
i=1 → min = 3 (Value 2) → Swap A[1] & A[3] → [1, 2, 6, 3, 5]
i=2 → min = 3 (Value = 3) → Swap A[2] & A[3] → [1, 2, 3, 6, 5]
i=3 → min = 4 (Value = 5) → Swap A[3] & A[4] → [1, 2, 3, 5, 6]

⇒ Logic

"Fix the smallest element in beginning, one by one".

Mtlb:
(1) Pehle Position (0) Pe sabse chhoti value lao.
(2) Fir doosri Position (1) Pe uske baad wali smallest value lao.
(3) Aise hi har step per minimum dhoondho aur uss position pe rkh do.

∴ In one line : Har step per minimum element dhoondo aur usse sahi jagah per fix kr do.

Time Complexity:-
↳ O(N²) for best worst & average case.

Bubble Sort

- Bigger element at last index.
- adjacent element ko compare karta hai aur agar wo galat order mein hote hai to swap kr deta hai

⇒ Dry run:

arr[] = {5, 3, 2, 4}
↑ Big

S-1: [5, 3], 2, 4
S-2: 3, [5, 2], 4
S-3: 3, 2, [5, 4]
S-4: [3, 2, 4, 5] ⇒ 5 is at Right Place

⇒ Logic:

- 2 loop hote hai
  outer loop : (n-1) times (kitne passes)
  inner loop : (n-i-1) times (compare & swap)
- Har pass ke baad ek bada element right end per fix ho jata hai

Pseudo code:

```
BubbleSort(arr, n):
for i from n-1 to 0;
  for j from 0 to i-1;
    if arr[j] > arr[j+1];
      swap arr[j] & arr[j+1];
```

⇒ Time Complexity : O(N²) → worst case

Optimized Approach

```
optimized Bubble Sort [arr, n];
for i from n-1 loo;
swapped ← false
for j from 0 to i-1;
    if (arr[j] > arr[j+1];
        swap arr[j] & arr[j+1]
        swapped ← true;
if swapped = false:
    break;
```

∴ TC : O(N) (Best case)

Insertion-Sort

- Takes an element & places it in correct order.

Ex: [14, 9], 15, 12, 6, 8, 13
[9, 14, 15], 12, 6, 8, 13
[9, 14, 15, 12], 6, 8, 13
9, 12, 14, 15 ----

Concept summary:
- At every index i, place arr[i] in its correct position by swapping it backward until the left element is not greater.
- The while loop help in Pushing smaller element to the front.

Pseudo code:

```
for i from 0 to n-1
  j=i // start from current element.
while (j>0 && arr[j-1]>a[j]) // Push current element to its right spot
  swap (arr[j-1]>a[j] // if left is greater than current
  j--
```

⇒ Dry run example:

arr = [5, 3, 4, 1]
i = 0 1 2 3

For i = 0
j = i = 0
j > 0 → false
Nothing happen
Arr = [5, 3, 4, 1]

For i = 1
j = 1
arr[0] > arr[1] → 5 > 3 → ✓
∴ swap 5 & 3 ⇒ [3, 5, 4, 1]
j = 0 → loop end.
Array : [3, 5, 4, 1]

Pass 3: i = 2
j = 2
arr[1] > arr[2] ⇒ 5 > 4 ✓
swap 5 & 4 → [3, 4, 5, 1]
j = 1
arr[0] > arr[1] → 3 > 4 → ✗
stop here
Arr : [3, 4, 5, 1]

