→ Recursion :

- **factorial of n :**

$$n! = n * (n-1)!$$

↓

```
fact(n) = n * fact(n-1);
```

```
int factorial(int n)
{
    if (n == 0)  // Base condition
        return 1;

    return n * factorial(n-1);
}
```

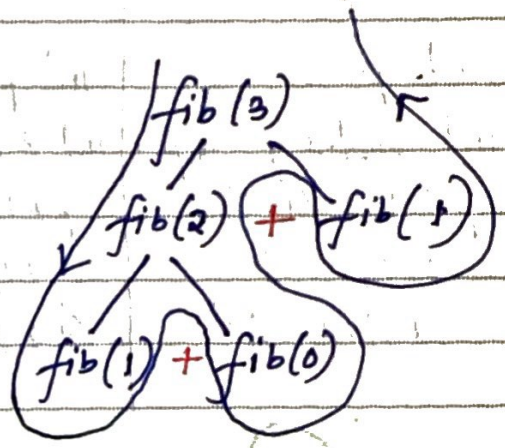
Alternative:

```
int f = factorial(n-1)
return n * f;
```

- **fibonacci no. :** [to find n^{th} fibonacci no.]

0 1 1 2 3 5 8 13 .....
0th 1st 2nd 3rd 4th 5th 6th 7th

```
int fib(int n)
{
    if(n==0 || n==1)
        return n;
    int f1 = fib(n-1);
    int f2 = fib(n-2);
    return f1+f2;
}
```



∴ for n=3 i.e. 3rd fibonacci no.
O/P = 2

- **pow(x, n) → [x^n] :**

```
int power(int x, int n)
{
    if(n==0)
        return 1; //for x^0 = 1
    int pow = x * power(x, n-1);
    return pow;
}
```

- **To count no. of digits :**

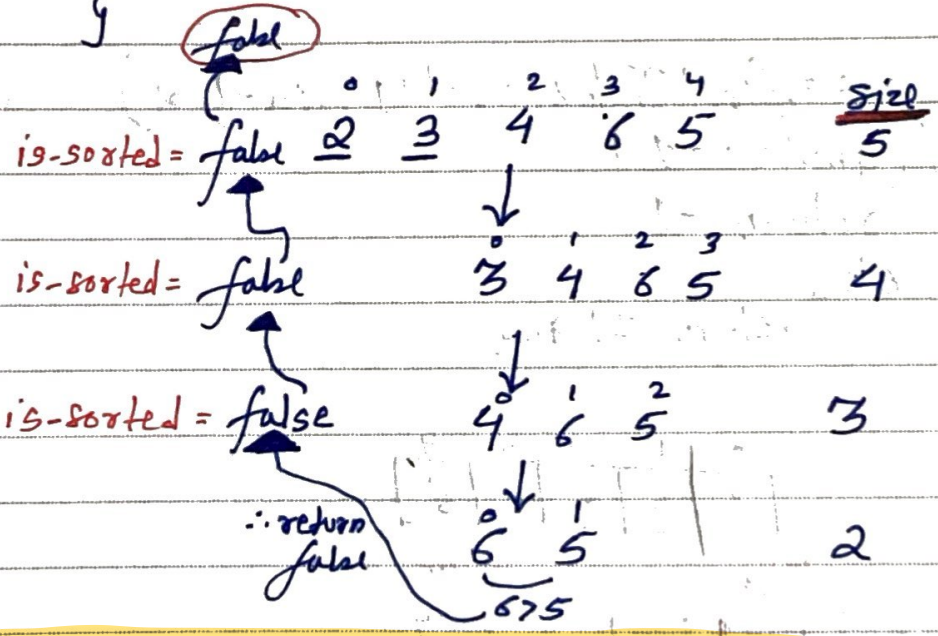
```
int count(int n)
{
    if(n<10)
        return 1;
    int d = 1 + count(n/10);
    return d;
}
```

- To check if array is sorted using recursion:
  ↳ in ↑ order

```
bool sorted (int input[], int size)
{
    if (size == 0 || size == 1)   // if size of array is 0 or 1, then it's sorted.
        return true;

    if (input[0] > input[1])
        return false;

    bool is_sorted = sorted (input+1, size-1);
    return is_sorted;
}
```



- find ~~sum~~ sum of all elements of array of size = n:

```
int sum_Array (int input[], int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return input[0];

    return input[0] + sum_Array (input+1, n-1)
}
```

- To return first index of a given no. (x) in an array:

