In [ ]:

```
#LAB 1 TASK 1


inputted_file = open('input.txt', 'r')
temp = inputted_file.read().split()
output_file = open("output.txt", "w+") #used w+ for writing
#and reading at a same time

even_parity = 0
odd_parity = 0
no_parity = 0
palin_count = 0
nonpalin_count = 0


def paritycheck(item):

    global even_parity
    global odd_parity
    global no_parity

    #if no dot found then it must be integer
    if "." in item:
        no_parity += 1
        return f"{item} cannot have parity and"

    convert= int(item)
    if convert % 2 ==0:
        even_parity+= 1
        return f"{item} has even parity and"
    else:
        odd_parity += 1
        return f"{item} has odd parity and"


def isPalindrome(item):


    global palin_count
    global nonpalin_count


    if len(item) <= 0:
        nonpalin_count += 1
        return f"{item} no word found & is not a palindrome\n"

    n = len(item)
    j = 0

    while j< n /2:
        if item[j] != item[n-1-j]: #Pseudocode part
            nonpalin_count +=1
            return f"{item} is not a palindrome\n"
        j += 1
    else:
        palin_count += 1
```

```
        return f"{item} is a palindrome\n"




for i in range(len(temp)):
    if i % 2 == 0:
        x = paritycheck(temp[i])
        output_file.write(x)
    else:
        x = isPalindrome(temp[i])
        output_file.write(x)


#recordingTXT part
n = (len(temp) / 2)
odd_parity = (odd_parity / n) * 100
even_parity = (even_parity / n) * 100
no_parity = (no_parity / n) * 100
palin_count = (palin_count / n) * 100
nonpalin_count = (nonpalin_count / n) * 100


record = open("records.txt", "w+")
record.write(f"Percentage of odd parity is: {odd_parity}%\n")
record.write(f"Percentage of even parity is: {even_parity}%\n")
record.write(f"Percentage of no parity is: {no_parity}%\n")
record.write(f"Percentage of palindrome is : {palin_count}%\n")
record.write(f"Percentage of non-palindrome is : {nonpalin_count}%")


isPalindrome(temp)
inputted_file.close()
```

In [ ]:

```python
#LAB 1 TASK 4 (MATRIX)

inputted_file = open("input4.txt",'r')
A = []
B = []  #2 matrix n x n given in question
#so taken 2 empty LIST A,B

list= inputted_file.read().split()

for i in range(len(list)):
    if i<(len(list)//2):
        A.append(list[i].split(","))
    else:
        B.append(list[i].split(","))

def Multiply_matrix(A,B):
    C=[[0]*len(B) for i in range(len(B))]
    for i in range(0,len(B)):   #O(n) iterate
        #through row b
        for j in range(0,len(B)):   #O(n) iterate
            #through column B
            for k in range(0,len(B)):   #O(n)
                C[i][j] += int(A[i][k]) * int(B[k][j])
    return C

output= open("output4.txt", mode= "w")
Final= Multiply_matrix(A,B)


output.write(str(Final))
#output.write("/n")
output.write("\nThe worst case time complexity of the program is O(n^3)")
output.close()

#pseudo code
#Procedure Multiply_matrix(A,B)
#Input A,B nxn matrix
#Output C nxn matrix
#begin
#Initialize C as a nxn zero matrix
#for i = 0 to n-1
#for j = 0 to n-1
#for k = 0 to n-1
#C[i,j] += A[i,k]*B[k,j]
#end for
#end for
#end for
#end Multply_matrix
```

LAB 2

In [1]:

```python
#TASK 1 (bubble sort)

def bubblesort(arr):
  for i in range(len(arr)-1):
    flag = False

    for j in range(len(arr)-1): #len(arr) - 1? =>
        #after the first iteration is complete
        #and we get the largest element at the end.
        #Now we need to the len(list1) - 1
      if arr[j] > arr[j+1]:
        #swapping
        arr[j+1], arr[j] = arr[j] , arr[j + 1]
        flag = True

    if flag == False:
      break
  return arr


file1 = open ("input1.txt","r")
inp_file = file1.read().split(' ')
inp_file = [int(i) for i in inp_file] #file er element gula
#akta akta read korbe
#ar int type kore dibe
print(inp_file[1::]) #1 bade cause input er
#1st ta hoilo koita element thakbe
#sheta tai bubble sort hobe
#oita bade naile oi number double chole ashbe
temp = bubblesort(inp_file[1::])
print(temp)

x = ""
for i in temp:
  x+= str(i) + " "

output = open("output1.txt","w")
output.write(x)
output.close()

#Complexity : generally bubble sort
#complexity is θ(n2). The best case θ(n)
#happens when the given array is
#already sorted, then the 1st loop
#will execute 1 time.
#when i = 0 the inner loop run
#and check if any value bigger than prev one
#when arr is already sorted flag
#will be false which will execute the 1st loop 1 time only &
#thus the best casecomplexity
#will be theta(n)
```

```
[3, 2, 1, 4, 5]
[1, 2, 3, 4, 5]
```