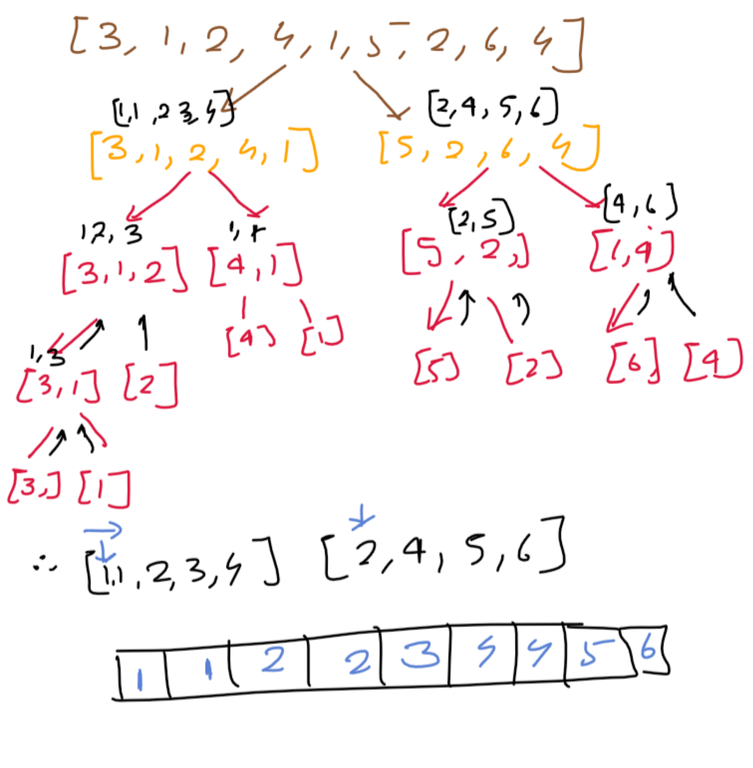
For i = 3:
j = 3
j-- (5 > 1 → ✓ swap → [3, 4, 1, 5]
j = 2
j-- (4 > 1 → ✓ swap → [3, 1, 4, 5]
j = 1
j-- (3 > 1 → swap → [1, 3, 4, 5]
j = 0 → stop
Array : [1, 3, 4, 5] ⇒ Sorted.

Time Complexity :-

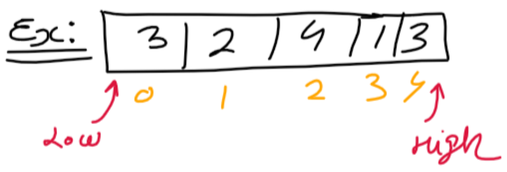
worst case : O(n²)
Best case : O(n) [sorted order]

Merge Sort :- ⇒ T.C = O(n log n), S.C = O(n)

↳ Divide & Merge.



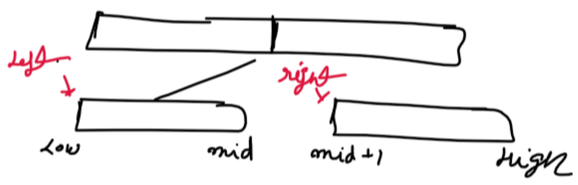
Ex:



Pseudo Code:

```
MergeSort(arr, low, high)
{  if (low >= high) return;
   mid = (low + high)/2      // find where to split
   mergeSort(arr, low, mid)      } subproblem
   mergeSort(arr, mid+1, high)   }
   merge(arr, low, mid, high)  // combine.
}
```

Pseudo code of Merge operation




```
Merge(arr, low, mid, high)

  temp → [ ]  ⇒ auxiliary global array

   left = low
   right = mid+1
while (left <= midd && right <= high)
{
  if arr[left] <= arr[right]
     temp.add(arr[left])
        left++;
  else
     tem add [arr[right])
        right++;
  }
```

If on left:

```
while (left <= mid) {
   temp add (arr[lft];
      left++
  }
```

If on right:

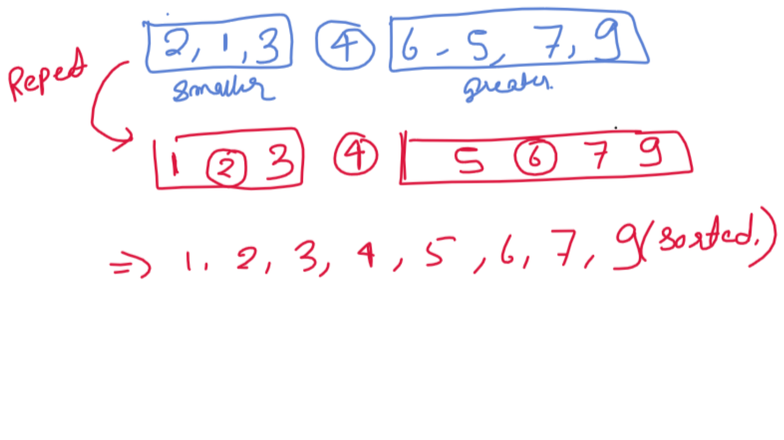
```
while (right <= high)
{  temp.add (arr [right])
      right++
  }
for (i = low → high)
   {  arr[i] = temp(i - low);
   }
```

} moving sorted element from temp[] to original array

Quick Sort :-

- Quick Sort is a divide & conquer algorithm like merge Sort.
- It does not use any extra array for sorting (though it uses an auxiliary stack space.)
- Quick sort is slightly better than Merge Sort.

