Sorting

dictionary is important

① merge short →

```
def mergesort(l, lt)
    if lt<=1:
        return l
    mid= lt//2
    left= l[:mid]
    right= l[mid:]
    mergesort(left, mid)
    mergesort(right, lt-mid)
    i=j= K=0
    while i< mid and j< lt-mid:
        if right[j] < left[i]:
            l[K]= right[j]
            j+=1
        else
            l[K]= left[i]
            i+=1
        K+=1
    if i=mid:
        l[K:]= right[j:]
    else
        l[K:]= left[i:]
    return l
```

lt → length of array to sort.

mid will be length of left array.

lt-mid will be length of right array.

Time Complexity = O(n log n).

Problem statement —

Count no. of inversions or no. of adjacent swaps required to sort an integer array.

★

```
def mergecount(l, lt):
    if lt == 1:
        return 0
    mid = lt//2
    left = l[:mid]
    right = l[mid:]
    c = mergecount[left,mid] + mergecount[right, lt-mid]
    i=j=k=0
    while i< mid and j< lt-mid:
        if right[j] < left[i]:
            c+= mid-i  → (we have to do these many no. of swaps).
            l[k] = right[j]
            j+=1
        else
            l[k] = left[i]
            i+=1

        k+=1
    if i==mid:
        l[k:] = right[j:]
    else
        l[k:] = left[i:]
    return c
```

Quicksort

```
def quicksort(l, low, high):
    if low == high:
        return
    if high > low:
        mid = partition(l, low, high)
        quicksort(l, low, mid-1)
        quicksort(l, mid+1, high)
```

Driver code

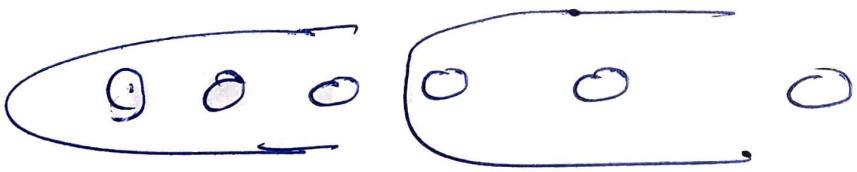
```
l = [ _ _ ]
quicksort(l, _, _)
print(l)
```

```
def quicksort(array, l, r):
    if l >= r:
        return
    mid = partition(array, l, r)
    quicksort(array, l, mid-1)
    quicksort(array, mid+1, r)

def partition(array, l, r):
    pivot = array[r]
    i = l-1
    for j in range(l, r):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
    array[i+1], array[r] = array[r], array[i+1]
    return i+1
```

Maximum contigous subarray problem

Kadane's Algo

```
def findmaxsum(l):
    lt = len(l)
    currres = l[0]
    res = l[0]
    for i in range(1, lt):
        if l[i] > l[i] + currres
            currres = l[i]
            res = max(currres, res)
        else
            currres = l[i] + currres
            res = max(currres, res)

    return res.
```



(String Manipulation)

Problem → Given a string, count total number of palindromic subsequence in this.

Solution → There will be total $\frac{n(n+1)}{2}$ substrings.

We will use dynamic programming.

Ex- abcbaba

{a, b, c, b, a, b, a, bcb, bab, aba, abcba}

output → 11

string started from. \ string end at	a	b	c	b	a	b	a
a	T	F	F	F	T	F	F
b	None	T	F	T	F	F	F
C	None	None	T	F	F	F	F
b	None	None	None	T	F	T	F
a	None	None	None	None	T	F	T
b	None	None	None	None	None	T	F
a	None	None	None	None	None	None	T

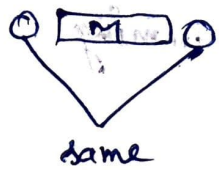
Logic → For a sequence to be palindromic, the necessary and sufficient condition is → first and last letter are same and the part in between them must be palindromic.

step → (1) We creat a 2-D array of size n×n. with all having value None.

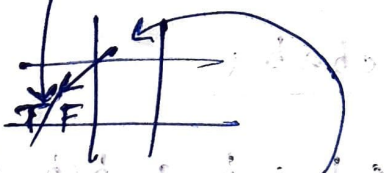
(2) For all single letter sequence, we make value at that index as True.

③ For all sequence of length 2, if both letters are same, make make the value at that index as True else make False.

④ Now traverse the 2-D array diagonally from the 3rd Column and first row.

if 

&& (m) is True



answer is True

~~End~~

⑤ return total no. of True values in the 2-D array.