In [1]:

```python
#TASK 2

def selectionSort(arr, pas, count):
    for i in range(count): #whole array traversed
        temp = 0 #to store the sorted array part
        min = arr[-1] #sorting from the last index
        for j in range(i, pas):
            if arr[j] <= min:  #FIRST E ARRAY ER
                #LAST ELEMENT TA MIN CONSIDER KORE
                #TRAVERSE THEN IF FOUND MIN WE
                #UPDATE THE MIN AND THEN SWAP
                min = arr[j]
                temp = j #sorted min value array stored here
        arr[i], arr[temp] = arr[temp], arr[i]
    return arr


file2 = open("input2.txt", 'r').read().splitlines()
pas = int(file2[0].split()[0])
count = int(file2[0].split()[1])
array = file2[1].split()
form_arr = []


for i in array:
    form_arr.append(int(i))
print(f"given arr: {form_arr}")

func_call = selectionSort(form_arr, pas, count)
print(f"sort arr by preference:", end=" ")
output2 = open("output2.txt", 'w+')
for i in range(count):
    print(func_call[i], end=" ")
    output2.write(f"{str(func_call[i])} ")



#COMPLEXITY

The time complexity of Selection
Sort is O(N2) as there are two nested loops:

One loop to select an element of Array
one by one = O(N)
Another loop to compare that element
with every other Array element = O(N)


```

```
given arr: [5, 10, 2, 1, 4]
sort arr by preference: 1 2 4
```

In [2]:

```
#TASK 3 insertation sort

#TASK 3

def insertationsort(arr1,arr2,new):
  i = 1 #staring pointer 1
  while i < new:
    j = i - 1 #pointer 2 always before pointer 1

    while j >= 0:
      if int(arr1[j]) < int(arr1[j+1]):
        #we will do 2 swaps as 2 arrays
        arr1[j] , arr1[j+1] = arr1[j+1], arr1[j] #ARR 1 A MARKS
        arr2[j], arr2[j+1] = arr2[j+1], arr2[j] #ARR 2 E ID

      else:
        break

      j -= 1

    i+=1

  return arr2 #CAUSE ARR 2 TE ID GULA ASE JEGULA CHAISE

file3 = open("input3.txt","r").read().splitlines()
new = int(file3[0])
id_list = file3[1].split()
mark_list = file3[2].split()


func_call3 = insertationsort(mark_list,id_list,new)
output = open("output3.txt","w+")
for i in func_call3:
  print(i, end= " ")
  output.write(f"{i}")



#COMPLEXITY :
 The worst-case (and average-case) complexity
    of the insertion sort algorithm is O(n²).
Meaning that, in the worst case, the time taken
to sort a list is proportional to the square
of the number
 of elements in the list.

The best-case time complexity of insertion sort
algorithm is O(n) time complexity.
Meaning that the time taken to sort a list
is proportional to the number of elements
in the list;
 this is the case when the
    list is already
    in the correct order ASCENDING ORDER.
```

```
57
58
```

```
3 2 1 6 4 5
```