Ex

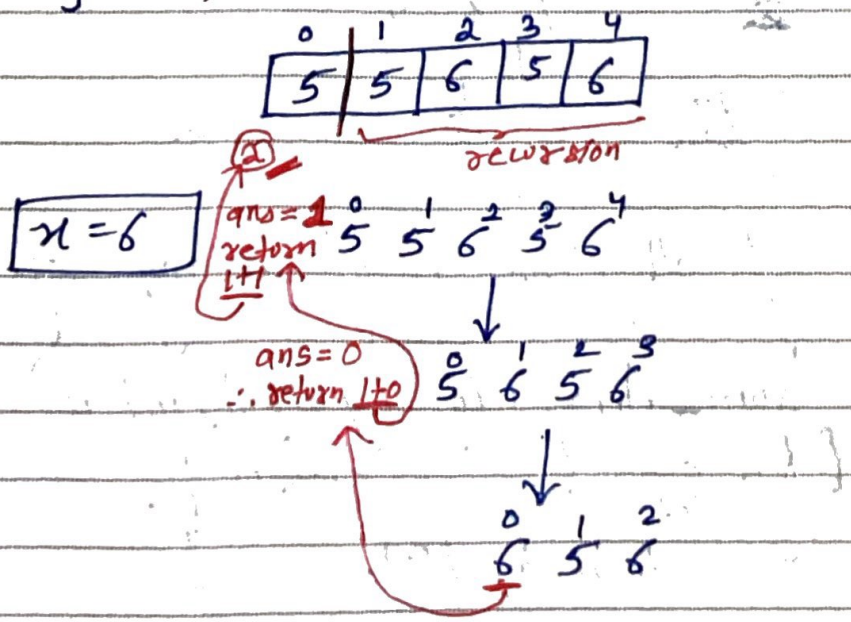
input[] = | 5 | 5 | 6 | 5 | 6 |
0 1 2 3 4

size	x	output
5	5	0
5	6	2
5	10	-1

```
int firstIndex (int input[], int size, int x)
{
    if (size == 0)
        return -1;
    if (input[0] == x)
        return 0;

    int ans = firstIndex (input+1, size-1, x)
    if (ans == -1)
        return -1;
    else
        return 1+ans;
}
```

Working



- To return last index of a given no. (x) in an array.

	0	1	2	3	4
input[]	5	5	6	5	6

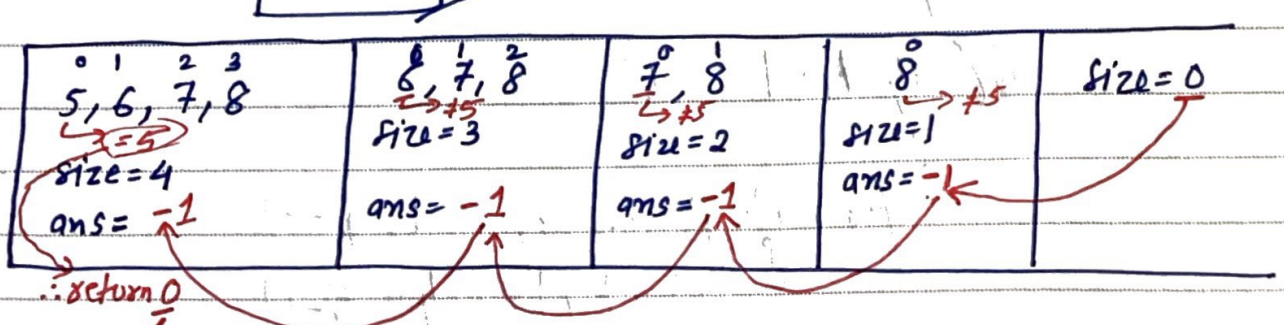
size	x	output
5	5	3
5	6	4
5	10	-1

M1

```
int lastIndex(int input[], int size, int x)
{
    if(size==0)
        return -1;
    int ans = lastIndex(input+1, size-1, x);
    if(ans != -1)
        return 1+ans;
    else if(input[0] == x)
        return 0;
    else
        return -1;
}
```

Working

x=5

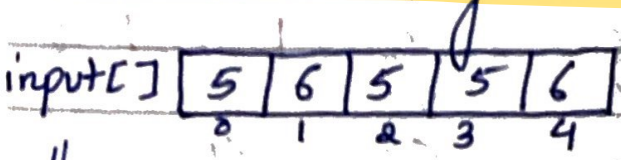


o/p = 0

M2

```
int lastIndex(int input[], int size, int x)
{
    if(size == 0)
        return -1;
    if(input[size-1] == x)
        return size-1;
    return lastIndex(input, size-1, x);
}
```

All Indices of a number:



size	x	output[]
5	5	0 2 3 (0 1 2)
5	6	1 4 (0 1)
5	1	return 0

it is sorted order

at 0,2,3 in input[], 5 is present. & output[] size is 3

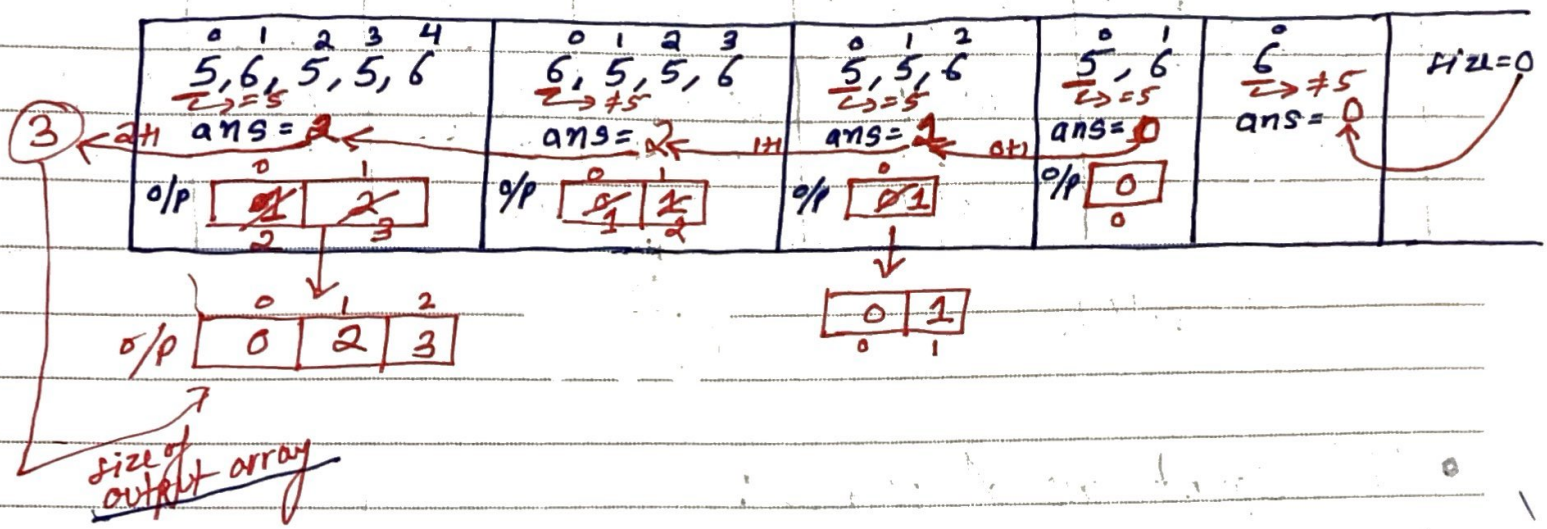
& size of output[]=2

Method 1:

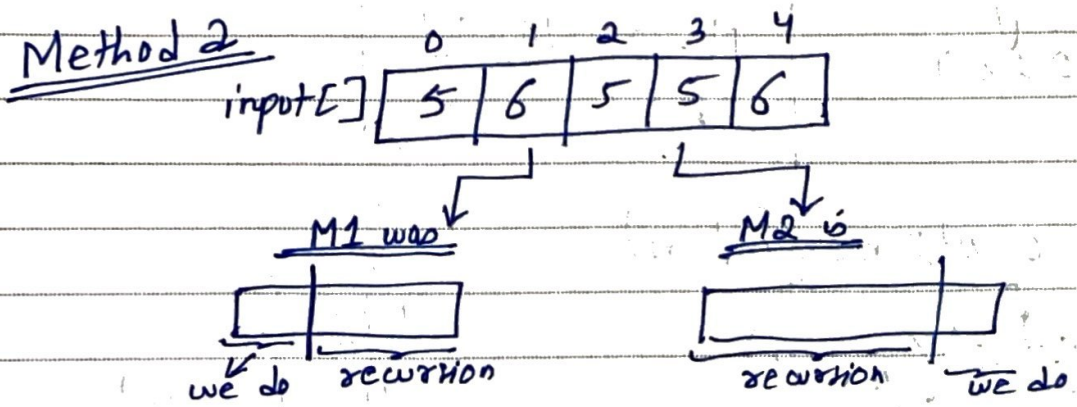
```
int allIndexes(int input[], int size, int x, int output[])
{
    if(size==0)
        return 0

    int ans = allIndexes(input+1, size-1, x, output);
    for(int i=0; i<ans; i++)
    {
        ++output[i];
    }
    if( input[0]==x)
    {
        for(int j=ans ; j>=1; j--)
        {
            output[j] = output[j-1];
        }
        output[0]=0;
        return 1+ans;
    }
    else
        return ans;
}
```

x = 5



Method 2




```
int AllIndexes (int input[], int size, int x, int output[])
{
    if (size == 0)
        return 0;
    int ans = AllIndexes (input, size-1, x, output);
    if (input[size-1] == x)
    {
        output[ans] = size-1;
        ans = ans+1;
    }
    return ans;
}
```

n=5

0 1 2 3 4 5, 6, 5, 5, 6 size=5 ans=3 ← 2+1 o/p 0 2 3	0 1 2 3 5, 6, 5, 5 5=5 size=4 ans=2 ← 1+1 o/p 0 2 3	0 1 2 5, 6, 5 5=5 size=3 ans=1 ← 0+1 o/p 0 2	0 1 5, 6 5≠5 size=2 ans=1 ← 0+1 o/p 0	0 5 5=5 size=1 ans=0 o/p 0

size=0

returned ③ → size of output[].