Code

```
def countpalin (l, n, array):
    p = 2
    for i in range(n-2):
        k = 0
        for j in range (p, n):
            if l[k] == l[j]:
                if array[k+1][j-1] == True:
                    array[k][j] = True
                else
                    array[k][j] = False
                k += 1
            else
                array[k][j] = False
                k += 1
        p += 1
~~for i in~~
```

```
res=0
for i in range(0,n):
    for j in range(i,n):
        if array[i][j]== True:
            res+=1

return res.
```

Driver Code

```
n = int(input())        #length of string
s = input()             # Given string.
import numpy as np
temp= np.array([[None]*n]*n)  → creating 2-D array.
for i in range(n):
    temp[i][i] =True          make all single length string
                              array value as True.
for i in range(0,n-1):
    if s[i]=s[i+1]:           → For all string of length 2:
        temp[i][i+1]=True
    else
        temp[i][i+1] = False
print(countpalin(s,n,temp))
```

longest common subsequence problem

→ Given two string find the longest common subsequence of both (not necessary consecutive sequence). (strings of equal length).

this can be solved using dp.

Code →

```
def lcs(s1, s2, n):                    n = length of strings
    column 1 = [0] * (n+1)
    column 2 = [0] * (n+1)
    for j in range(0,n):
        for i in range(0,n):
            if s1[i] = s2[j]:
                column2[i+1] = 1 + column1[i]
            else
                column2[i+1] = max(column1[i+1], column2[i])
        column1, column2 = column2, column1
    return column1[-1]
```

Ex-

		A	N	K	U	X	P
	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1
K	0	1	1	2	2	2	2
X	0	1	1	2	2	3	3
P	0	1	1	2	2	3	4
R	0	1	1	2	2	3	4
X	0	1	1	2	2	3	(4)

result will be 4 & AKXP.

* in code we could have also used 2-d array as shown above but we used two columns to reduce space complexity & time complexity (to create 2-D array with all value Now 1 or zero).

(KMP Algorithm)

used for pattern matching.

```
def prefixfunction(l, n):
    border = 0
    s = [0]*n    # final return array
    s[0] = 0
    for i in range(1,n):
        while border > 0 and l[i] ≠ l[border]:
            border = s[border-1]
        if l[i] == l[border]
            border += 1
        else
            border = 0
        s[i] = border
    return s.
```

Now to find pattern in text, we do

pattern + "$" + text

↓ we will do prefixfunction on this string.

a process

pattern +

Ex- a b a b c a b x a b a b y

s 0 0 1 2 0 1 2 0 1 2 3 4 0

(Dynamic Programming)

① Hacker problem –

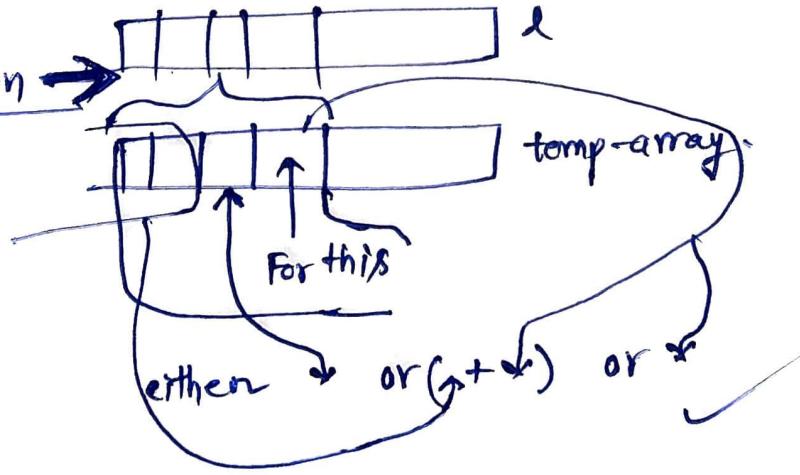
→ Given a list of integers

→ Find maximum sum of numbers present in list which are non adjacent to each other.

* you can also choose no (zero) numbers.

```
def sum(l):
    lt = len(l)
    temp_array = [0] * lt
    temp_array[0] = max(0, l[0])
    temp_array[1] = max(0, l[0], l[1])
    for i in range(2, lt):
        temp_array[i] = max(temp_array[i-1], temp_array[i-2]+l[i], l[i])
    return max(max(temp_array), 0)
```

Explanation →



(Stack and Queue)

Next smaller element to ~~right~~ left :

Given an array. We have to find an array of length, the size of the original array which gives us the index of next smaller element of a given ~~array~~ element in an array. If no such element is found put -1 at that place.

