Sorting

Inbuilt method.

(1) sort()

(a) Works only for list
(b) sorts in place
(c) if you want to perform sorting in asc order → L.sort()
(d) " " " " " " desc order → l.sort(reverse=True)
(e) " " " " " " based on key → l.sort(key=function)
(f) T.C = O(nlogn)
(g) It is stable sorting —
(h) It modifies the same list to be sorted.
(i) NO new list will be created.

(2) sorted() function.

(a) It is applicable for any iterable object.
(b) It takes iterable object as input and return new sorted list.
(c) sorted(iterable object) → for asc order
(d) sorted(iterable object, reverse = True) → list with all obj in parameter like reverse & keys works same as sort() desc order.
(e) T.C = O(nlogn)

Both use Tim sort and both are stable.

Time sort is a hybrid algorithms that uses merge sort and insertion sort internally.
T.C = O(nlogn)

eg Stable sort — Bubble sort, ~~selection~~ Insertion sort, merge sort
eg unstable sort — Selection sort, Quick sort, Heap sort.

APRIL
05 '24
14th Week
096-270
FRIDAY

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

eg1 l1 = [5, 10, 15, 1]
Print(l1.sort()) o/p — 1, 5, 10, 15

eg2 l2 = [1, 5, 3, 10]
Print(l2.sort(reverse = True)) o/p — 10, 5, 3, 1

eg3 l3 = ['gfg', 'de', 'course']
l3.sort()
Print(l3) o/p — ['course', 'gfg', 'de'].

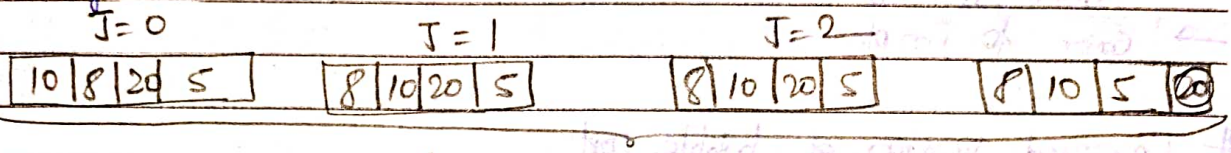
eg4 def fun(s)
    return len(s)
l = ['gfg', 'course', 'python']
l.sort(key = fun)
Print(l) o/p — ['gfg', 'python', 'course']

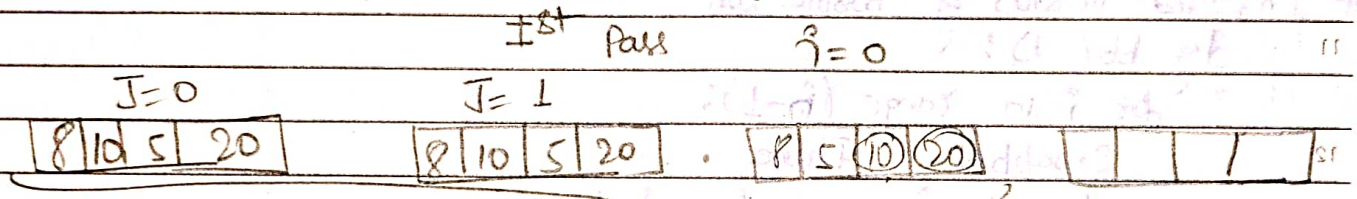
l.sort(key = fun, reverse = True)
Print(l) o/p — ['course', 'python', 'gfg']

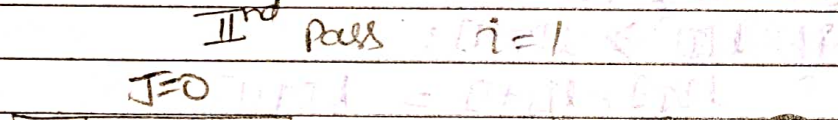
The better part of velour is discretion

Bubble sort

It is a simplest sorting algo that works by repeatedly swapping the adjacent element if they are in wrong order. This algorithms is not suitable for large data set as its average and worst case TC is quite high.







III rd Pass i=2

i=0 : J=0 , swap l[0], l[1]
J=1 , No swap
J=2 , swap l[2], l[3]

i=1 : J=0 , No swapping
J=1 , swap l[1], l[2]

i=2 : J=0 , swap l[0], l[1]

```
def bb(l):
    n = len(l)
    for i in range(n-1):
        for J in range(n-i-1)/(n-1):
            if l[J] > l[J+1]:
                l[J], l[J+1] = l[J+1], l[J]
```

APRIL
07
SUNDAY
'24
15th Week
098-268

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

→ It is stable algorithms.