Ex: 6, 2, 5, 7, 9, 1, 3

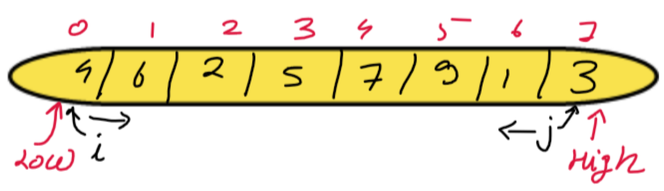


⇒ 1, 2, 3, 4, 5, 6, 7, 9 (sorted.)

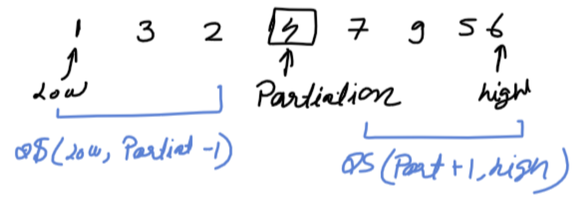
Two simple step to follow

(1) Pick a Pivot and Place it in its correct Place in the sorted array (generally first element)

(2) Shift smaller element on the left of the Pivot and larger one to the right.



Pivot = a[low]



Pseudo Code:

```
QS (arr, low, high){
  if (low < high){
Partition = f(arr, low, high)
     QS (arr, low, Partition -1);
     QS (arr, Partition+1, high);
   }
}
```

```
Partition f(arr, low, high){
   Pivot = arr[low]
     i = low;
     j = high;
while (i<j)
 {
  while (arr[i]<=arr[Pivot] && i<=high){
          i++;
    }
  while (arr[i] > arr[Pivot] && j >= low){
          j--;
    if (i<j) swap (arr[i], arr[j]
  }
 swap(arr[low], arr[j])
  return j;
}
```

#Time complexity : O(n log n)

space " : O(

Arrays

#Basic of Array:-

- Data structure, contains similar element.
- max length of Array

int arr[10^6] => inside main()

int array[10^7] => Globally.

accessing element

```
for( i=0; i<n; i++)
    arr[i];
}
```

Q(1) Largest element in an array.

arr[ ] = {3, 2, 1, (5), 2}

Largest

⇒ Brute force Approach

If we sort the array

∴ Largest = arr[n-1]

T.C = O(NlogN)

S.C = O(1)

#Optimal Solution:-

arr[] = {3, 2, 1, 5, 2}

Largest

(1) Take first element is largest

(2) Compare the element from 1st element, if a[0] < a[i] then largest is a[i]

⇒ My approach:-

(1) Largest = a[0]

(2) Loop from i=0 to i<n

```
if a[0] < a[i]{
    largest = a[i]
}
Print a[i]
```

∴ TC = O(N)

Dry Run:-

arr[ ] = {3, 2, 1, 5, 2}

Largest = a[0] = 3

For i=1 ⇒ 3<2 → X

For i=2 ⇒ 3<1 → X

For i=3 ⇒ 3<5 → ✓

Largest = 5

Q(2) Second Largest Element in array

arr[ ] = {1, 2, 4, (7), 7, (5)}

1st Large

2nd Largest.

Brute force:=>

(1) Sort the array

(2) arr[n-2] = 2nd Largest array

(3) T.C = O(N logN)

(4) Start from Back, consider last element is largest

```
largest = arr[n-1]
for(i=n-2; i>=0; i--)
    if (arr[i] = largest){
        second largest = arr[i];
        break;
    }
}
}
```

∴ T.C = O(n log n) + O(n)

#Better approach:-

arr[ ] = {1, 2, 4, 7, 7, 5}

```
first Pass
Largest:
Largest = X X X 7
Largest = arr[0]
for i<n.
  if (arr[i] > largest)
      largest = arr[i];
}
```

```
2nd Largest = X X X 4 5
2nd Larg = -1
for i=0<n;
  if (arr[i] > 2nd Largest &&
      arr[i] != largest)
      2nd largest = arr[i]
}
Print (2nd Larg
```

∴ T.C = O(N) + O(N)
= O(2N)

#Optimal Approach:- ⇒ T.C = O(N)

current

arr[ ] = {1, 2, 4, 7, 7, 5}

largest = arr[0], 2nd Largest = INT-Min / -1

Dry Run:-

Largest = a[0] 2 4 7

2nd Largest = X X 4 5

Pseudo code:- 2nd Largest

(1) if a[i] > largest, then update 2nd largest & largest

(2) Else a[i] > 2nd largest, update the 2nd largest.