- To count no of zeroes:

N=0
o/p=1

N=10320
o/p=2

N=00010204
o/p=2

Since it is stored in (n) → type int
∴ it is stored as n=10204

```
int countZeroes(int n)
{
    if(n<=9)
    {
        if(n==0)
            return 1;
        else
            return 0;
    }
    int d = n%10;
    if(d==0)
        return 1 + countZeroes(n/10);
    else
        return countZeroes(n/10);
}
```

i.e. 0 + countZeroes(n/10);

Check Palindrome :

a ✓ — a b c b a ✓ — a b c ✗ — '\0' → null string ✓

```
bool Helper(char input[], int start, int end)
{
    if(input[0] == '\0' || input[1] == '\0')
        return true;
```

(input[0] == '\0' : zero length string; input[1] == '\0' : length = 1)

```
    if(start >= end)
        return true;
    if(input[start] == input[end])
    {
        return helper(input, start+1, end-1);
    }
    else
        return false;
}

bool isPalindrome(char c[])
{
    int i=0;
    for(i=0; c[i]!='\0'; i++); // length of string
    bool ans = helper(c, 0, i-1);
    return ans;
}
```

- Given a string, compute recursively a new string where all appearances of "pi" have been replaced by "3.14"

Working:

1) x | pi x pi (0 1 2 3 4 5)

recursion ↓

3.14 x 3.14

x — see if it's forming "pi" then put 3.14 else just chordo.

2) p i x p i (0 1 2 3 4)

i x 3.14

p — forming "pi" ∴ replace with 3.14

we get, 3.14 x 3.14 (0 1 2 3 4 5 6 7)

3) **Note:** pi → 2 characters; 3.14 → 4 characters] 2 extra characters needed

p | i p p p p p i i i i p i (0 1 2 3 4 5 6 7 8 9 10 11)

recursion ↓

p i p p p 3.14 i i i 3.14 (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

Small work

p — replace with 3.14 → ∴ shift left by 2 spaces ahead and replace with 3.14

3.14 p p p 3.14 i i i 3.14 (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)

index 0 = '3'
" 1 = '.'
" 2 = '1'
" 3 = '4'

```
#include <cstring>

void replaceHelper(char str[], int start)
{
  if (str[start] == '\0')
    return;
  replaceHelper(str, start+1);
  if (str[start] == 'p' && str[start+1] == 'i')
  {
    for (int i = strlen(str); i >= start+2; i--)
    {
      str[i+2] = str[i];
    }
    str[start] = '3';
    str[start+1] = '.';
    str[start+2] = '1';
    str[start+3] = '4';
  }
}

void replacePi(char input[])
{
  replaceHelper(input, 0);
}
```

(or start)

Remove 'x':

1) i/p = xaxb
   o/p = ab

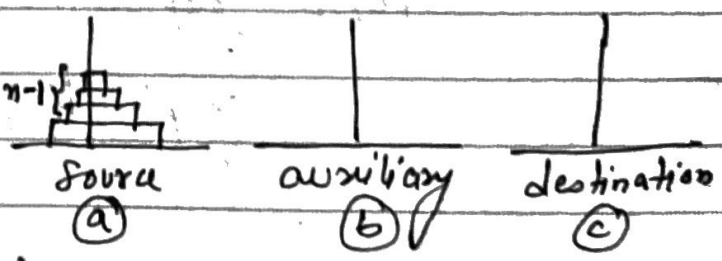
2) i/p = abc
   o/p = abc

3) i/p = xx
   o/p = ... → null string (\0)

```
#include<cstring>
void removeX(char input[])
{
    if (input[0] == '\0')
        return;
    removeX(input + 1);
    if(input[0] == 'x')
    {
        for(int i=0; i<strlen(input); i++)
        {
            input[i] = input[i+1];
        }
    }
}
```

Towers of Hanoi:

Say, Source rod : 'a'
Auxiliary rod : 'b'
Destination rod : 'c'



1) i/p = 2 (n)
   o/p = a b
         a c
         b c

→ $2^2 - 1$ moves

2) i/p = 3 (n)
   o/p = a c
         a b
         c b
         a c
         b a
         b c
         a c

→ $2^3 - 1$ moves

```
void TOH(int n, char source, char auxiliary, char destination)
{
  if(n==0)
    return;
  TOH(n-1, source, destination, auxiliary);
  cout<<source<<" "<<destination<<endl;
  TOH(n-1, auxiliary, source, destination);
}
```