→ Just change < for desc order.

When to use?

→ When the input is already sorted

When to Not?

↔ Average time complexity is poor.

→ Space is concern

→ easy to implement

Improved version of bubble sort

```
def bb(l):
    for i in range(n-1):
        Swapped = False
        for J in range(n-i-1):
            if l[J] > l[J+1]:
                l[J], l[J+1] = l[J+1], l[J]
                Swapped = True
        if Swapped == False
            return.
```

TC — O[n²] ← Avg, worst case

O[n] ← Best case

$$\text{Swaps} = \frac{n(n-1)}{2}$$

Thinking is the talking of the soul with itself

Selection sort.

→ Tc - $O(n^2)$ [All cases]

→ Does less memory writes compared to quick sort, merge sort insertion sort etc. but cycle sort is optimal in terms of memory writes.

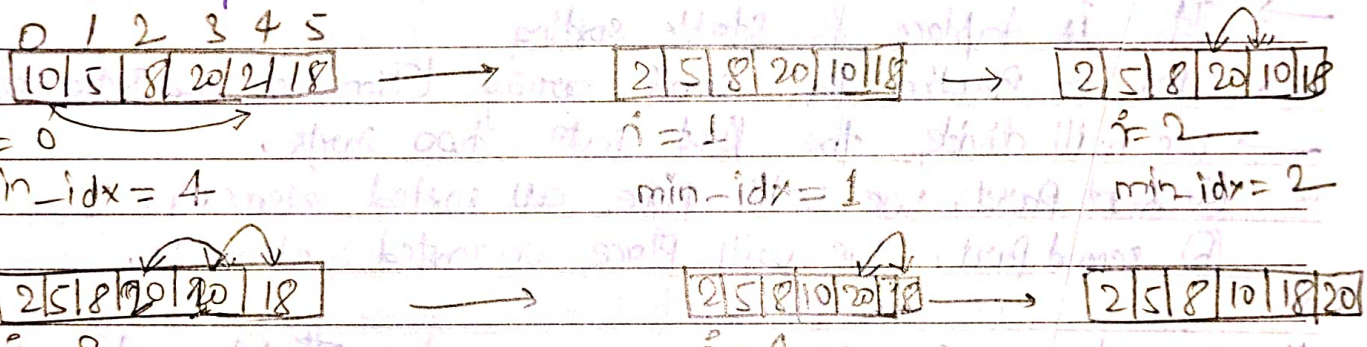
→ In place sorting

→ Not Stable

Best / basic idea for heap sort.

When to use?
When we have insufficient memory, easy to implement

When not to Not?
When time is concern.



{ finding 1st min element in list, move to the 1st location.
" 2nd " " " " " " " " 2nd location and so on

```
def ss(l):
    n = len(l)
    for i in range(n-1):
        min_idx = i
        for j in range(i+1, n):
            if l[j] < l[min_idx]:
                min_idx = j
        l[min_idx], l[i] = l[i], l[min_idx]
```

APRIL
09 '24
TUESDAY
15th Week
100-266

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

Insertion Sort

Insertion sort is a simple sorting algorithms that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

It will take O(n²) — Avrge case / worst case (reverse sorted)
" " " O(n) — Best case. (Already sorted)

→ It is inplace & stable sorting
→ Used in Practice for small array (Tim sort and Intro sort)
→ We will divide the list into two parts.
(a) first part, we will place all sorted elements.
(b) second part, we will place unsorted elements.

When to use?
→ When we have insufficient memory
→ easy to implement
→ When we have continuous inflow of numbers and we want to keep them sorted.

When not?
→ When time is concern

[20, 5, 40, 60, 10, 30]
i=1 [5, 20, 40, 60, 10, 30]
i=2 [5, 20, 40, 60, 10, 30]
i=3 [5, 20, 40, 60, 10, 30]
i=4 [5, 10, 20, 40, 60, 30]
i=5 [5, 10, 20, 30, 40, 60]

0 i-1 i n-1
[sorted | unsorted]

The shame is in the crime not in the punishment

```
def is(l):
    for i in range (1, len(l)):
        n = l[i]
        J = i-1
        while J>=0 and n < l[J]:
            l[J+1] = l[J]
            J = J-1
        l[J+1] = n
```

Merge Sort

1. Divide and Conquer Algorithms (Divide, Conquer and combine)
2. Stable algorithms
3. $\Theta(n \log n)$ time and $O(n)$ Aux Space
4. well suited for linked list, works in $O(1)$ Aux Spaces
5. used in External Sorting
6. In general, for arrays, QuickSort out performs it.