- Print 2nd Largest.

⇒

```
Large = Int_Min
second_Large = Int_Min
for (i=0; i<n; i++)
  if (arr[i] > Large){
    second_Large = Large;
    Large = arr[i];
  }
else if (arr[i] > second_Large && arr[i] != Large)
    second_Large = arr[i];
}
return second_Larg;
```

Pseudo code of 2nd smallest:

small = INT_Max, 2nd small = Int_Max

Q3 Check if the array is sorted

Ex: a[] = {1, 2, 2, 3, 3, 4} ⇒ Sorted

a[] = {1, 2, 1, 3, 4} ⇒ Not Sorted.

My approach

a[] = {1, 2, 2, 5, 3, 4}

a[0] < a[1] < a[2] < a[3] .. a[n]

```
for(i=1; i<n; i++){
  if(a[i] < a[i+1]){
     array is sorted
  else, not sorted.
  return true;
```

Q4 Remove duplicates in place from sorted array.

arr[] = {1,1,2,2,2,3,3}

= {1, 2, 3, ------ }

unique elements.

return no. of unique elements

#Brute force:

- we use a Hash set.

As we know Hashset only store unique element.

Pseudo code:

- Declare a Hash set
- Run a for loop from starting to the end
- Put every element of the array in the set.
- Store size of set in a variable k.
- Now Put all the element of the set in the array from the starting of the array.
- Return k.

T.C = O(n log(n)) + O(n)

↑ Insertion in set take

S.C = O(N)

#Optimal Approach (Two Pointers)

Pseudo code:

(1) Take a variable i as 0;

(2) use a Loop by using a variable 'j' from 1 to length of the array

(3) If arr[j] != arr[i], increase 'i' and update arr[i] = = a[j]

(4) After completion of the loop returns i+1 ; size of the array of unique elements.

Dry Run:

i=0

arr[] = {1, 1, 2, 2, 2, 3, 3}

j

arr[i] != arr[j]

arr = [1, 1, 2, 2, 2, 3, 3]

i j

⇓ update and move

arr = [1, 2, 2, 2, 3, 3]

i (1)

⇓

arr[i] != arr[j] (update & move)

arr = [1, 2, 3, 2, 3, 3]

0 1 2 3 4 5

i (2)

return = i+1 ; // 2+1 = 3

∴ T.C = O(N)

S.C = O(1)

Q(5) Left Rotate an array by one Place

a[] = [1, 2, 3, 4, 5]

⇓

[2, 3, 4, 5, 1]

Left Rotate an array by one Place = First element is Present at the last index.

Optimal solution:

arr[] = [1, 2, 3, 4, 5]

i=0 ; n-1, i++

temp = arr[0]

Pseudo code:

- store 0th index element in variable

  temp = arr[0]

- Iterate the array from 0th index to n-1 index.
- store the value Present in the next index to the current index
- Put the variable temp in the last index of the array.

```
for (int i=0; i<n-1; i++){
    arr[i] = arr[i+1];
}
arr[n-1] = temp;
```

- Now Print the array, it will be left rotated by 1 Place.

T.C: O(n), as we iterate through the array only once.

S.C: O(1) as no extra space is used.

Q(6) Left rotates the array by D Places

arr[] = [1, 2, 3, 4, 5, 6, 7] d=2

(0 1 2 3 4 5 6)

[3, 4, 5, 6, 7, 1, 2]

arr[] = [1, 2, 3, 4, 5, 6, 7] d=7

for d = 8 — (7+1) rotation

d = 9 — (7+2) rotation

d = 15 — (7+7+1) rotation

we can say that $d = d \% N$ rotation, for d > size of Array.

Brute force:

- using temporary array

(1) Copy the first k element into the temp array.

temp[] = [1, 2, 3] k=3

```
for(i=0; i<k; i++)
    temp[i] = arr[i]
```

(2) Shift n-k element from the last by k position to the left

```
for(i=0; i<n-k; i++)
    arr[i] = arr[i+k];
```

(3) copy the element into the main array from the temp array.

```
for(int i=n-k; i<n; i++){
    arr[i] = temp[i-(n-k)]
```

Dry run:

arr[] = [1, 2, 3, 4, 5, 6, 7], k=3

temp[] = [1, 2, 3]

arr[] = [1, 2, 3, 4, 5, 6, 7] n-k = 7-3 = 4 to

= [4, 5, 6, 7, . . .]

copy temp[] to arr[]

arr[] = [4, 5, 6, 7, 1, 2, 3]

T.C = O(N)

S.C = O(k). since k array element need to be stored in temp array.