See the order

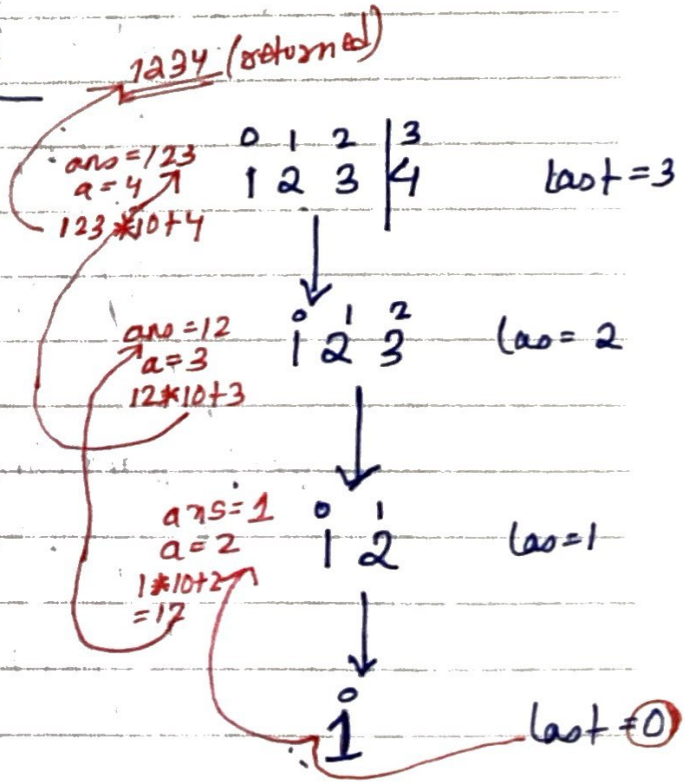
$$TC = O(2^n)$$

String to Integer:

1) i/p = 0001231	2) i/p = 12567
o/p = 1231	o/p = 12567

```
#include<cstring>
int helper(char input[], int last)
{
  if(last==0)
    return input[last]-'0';
  int ans = helper(input, last-1);
  int a = input[last]-'0';
  return ans*10+a;
}
int StringtoNumber(char str[])
{
  int len = strlen(str);
  return helper(str, len-1);
}
```

int — int data type



(Pair Star)

Q) Given a string S, compute recursively a new string where identical characters that are adjacent in original string are separated from each other by '*'

• i/p: hello	• i/p: aaaa
o/p: hel*lo	o/p: a*a*a*a

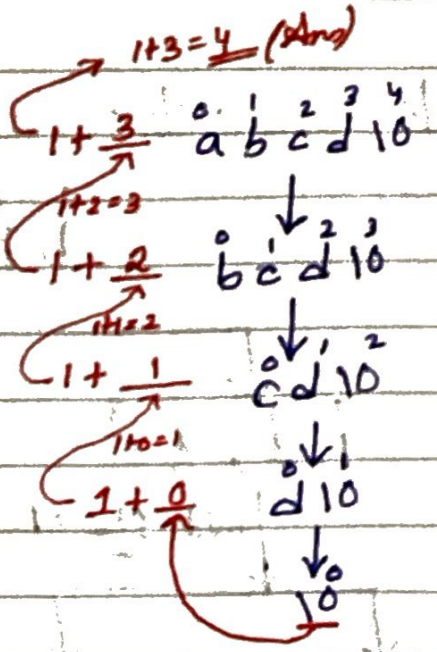
```
#include<cstring>
void helper(char input[], int start)
{
    if(i/p[start]=='\0')
        return;
    helper(input, start+1);
    if(input[start]==input[start+1])
    {
        for(int i=strlen(i/p); i>=start+1; i--)
        {
            i/p[i+1]=i/p[i];
        }
        i/p[start+1]='*';
    }
}
void pairstar(char str[])
{
    helper(str, 0);
}
```

(or)

```
void pairstar(char s[])
{
    if(s[0]=='\0')
        return;
    pairstar(s+1);
    if(s[0]==s[1])
    {
        for(int i=strlen(s); i>=1; i--)
            s[i+1]=s[i];
        s[1]='*';
    }
}
```

To find length of a string -

```
int length(char s[])
{
  if(s[0]=='\0')
    return 0;
  return 1+length(s+1);
}
```