When to use
→ When you need stable sort
→ When average expected time is $\Theta(n \log n)$

When to avoid
→ When space is concern

Merge two sorted list

i/p — a = [10, 15, 20], b = [5, 6, 6, 30]
o/p = [5, 6, 6, 10, 15, 20, 30]

Naive sol^n

```
def merge(a,b):
    res = a+b
    res.sort()
    return res.
```

TC — $(m+n) \times \log(m+n)$ ✓

APRIL
11
'24
THURSDAY
15th Week
102-264

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

Efficient Sol:

```
def merge(a,b):
    res = []
    m = len(a)
    n = len(b)
    i = j = 0
    while i < m and j < n:
        if a[i] < b[j]:
            res.append(a[i])
            i = i+1
        else:
            res.append(b[j])
            j = j+1

    while i < m:
        res.append(a[i])
        i = i+1

    while j < n:
        res.append(b[j])
        j = j+1

    return res.
```

TC = $\Theta(m+n)$

Merge Sub arrays

i/p — a = [10, 15, 20, 11, 13]

o/p — a = [10, 11, 13, 15, 20]

low = 0
high = 4
mid = 2

eg a = [10, 15, 20, 40, 8, 11, 55]

left [] = [10, 15, 20, 40]
Right [] = [8, 11, 55]

low = 0
high = 6
mid = 3

a = [8, 10, 11, 15, 20, 40, 55]

```
def merge(a, low, mid, high):
    left = a[low: mid+1]
    right = a[mid+1: high+1]
    i = j = 0                    res = 0
    k = low
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            a[k] = left[i]
            k = k+1
            i = i+1
        else:
            a[k] = right[j]
            k = k+1
            j = j+1              res += (len(left) - i)

    while i < len(left):
        a[k] = left[i]
        i += 1
        k += 1

    while j < len(right):
        a[k] = right[j]
        j += 1
        k += 1

    return res
```

bracket में जो दिया है Count inversion function में extra add किया only

APRIL

13 '24

15th Week
104-262

SATURDAY

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

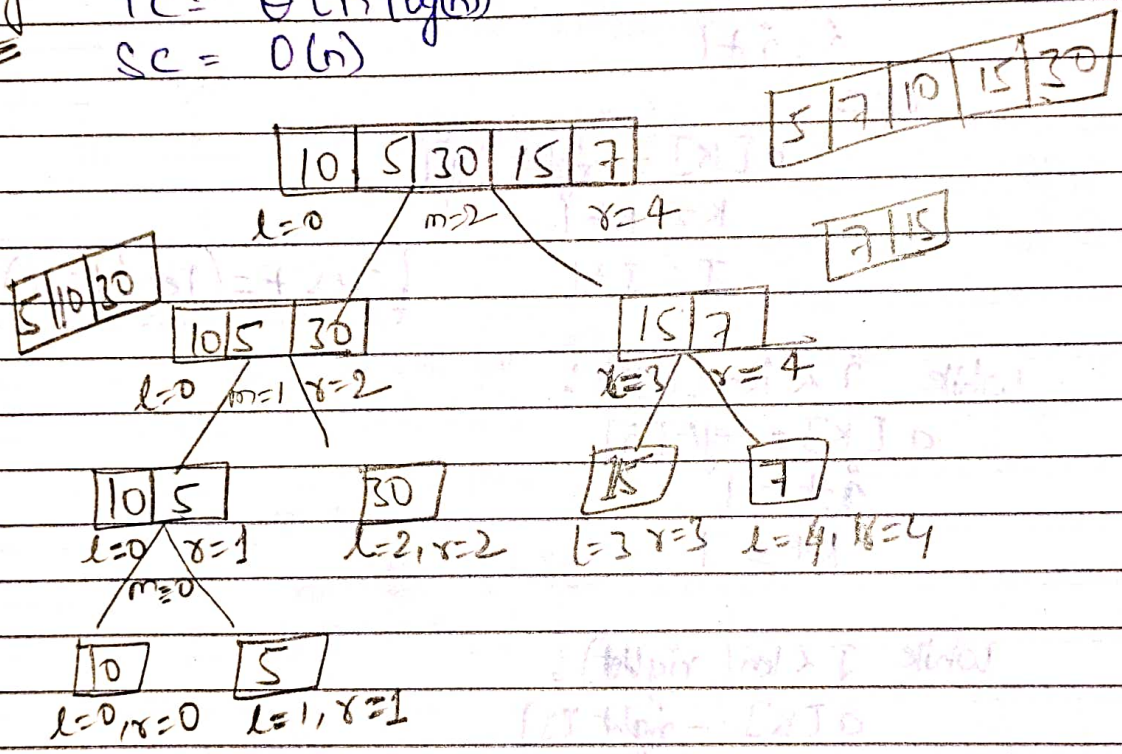
Merge Sort Algorithm.

```
def mergesort(arr, l, r):
    if r > l:
        m = (l + r) // 2
        mergesort(arr, l, m)
        mergesort(arr, m+1, r)
        merge(arr, l, m, r)
```