#Optimal Solution:- (Reversal Algorithm):-

a[] = {1, 2, 3, 4, 5, 6, 7} k=3

Rever. → 3, 2, 1, 7, 6, 5, 4

Reverse → 4, 5, 6, 7, 1, 2, 3 = Rotated arr.

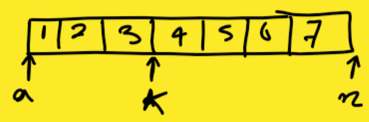
Pseudo code:

S-1: Reverse the first k element of the array

S-2: Reverse the last n-k element of the array.

S-3: Reverse the whole array

mem:



reverse(a, a+k)

reverse(a+k, a+n)

reverse(a, a+n)

T.C = O(N), n = no. of element in array

S.C = O(1), since, no extra space is required

Q7 Move all the Zero to the end of array.

arr[] = {1,0,2,3,2,0,0,4,5,1}
= {1,2,3,2,4,5,1,0,0,0}

⇒ Brute Force:

- use of temp array.
- store non-zero element from original array to temp array.
- copy the element from temp array one by one and fill the first xPlace of the original array.
- The last remaining Place of the original array will be filled with zero.

arr[] = {1,0,2,3,2,0,0,4,5,1}
(store) temp[] = {1,2,3,2,4,5,1}
(copy) arr[] = {1,2,3,2,4,5,1,0,0,0}

Pseudo code:

```
temp=[]
S-1: for (i=0 → n)
     { if a[i]!=0,
         temp.add(arr[i]);
     }
     nz = temp.size()        // no. of non-zero number
S-2: for(i=0; i<temp.size(); i++)
     {
         arr[i] = temp[i]
     }
S-3: for(i=temp.size(); i<n; i++)
     {
         a[i]=0;
     }
```

T.C = O(N)+O(n)+O(n-x) = O(2N)
S.C = O(x) → O(N)

|| Optimal Approach:

(1) My first thought Process is to sort the arry and then reverse it, But we can't do this, Because, sorting or reversing will not maintain the relative order of non-zero element.

(2) We can use two-Pointer approach.

Optimal approach (Two Pointer)

arr[] = {1,0,2,3,2,0,0,4,5,1}
first 0th element ← j, i

if a[i]=0, a[i]!=0, then
swap(a[j], a[i]), j++

Pseudo code:

```
S-1: j=-1
     for(i=0; i<n; i++)
       if(arr[i]==0){
         j=i;
         break;
       }
     }
S-2: for(i=j+1; i<n; i++)
     {
       if(arr[i]!=0){
         swap(arr[i], arr[j]);
         j++;
       }
     }
```

T.C → O(N)
S.C → O(1)

Q(8) Linear Search

arr[] = [6,7,8,4,1] num = 4

Approach:

- we will traverse the whole array and see if the element is Present in the array or not
- if found we will Print the index of the element
- otherwise, we will Print -1

Pseudo code:

```
for(i=0; i<n; i++)
  if(arr[i]==num){
    return i;
  }
return -1;
```

Q(9) Union of two sorted array

arr1[] = {1, 1, 2, 3, 4, 5}
arr2[] = {2, 3, 4, 4, 5, 6}
union[] = {1, 2, 3, 4, 5, 6}

Brute force :- using map

```
set <int> st
for (i=0; i<n1; i++)
    st.insert(a1[i])

for (i=0 → n2)
    st.insert(a2[i])

union[st.size()]
i=0
for (auto it : st)
    union[i++] = it;
```

T.C = O(n1+n2)
S.C = O(n1+n2)

#Optimal approach (2-Pointer)

arr1[] = {1, 1, 2, 3, 4, 5}
arr2[] = {2, 3, 4, 4, 5, 6}

Approach:-

- Pointer i, j pointing 0th index of arr1 & arr2
- union[] vector to store union arr1 & arr2
- Traverse through arrays

case-1: arr1[i] == arr2[j]
insert arr[i];
i++

case-2: arr1[i] < arr2[j]
insert arr2[i]
i++

case-3: arr1[i] > arr2[j]
insert arr2[j]
j++;

T.C = O(n1+n2)
S.C => O(n1+n2)