Ex -

input → | 1 | -1 | 0 | 2 | -7 |
0 1 2 3 4

output → | -1 | -1 | 1 | 2 | -1 |

```
def nsetl(l):
    lt = len(l)
    stackvalue = [l[0]]
    stackindex = [0]
    res = [-1]
    for i in range(1, lt):
        while len(stack)!=0 and stackvalue[-1]>=l[i]:
            stackvalue.pop()
            stackindex.pop()
        if len(stack)==0:
            res.append(-1)
            stackvalue.append(l[i])
            stackindex.append(i)
        else:
            res.append(stackindex[-1])
            stackvalue.append(l[i])
            stackindex.append(i)
    return res.
```

```
def nsetr(l):
    lt = len(l)
    res = [-1]
    stackvalue = [l[-1]]
    stackindex = [lt-1]
    for i in range(lt-2, -1, -1):
        while len(stack) != 0 and stackvalue[-1] >= l[i]:
            stackindex.pop()
            stackvalue.pop()
        if len(stack) == 0:
            res.append(-1)
            stackindex.append(i)
            stackvalue.append(l[i])
        else:
            res.append(stackindex[-1])
            stackindex.append(i)
            stackvalue.append(l[i])
    return res[::-1].
```

ngetr

```
def ngetr(l):
    lt = len(l)
    stackvalue = [l[lt-1]]
    stackindex = [lt-1]
    res = [lt]
    for i in range(lt-2, -1, -1):
        while len(stackindex) != 0 and stackvalue[-1] <= l[
```

```
            stackindex.pop()
            stackvalue.pop()

        if len(stack) == 0:
            res.append(lt)
            stackindex.append(i)
            stackvalue.append(l[i])
        else:
            res.append(stackindex[-1])
            stackindex.append(i)
            stackvalue.append(l[i])

    return res[::-1]
```

next greater element to left :-

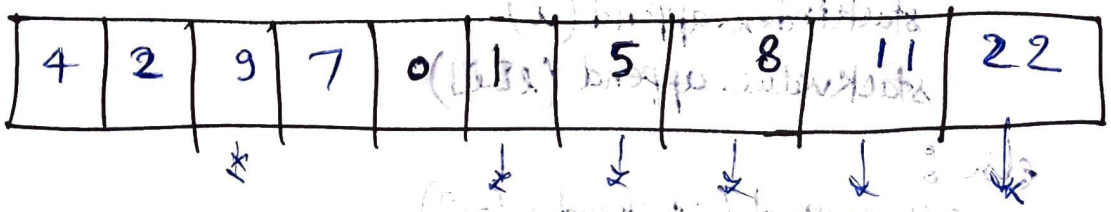
```
def ngetl(l):
    lt = len(l)
    res = [-1]
    stackindex = [0]
    stackvalue = [l[0]]
    for i in range(1, lt):
        while len(stackindex) != 0 and stackvalue[-1] <= l[i]
            stackindex.pop()
            stackvalue.pop()
        if len(stackindex) == 0:
            res.append(-1)
            stackindex.append(i)
            stackvalue.append(l[i])
        else:
            res.append(stackindex[-1])
            stackindex.append(i)
            stackvalue.append(l[i])
    return res.
```

Hackerrank Problem (Poisonous plants)

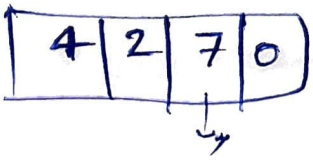
Problem statement →

Given an array of +ve integers.

Ex →



day 1 →

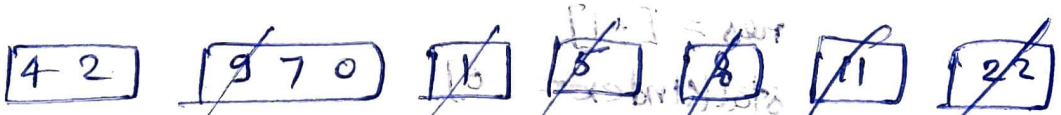


day 2 →

day 3 → Nothing will happen

Explaination → It can be solved by list of stacks
solution

step ① →



step ② → delete 1st element of every stack except 1st one.

step ③ → see if any stacks can be merge.

4 2 Here No. because 2 < 7

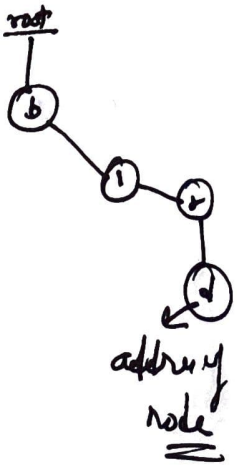
step 4 → repeat step ② untill length of list of stacks == 1.