To remove Consecutive Duplicates -

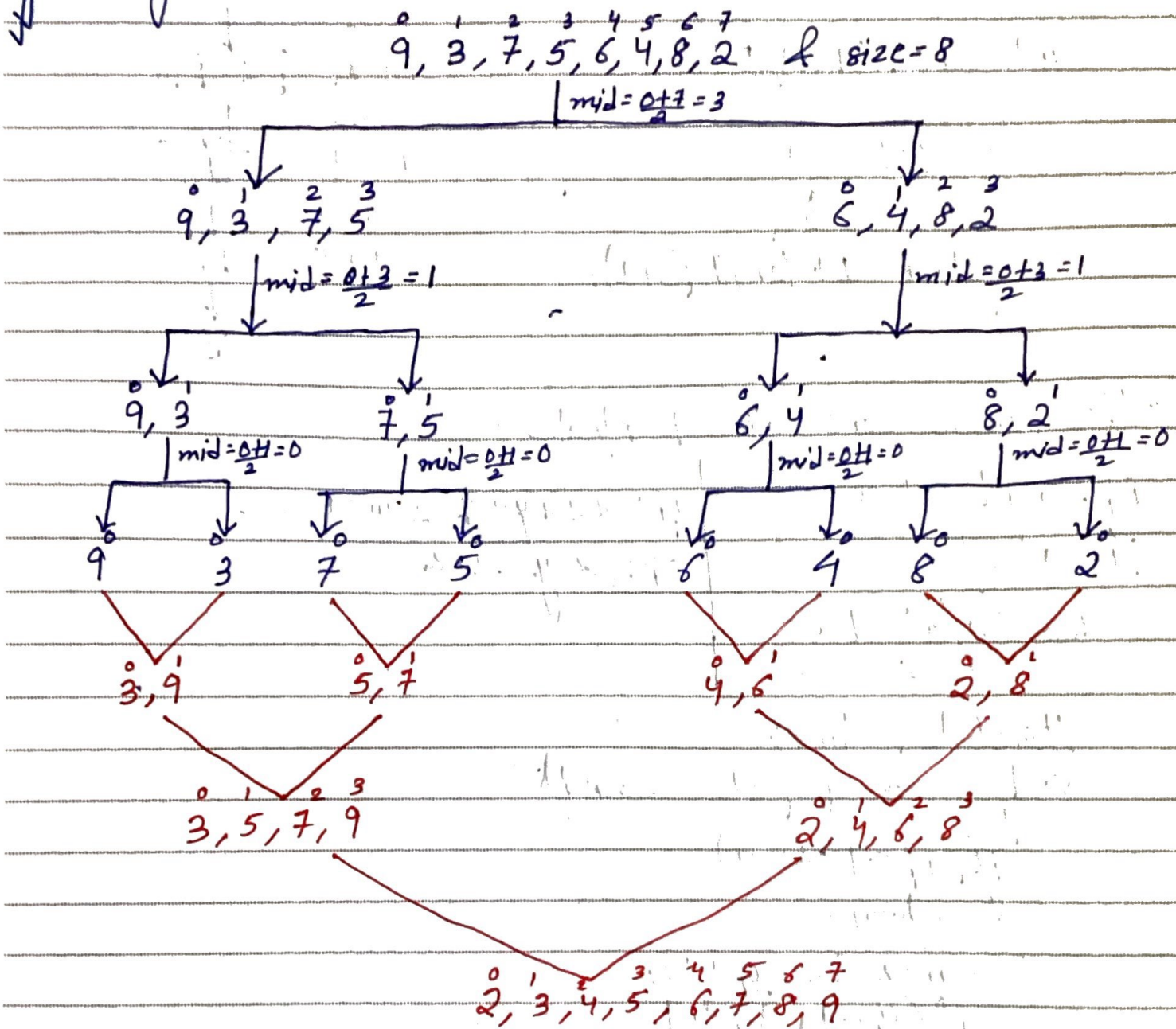
i/p: aabccba
o/p: abcba

here, a/abccba
(rewrite)

i/p: xxxyyyzwwzzz
o/p: xyzwz

```
#include<cstring>
void removeCons(char *input)
{
  if(i/p[0]=='\0')
    return;
  removeCons(i/p+1);
  if(i/p[0]==i/p[1])
  {
    for(int i=0; i<strlen(i/p); i++)
      i/p[i]=i/p[i+1];
  }
}
```

Merge Sort -



$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$= O(n \log n) \quad \text{// for all cases.}$$

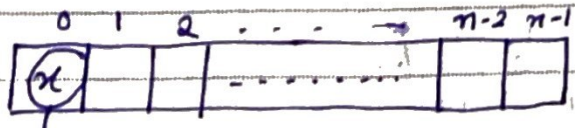
$$SC = O(\log n) + O(n) = O(n)$$

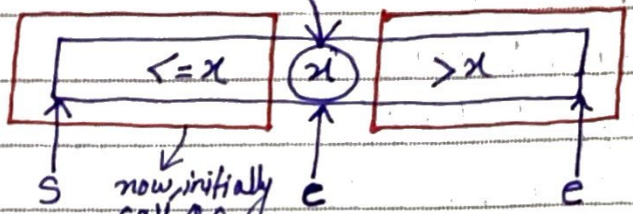
- $O(\log n)$ → for recursive stack
- $O(n)$ → for extra array

merging is done in postorder traversal.

```
void merge(int input[], int start, int mid, int end)
{
    int n1 = (mid-start)+1;
    int n2 = (end-mid);
    int a[n1], b[n2];
    for(int i=0; i<n1; i++)
        a[i] = i/p[start+i];
    for(int i=0; i<n2; i++)
        b[i] = i/p[mid+1+i];
    int i=0, j=0, k=start;
    while(i<n1 && j<n2)
    {
        if(a[i] <= b[j])
            i/p[k++] = a[i++];
        else
            i/p[k++] = b[j++];
    }
    while(i<n1)
    i/p[k++] = a[i++];
    while(j<n2)
    i/p[k++] = b[j++];
}
void mergeSorthelper(int i/p[], int l, int h)
{
    if(l<h)
    {
        int mid = (l+h)/2;
        mergeSorthelper(i/p, l, mid);
        mergeSorthelper(i/p, mid+1, h);
        merge(i/p, l, mid, h);
    }
}
void mergesort(int i/p[], int size)
{
    mergeSorthelper(i/p, 0, size-1);
}
```