Calling merge function of previous page

TC = $\Theta(n \log(n))$

SC = $O(n)$



So merge sort algo first compute the mid point then it recursively sort the left half then recursively sort the right half then it merge the two sorted parts.

→ Analysis of Merge sort :-

$T_c = O(n \log n)$ All cases

$T(n) = 2T(n/2) + \Theta(n)$

Union of two sorted array :- $O((m+n) * \log(m+n))$

i/p — a = [3,5,8] , b = [2,8,9,10,15]

o/p — [2,3,5,8,9,10,15]

```
def Printun(a,b):
    c = a + b                          // a.union(b)
    c.sort()
    for i in range(0, len(c)):
        if (i==0 or c[i] != c[i-1])
            print(c[i], end = " ")
```

Intersection of two sorted array :-

i/p = a[] = {3,5,10,10,10,15,15,20}

b[] = {5,10,10,15,30}

o/p = 5 10 15

```
def intersection(a,b,m,n):
    for i in range(m):
        if i>0 and a[i-1]==a[i]:                // a.intersection(b)
            continue
        for j in range(n):
            if a[i] == b[j]:
                print(a[i], end = ' ')
                break
```

TC = $\Theta(n+m)$

SC = $\Theta(1)$

APRIL
15
'24
16th Week
106-260
MONDAY

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

Count inversions in Arrays

Given an array a[], the task is to find the inversion count of a[]. Where two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j

eg: i/p: a = [8, 4, 2, 1]
o/p: 6

Total inversion = $\frac{n(n-1)}{2}$

M1

```
def CntInv(arr, n):
    count = 0
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] > arr[j]:
                count += 1
    return count
```

TC — $O(n^2)$
SC — $O(1)$

M2

```
def CntInv(arr, l, r):
    res = 0
    if (l < r):
        m = (l+r)//2
        res += CountInv(arr, l, m)
        res += CountInv(arr, m+1, r)
        res += CountInvMerge(arr, l, m, r)
    return res

def CountMerge(arr, l, m, r):
    see 2 page back.
```

Partition of Given array.

→ i/p = l = [3, 8, 6, 12, 10, 7] index = p = 5

o/p = l = [3, 6, 7, 8, 12, 10]

(or)

l = [6, 3, 7, 12, 8, 10]

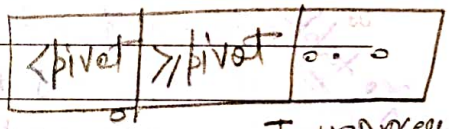
all the elements which are greater than p go to left of p and smaller go to right of p

```
def Partition(arr,b):
    n = len(arr):
    arr[b], arr[n-1] = arr[n-1], arr[b]
    temp = []
    for x in arr:
        if x <= arr[n-1]:
            temp.append(x)
    for x in arr:
        if x > arr[n-1]:
            temp.append(x)
    for i in range(len(arr)):
        arr[i] = temp[i]
```

T = Θ(n)