Q(9.b) Intersection of two sorted array:-

A[] = {1, 2, 2, 3, 3, 4, 5, 6} → n1
B[] = {2, 3, 3, 5, 6, 6, 7} → n2

```
vis[] = {0, 0, 0, 0, 0, 0}
ans → [];
for(i=0 → n1){
  for(j=0 → n2){
    if (a[i]==b[j] && vis[j]==0)
      { ans.add(a[i])
        vis[j] = 1;
        break;
      }
    if(b[j]>a[i]) break;
  }
}
```

T.C = O(n1 x n2) = O
S.C = O(n2)

#Dry Run:-

arr1[] = {1, 2, 4, 5}
arr2[] = {2, 3, 4}

i	arr1[i]	j	arr2[j]	compare	Action	Result
0	1	0	2	1!=2		
0	1	1	3	1!=3		
0	1	2	4	1!=4	i done	[]
1	2	0	2	2==2	Push 2 break loop	[2]
2	4	0	2	4!=2		[2]
2	4	1	3	4!=3		[2]
2	4	2	4	4==4	Push 4 break loop	[2,4]
3	5	6	2	5!=2		
3	5	1	3	5!=3		
3	5	2	4	5!=4	i done	[2,4]

Optimal approach (Two Pointer)

arr1[] = {1, 2, 3, 3, 3, 4, 5, 6}
arr2[] = {2, 3, 3, 5, 6, 6, 7}

Pseudo code:-

```
i = 0
j = 0
result ← empty vector
while i < length(arr1) && j < length(arr2):
    if arr1[i] == arr2[j]:
        add arr[i] to result
        i++
        j++
    else if arr1[i] < arr2[j]
        i++
    else
        j++
return result.
```

T.C = O(n+m)
S.C = O(min(n, m))

Dry Run:-

arr1 = {1, 2, 4, 5}
arr2 = {2, 3, 4}

1st iteration (i=0, j=0);
arr1[i]=1, arr2[j]=2
1<2 → i++

2nd iteration (i=1, j=0)
- arr1[i]=2, arr2[j]=2
- 2==2 → result=[2]
- move both i++, j++

3rd iteration (i=2, j=1);
- arr1[i]=4, arr2[j]=3
- 4>3 → move Pointer j, j++

4th iteration (i=2, j=2);
- arr1[i]=4, arr2[j]=4
- 4==4 → result=[2,4]
- Move both i, j ⇒ i++, j++

End of Loop:
- i=3, j=3 ⇒ both Pointer have reached the end of their respective array.
- The intersection is [2,4]

Q10 Finding the missing Number in an array.

arr[] = {1, 2, 4, 5} N = 5

ans = 3,

Brute Force:

- checking each element in array using linear search.

⇒ Pseudo code:

```
for i from 1 to N:
    set flag = 0
    for j from 0 to n-1
        if a[j] == i:
            set flag = 1 // Element Present
            Break
    if flag == 0; // if it doesn't exist.
        return i
Return -1
```

T.C = $O(N^2)$
S.C = O(1)

Better Solution

- using hashing

arr[] = {1, 2, 4, 5} N = 5

hash = [0,0,0,0,0,0] // index from 0 to N(N+1) size

hash = | 0 | ~~1~~ | ~~2~~ | 0 | ~~3~~ | ~~4~~ | ~~5~~ |
0 1 2 3 4 5

has[3] = 0 ∴ Missing Number is 3

Pseudo code:

```
hash [n+1] = {0};
for i = 0, to n{
    hash[arr[i]] = 1; // number i Present in array
}
for (i = 1 -> n-1){
    if (has[i] == 0) // number i is missing from array
        return i
}
```

T.C = O(N) + O(N)
S.C = O(N), hash array.

Optimal solution - I:

- summation approach

arr[] = [1, 2, 4, 5] N = 5

∴ Sum Total = $\frac{N \times (N+1)}{2} = \frac{5 \times 6}{2} = 15$

Sum 2 = 1+2+4+5 = 12 (Sum of each value in array.

missing = Sum - Sum 2
= 15 - 12
= 3 is missing

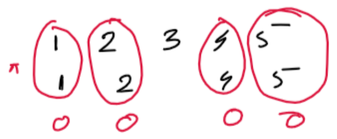
Pseudo code:

```
sum = n x (n+1) / 2
S2 = 0
for (i = 0 -> n-1) { sum of all Array elements.
    S2 += arr[i] }
missing = Sum - S2
return missing
```

T.C = O(N)
S.C = O(1)