→ Quick Sort



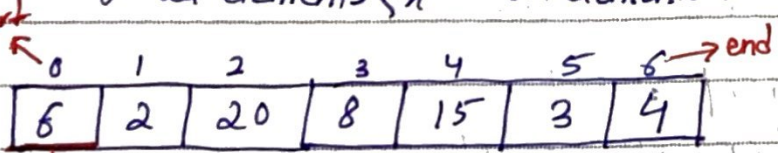


now, initially call QS on this part

from s to (c-1), all elements $\leq x$

from (c+1) to e, all elements $> x$

Say, start

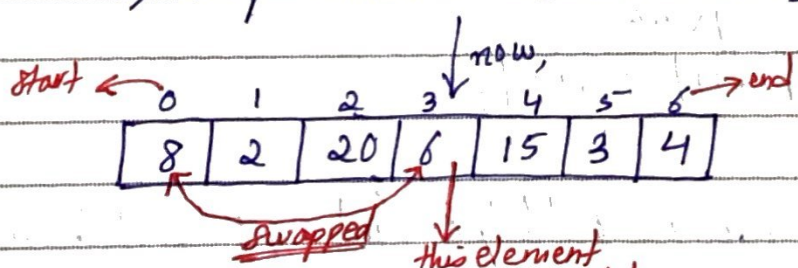


- in Partition( ), count no. of elements ≤ 6

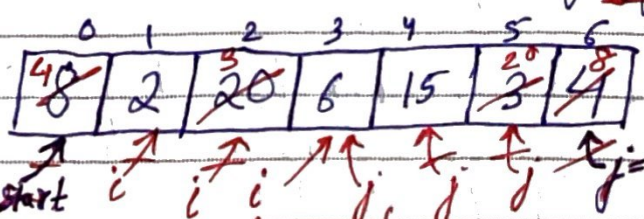
  ∴ count = 3

  now, pivotIndex = start + count
  (c)

  &, swap arr[start] with arr[pivotIndex]



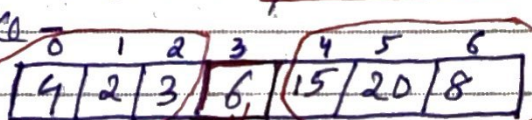
now, to left of element '6' we want all elements ≤ 6 & to right of it > 6



agar equal elements bhi like 5 5 5 ..... to usko mein left side rkhna chahta hu [you can put on right side if you want]