Stable

pivot = 70



Lomuto Partition. l = 0 h = 6

10	80	30	90	50	70

This algorithm works by assuming the pivot elements as last elements. If any other elements is given as a pivot element then swap it first with the last element. now initialize two variables i as low and J also low, iterate over the array and increment i when arr[J] <= pivot and swap arr[i] and arr[J] otherwise increment only J. After coming out from the loop swap arr[i] with arr[h]. Thus i stores the pivot element.

```
def lomuto(arr, l, h):
    pivot = arr[h]
    i = l-1
    for J in range(l,h):
        if arr[J] <= pivot:
            i = i+1
            arr[i], arr[J] =
                arr[J], arr[i]
    arr[i+1], arr[h] =
        arr[h], arr[i+1]
    return i+1
```

APRIL

17

'24

16th Week
108-258

WEDNESDAY

S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

Hoare's partition Scheme:

Hoare's partition scheme works by initializing two indexes that start at two ends, the two indexes move towards each other until an inversion is (A smaller value on the left side and greater value on the right side) found.

When an inversion is found, two values are swapped and the process is repeated.

```
def hoare (arr, l, h):
    pivot = arr[l]
    i = l-1
    J = h+l
    while True:
        i = i+1
        while arr[i] < pivot:
            i = i+1
        J = J-1
        while arr[J] > pivot:
            J = J-1
        if i >= J:
            return J
        arr[i], arr[J] = arr[J], arr[i]
```

Not a table

Quick Sort

Like merge sort, quicksort is advide & conquer algorithms. It picks an element as a pivot and partitions the given array around the picked pivot.

→ Divide and conquer algorithms

→ Worst case Time : $O(n^2)$

→ Despite $O(n^2)$ worst case, it is considered faster, because of the following reasons.

(a) In-Place
(b) Cache Friendly
(c) Average case in $O(n \log n)$
(d) Tail recursive.

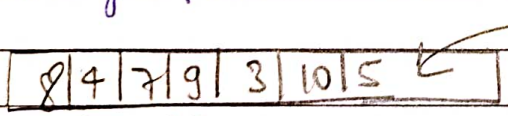
→ Partition is key function (Naive, Lomuto, Hoare)
↳ Stable

When to use:-
→ When average expected time is $O(n \log n)$

When to avoid:-
→ When space is concern
→ When you need stable sort.

Lomuto Quick sort.

→ Always pick the last element as a pivot.

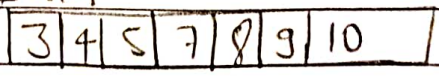


After partition:- 

p = 2

After qSort(arr, 0, 1) 

will be done

After qSort(arr, 3, 6) 

APRIL
19
'24
FRIDAY
16th Week
110-256

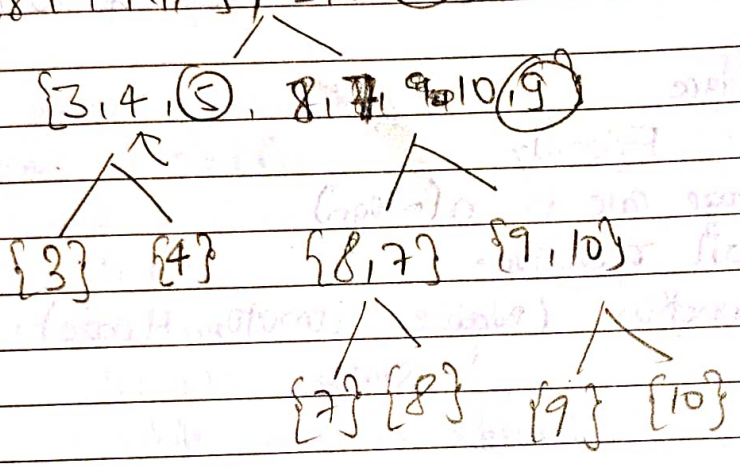
S	M	T	W	T	F	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

APR 2024

```
def qsort(arr, l, h):
    if l < h:
        p = partition(arr, l, h)
        qsort(arr, l, p-1)
        qsort(arr, p+1, h)
```

```
def lomuto(arr, l, h):
    pivot = arr[h]
    i = l-1
    for J in range(l, h):
        if arr[J] <= pivot:
            i = i+1
            arr[i], arr[J] = arr[J], arr[i]
    arr[i+1], arr[h] = arr[h], arr[i+1]
    return i+1
```

eg {8, 4, 7, 9, 3, 10, 5}



Kth smallest element

M-1

```
def kthsmall(arr, n, k):
    arr.sort()
    return arr[k-1]
```

TC = O(n log n)
SC = O(1)

M-2

```
def kthsmall(arr, k):
    l = 0
```