Trie

```
Class Node:
    def __init__(self):
        self.child = {}
        self.isend = False

Class Trie:
    def __init__(self):
        self.root = Node(); self.words = []

    def insert(self, word):
        lt = len(word)
        start = self.root
        for char in word:
            if char in start.child:
                start = start.child[char]

            else
                Newnode = Node()
                start.child[char] = Newnode
                start = Newnode
        start.isend = True
    def search(self, word):
        start = self.root
        for char in word:
            if char in start.child:
                start = start.child[char]
            else
                return False

        return True
```

```
def tellLastnode(self, word)
    start= self.root
    for char in word:
        if char in start.childdic:
            start= start.childdic[char]
        else
            return None
    return start
```

ex→ bird




```
def final_print(self, pre, node):    # return a list of words starting with a prefix (pre).
    if node.end == True
        self.words.append(pre)
    for i in node.childdic:
        self.finalprint(pre+i, node.childdic[i])
```

Linked List

Singly linked list node

```
class Node:
    def __init__(self, value)
        self.value = value
        self.next = None.

class LinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def insert(self, value):
        if self.head = None:        node = Node(value)
            self.head = node
            self.tail = node
        temp = self.head
        node = Node(value)
        while temp.next != None:
            temp = temp.next
        temp.next = node
        self.tail = node.
```

* Be carefull about end (edge cases) on every problems.

ex – Reverse a link list
- delete a node
- insert at a position
- and check temp.next

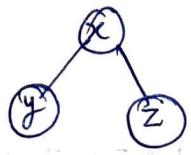
Priority Queue (Coursera)

Binary tree like structure.

Also called heap.

Min heap and max heap.

property →

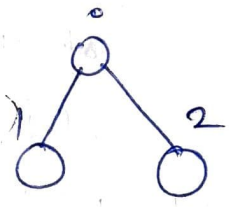


$x \geq y$ and $x \geq z$ → (max heap)

$x \leq y$ and $x \leq z$ (min heap)

minheap

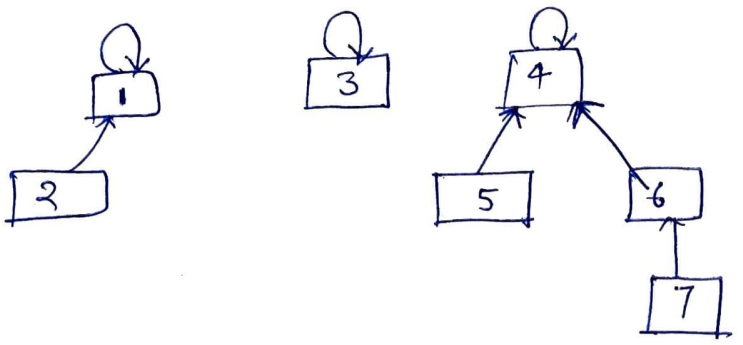
```
Class Heap:
    def __init__(self):
        self.l = []
        self.count = 0          # no. of elements in heap

    def parent(self, i):        # i -> index in list (heap).
        return (i-1)//2
    def leftchild(self, i):
        return 2*i+1
    def rightchild(self, i):
        return 2*i+2
    def swap(self, a, b):
        self.l[a], self.l[b] = self.l[b], self.l[a]
    def shiftup(self, i):
        if i==0:
            return
        par = self.parent(i)
        if self.l[par] > self.l[i]:
            self.swap(par, i)
            self.shiftup(par)
```



```
def insert(self, num):
    self.l.append(num)
    self.count += 1
    self.shiftup(self.count-1)

def shiftdown(self, i):
    temp = i
    lt = self.leftchild(i)
    if self.count-1 >= lt and self.l[lt] < self.l[temp]:
        temp = lt
    rt = self.rightchild(i)
    if self.count-1 >= rt and self.l[rt] < self.l[temp]:
        temp = rt
    if i != temp
        self.swap(i, temp)
        self.shiftdown(temp)

def buildheap(self):
    if self.count <= 1:
        return
    start = self.parent(self.count-1)
    for i in range(start, -1, -1):
        self.shiftdown(i)

def sortlist(self):
    if self.count <= 1:
        return
    self.buildheap()
    temp = self.count
```

```
        for i in range(self.count-1):
            self.swap(0, self.count-1)
            self.count-=1
            self.shiftdown(0)
        self.count=temp

    def changepriority(self, i, new):
        if new > self.l[i]:
            self.l[i]=new
            self.shiftdown(i)
        else:
            self.l[i]=new
            self.shiftup(i)
```

Disjoint Set



☆ There are three groups given.

☆

```
def find(nod):                    # In which group, this
    if node.parent == node:          node falls.
        return node.group
    return find(node.parent)
```

```
class node:
    def __init__(self):
        self.value = value
        self.parent = ~~None~~ self
        self.child = {}
        self.group = value
```

union by compression

```
def union(a, b):                  # a is
    temp = a                                  combining two
    while temp.parent != temp:
        temp = temp.parent
    b.parent = temp
    b.group = temp.group
```