Stop → now partition work is over & return pivotIndex

&, now, array look like →

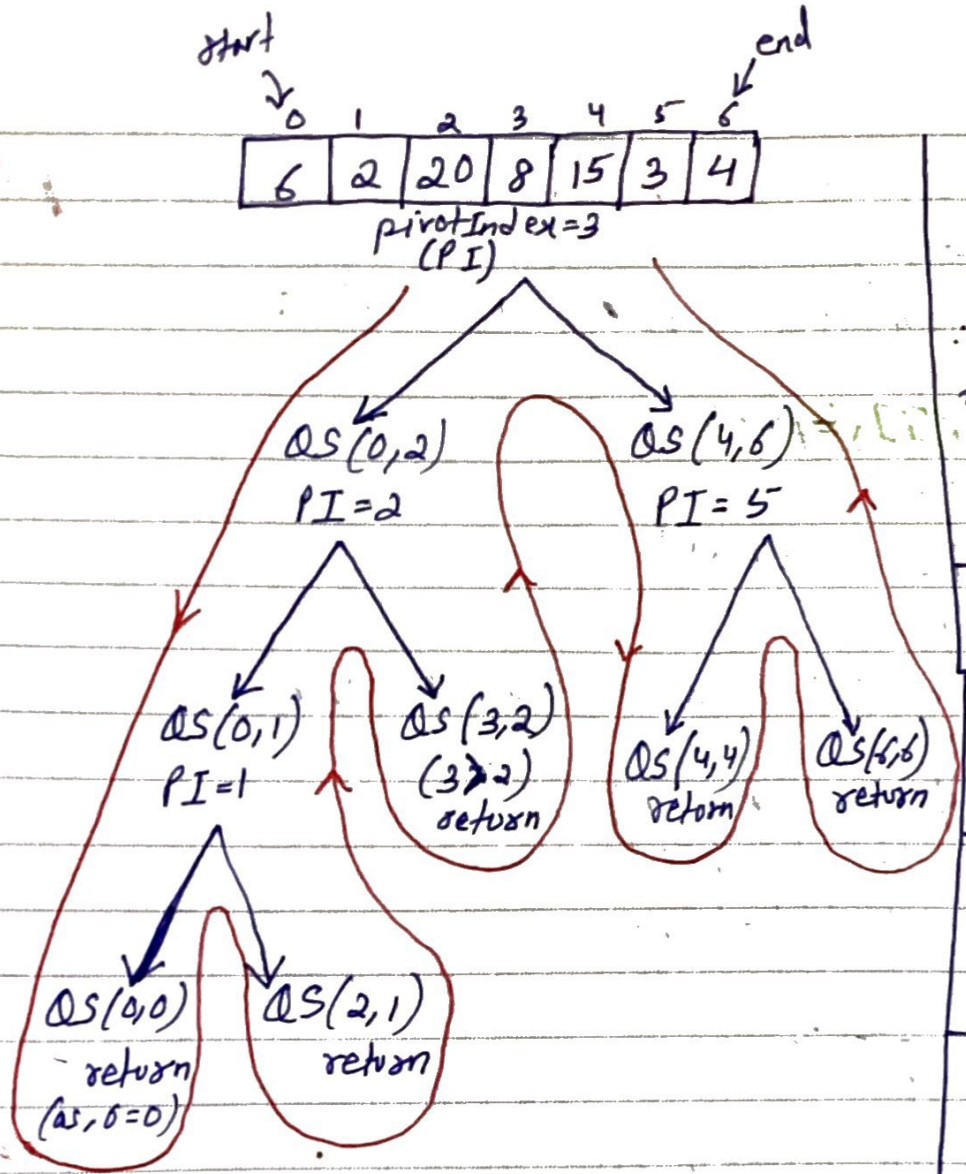


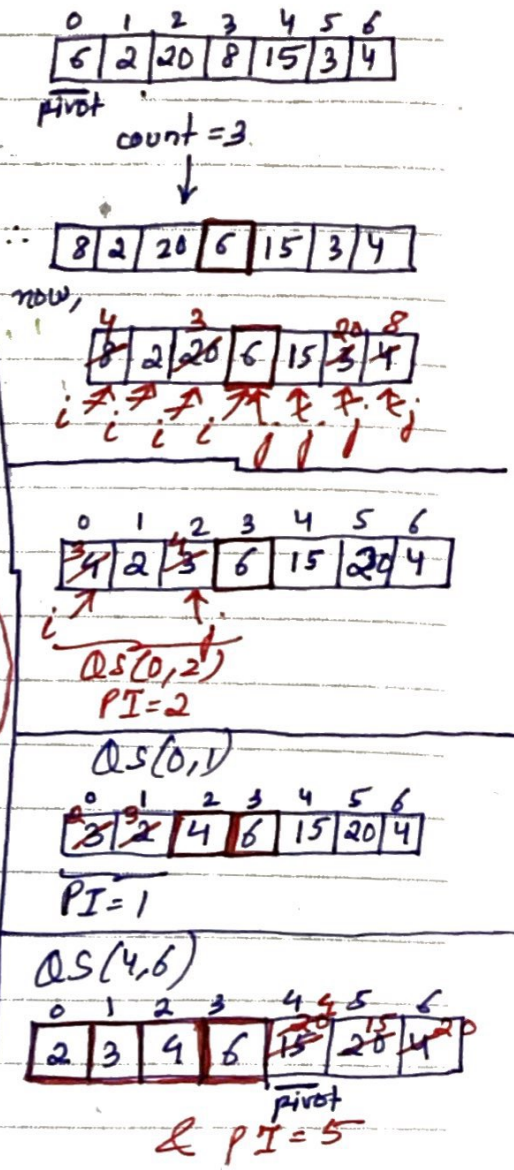
now call QS on this part

pivotIndex = 3 is returned

now call QS on this part

eg)






```
int partition(int i/p[], int start, int end)
{
   int pivot = i/p[start]; // chosing first element as pivot
   // count no. of elements less than or equal to pivot & swap.
   int count=0;
   for(int i= start+1 ; i<=end ; i++)
   {
      if(i/p[i] <= pivot)
         ++count;
   }
   int pivotIndex = start + count;
   int temp = i/p[start];
   i/p[start] = i/p[pivotIndex];
   i/p[pivotIndex] = temp;
```

swap with pivot

PTO

```
//ensure left part contains elements <= pivot & right part > pivot
    int i=start, j=end;
    while(i<pivotIndex && j>pivotIndex)
    {
        while(i/p[i]<=pivot)
            ++i;
        while(i/p[j]>pivot)
            --j;
        if(i<pivotIndex && j>pivotIndex)
        {
            int temp = i/p[i];
            i/p[i] = i/p[j];
            i/p[j] = temp;
            ++i;
            --j;
        }
    }
    return pivotIndex;
}
void QuickSorthelper(int i/p[], int start, int end)
{
    if(start>=end) // 0 or 1 element
        return
    int pivotIndex = partition(i/p, start, end);
    QuickSorthelper(i/p, start, pivotIndex-1);
    Quicksorthelper(i/p, pivotIndex+1, end);
}
void QuickSort(int i/p[], int size)
{
    Quicksorthelper(i/p, 0, size-1);
}
```