Optimal solution: 2

using XOR (^) ⇒ a^a = 0, a^0 = a



XOR1 = 1^2^3^4^5 (Numbers from 1 to N)
XOR2 = iterate through array
= 1^2^4^5

XOR1 ^ XOR2
(1^1) ^ (2^2) ^ (3) ^ (4^4) ^ (5^5)
0^0^3^0^0
0^3^0 = 3

Pseudo code:

```
XOR1 = 0
XOR2 = 0
for (i = 0, n-1){
    XOR2 = XOR2 ^ a[i] // XOR of array elements
    XOR1 = XOR1 ^ (i+1) // XOR up to [1 -> N-1]
}
XOR1 = XOR1 ^ N; // XOR up to [1-N]
missing = XOR1 ^ XOR2 // missing number
return missing.
```

T.C = O(N)
S.C = O(1)

NOTE: Among the optimal approaches, the XOR approach is slightly better than the summation one because term (N x (N+1))/2, cannot be stored in an integer if value of N is big like (10^5). In that that case, we have to use some bigger data types. But we will face no issue like while using XOR approach.

Q(11) Maximum consecutive one's in the array

arr[] = {1, 1, 0, 1, 1, 1, 0, 1, 1}
(2) (3) (2)

ans = 3

approach:

arr[] = {1, 1, 0, 1, 1, 1, 0, 1, 1}

count = ~~0~~ 1 2
0 1 ~~2~~ ~~3~~
0 1 2

max = ~~0~~ 1 ~~2~~ 3

∴ T.C = O(N)

T.C = O(N)

Pseudo code:

```
count = 0
max = 0
for each element in array
    if element is 1
        increase count by 1
    else
        set count to 0
    max = maximum (max, count)
return max
```

Q12 Find the number that appear one, and the other number twice.

a[] = {1, 1, 2, 3, 3, 4, 4}

every no. will appear twice, but one number will appear once, so find that number

ans = 2, appear once.

Approach comes in my mid
(1) linear search
(2) hashing
(3) XOR

Brute Force :-

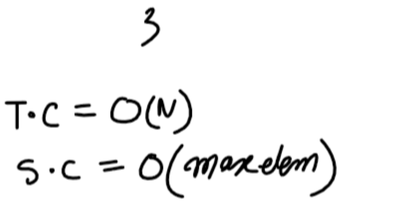
Doing Linear search :-

```
for(i=0 ton){
   num =a[i]
    count=0
  for(j=0 → n)
    if (arr[j]==num)
      count++;
  }
  if(count==1) return num;
}
```

T.C = O(N²)
S.C = O(1)

Better Solution :1:

Using array hashing.



max element = 4
∴ hash[] size = maxelem + 1

```
maxelem = arr[0]
for (i=0; i<n; i++)
  maxelem = max(maxelem, arr[i])

has[maxelem] ={0}
for(i=0, to n)
   has(arr[i])++;

for (i=0 to n){
  if (has(arr[i])==1)
     return arr[i];
  }
```

T.C = O(N)
S.C = O(maxelem)

Better solution :-2

Using map hashing-

```
for(i=0 → n)
  map(arr[i])++

for(auto it: map){
  if (it.second==1)
    return it.first
}
```

T.C = O(N log M) + O(n/2 + 1)
S.C = O(n/2 + 1)

Optimal solution (XOR)

```
XOR=0
for (i=0 → n)
   XOR = XOR ^ arr[i];
  return XOR;
```

T.C = O(N)
S.C = O(1)

Q.13 Longest subarray with given sum k (Positives)

arr[] = {1, 2, 3, 1, 1, 1, 1, 4, 2, 3} k = 3

Subarray => Contiguous Part of the array.

Brute Force :

Generate all sub-array

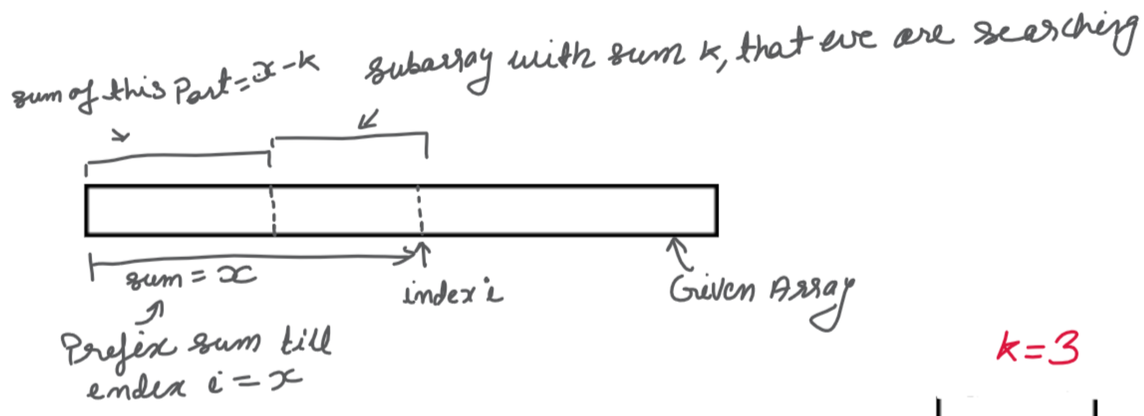
```
length = 0
for(i=0; i<n; i++){
   for (j=i ; j<n ; j++){
      for (k=i, k<j, k++)
         sum += a[k]
      if (sum == k) length = max(length, j-i+1)
      3
      3
      Print length;
```