In [ ]:

```
#TASK 4

#TASK 4
file4 = open("input4.txt","r")
output4 = open("output4.txt","w")
s = file4.read()
ifile = s.splitlines()
N = int(ifile[0])
A = ifile[1].split(" ")
A = [int(i) for i in A] #file er elemenet 1 ta 1
#ta read korbe and integer type kore dibe



def merge (A,p,q,r):
  n1 = q-p+1 #1st sublist including middle
  n2 = r-q #2nd sublist without middle
#p = FIRST, q = MIDDLE , r = LAST
  #2 temp array for storing 2 sub arrays

  L = [0] * n1
  R = [0] * n2

  #Copy data to temp arrays L[] and R[]
  for i in range(0,n1):
    L[i] = A[p+i]

  for j in range(0,n2):
    R[j] = A[q+1+j]

  f = 0  # Initial index of first subarray
  g = 0 # Initial index of second subarray
  h = p  # Initial index of merged subarray
#maintains the current index of A

  #merging condition
    #(we will fill list A with smaller
    #of L,R subarrays element)
  while f< n1 and g <n2: #f index will always
    #be smaller than len(list)

    if L[f] <= R[g]: #if left item < right item copy in A
      A[h] = L[f] #element merged to the new sub array
      f+=1

    else:
      A[h] = R[g] #if right item < left item copy in A
      g+=1
    h+=1
  #check any value left in LEFT SUBLIST
  while f < n1:
    A[h] = L[f]
    f+=1
    h+=1
  #check any value left in RIGHT SUBLIST
  while g<n2:
```

```
    A[h] = R[g]
    g+= 1
    h+=1
#if any value left we add them in merged sub array list h
def mergesort(A,p,r):
  if p < r: #more than 1 item in the list
    #then we break in half
    q = p+(r-p)//2 #middle formula
    mergesort(A,p,q) #mergesort 1
    mergesort(A,q+1,r) #merge sort 2
    merge(A,p,q,r) #we combine them together
    #with MERGE FUNCTION

  return A

output4.write("merged arr: \n")

if N ==len(A):
  Final = mergesort(A,0,N-1) #N = LEN(ARR) - 1
  final_output = " "
  for h in Final:
    final_output = final_output + str(h) + " "

  output4.write(final_output)

else:
  output4.write("raised error!")

print(final_output)
file4.close()
output4.close()



#TIME COMPLEXITY

Merge Sort is a recursive algorithm.  T(n) = 2T(n/2) + O(n)
The solution of the above recurrence is O(nLogn).
The list of size N is divided into a max of Logn parts,
 and the merging of all sublists into a single list takes O(N)time,
 the worst-case run time of this algorithm is O(nLogn)
```

In [3]:

```
#TASK 5 A

#TASK 5

import time

def partition(A,p,q): #p = low , q = high, A = array
  piv = A[p] #leftmost element as pivot
  i = p #pointer
  for j in range(p+1, q+1):
    if A[j] <= piv:
      i = i+1 # If element smaller than pivot is found
      # swap it with the greater element pointed by i
      A[i], A[j] = A[j] , A[i]

  A[p], A[i] = A[i], A[p] #Swap the pivot element
#with the greater element specified by i

  return i

def quicksort(A,p,r):
  if p< r: #if low < high, p = low , r = high
    q  = partition(A,p,r) # Find pivot element such that
    # element smaller than pivot are on the left
    # element greater than pivot are on the right
    quicksort(A,p,q-1)  # Recursive call on the left of pivot
    quicksort(A,q+1,r) # Recursive call on the right of pivot

  return A


start  = time.time()
file5 = open("input5a.txt","r")
infile5 = file5.read().split()

unsort = " "
for i in infile5:
  unsort+=i + " "

print("given arr:",unsort)


infile5 = [int(i) for i in infile5]  #file er elemenet 1
#ta 1 ta read korbe and integer type kore dibe
sorted_arr = quicksort(infile5, 0 ,len(infile5)-1)

st= " "
for i in range(len(sorted_arr)):
  st+= str(sorted_arr[i]) + " "

unsort = "given arr:" + unsort
st = "sorted arr:" + st
end = "time taken:" + str(time.time())

output5 = open("output5a.txt","w")
output5.write(unsort)
```

```
output5.write(st)
output5.write(end)
print(st)
output5.close()
```

```
given arr:  11 -1 9 4 7
sorted arr: -1 4 7 9 11
```

In [ ]:

```
#partition part from 5a
def partition(A,p,q):
    x=A[p]
    i=p
    for j in range(p+1,q+1):
        if A[j]<=x:
            i=i+1
            A[i],A[j]=A[j],A[i]
    A[p],A[i]=A[i],A[p]
    return i

def find_k(A,p,r,k):
    if p == r:
        if p == k:

            return A[p]
        else:
            return
    else:
        q = partition(A,p,r)
        if q == k:
            return A[q]
        elif q<k:
            return find_k(A,q+1,r,k)
        else:
            return find_k(A,p,q-1,k)


file=open("input5b.txt", mode="r")
inputted_file=file.read().split()

inputted_file=[int(i) for i in inputted_file]
k=inputted_file[0]
array=inputted_file[1::]
out= str(find_k(array, 0, len(array)-1,k-1))
output= open("output5b.txt", mode="w")
output.write(out)
output.close()

```

In [ ]:

```

```