T.C ≈ $O(n^3)$

S.C = $O(1)$

Better solution :

Using hashing (concept of Prefix sum)

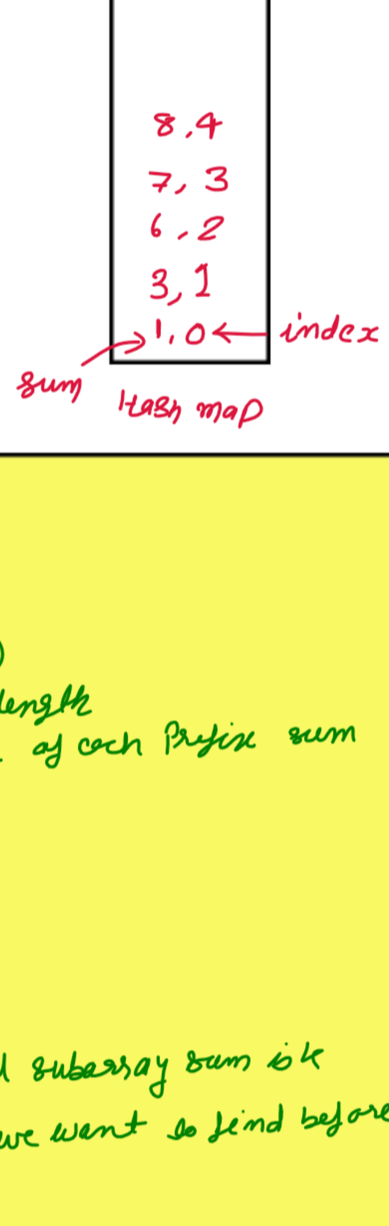


Dry Run :

arr[] = {1, 2, 3, 1, 1, 1, 1, 4, 2, 3}
(indices 0 1 2 3 4 5 6 7 8 9)

sum = ~~0~~ 1 3 6 ~~7~~ 9

Length = ~~0~~ 2 3



Optimal solution :

using two Pointer approach

arr[] = {1, 2, 3, 1, 1, 1, 1, 3, 3} k = 6

i, j

sum = ~~x~~ ~~3~~ ~~6~~ ~~7~~ 6

Length = ~~0~~ 3

Pseudo code :

Pseudo code :

(1) initially
```
left = 0
right = 0
sum = arr[0]
max len = 0
```

(2) while (right < n)

(a) if (sum > k)
- shrink window from left:

```
while (sum > k && left ≤ right)
    sum -= arr[left]
    left ++;
```

(b) if sum == k;
```
max len = max(max len, right - left + 1);
```

(c) move right forward by 1 (right ++)
if right is still inside array (if (right < n)
add array[right) to sum
```
sum += arr[right]
```

(3) return max len

T.C = O(2N)
S.C = O(1)

Pseudo code :

- initialize :
```
sum = 0          // running total (Prefix sum)
maxlen = 0       // store maximum subarray length
map = empty      // store first occurrence of each Prefix sum
```
- loop through the array using index i from 0 to arr.length - 1:

Add arr[i] to sum (sum += a[i]

```
if (sum == k)
    maxlen = i+1   // from index 0 to i, full subarray sum is k
remaining = sum - k   // this is the Part we went to find before current i

if map contains 'remaining':
    subarray-length = i - map[remaining]
    update maxlen = maximum of (maxlen, subarray-length)

if sum is not already in map:
    map[sum] = i    // store the first time this sum appear

return maxlen
```

T.C = O(n log n) using map, $O(n^2)$ if using unordered map

S.C = O(n)

NOTE : If array contains Positive and negative, then optimal solution is using hashing.