Sorting

Inbuilt method.

① sort()

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

② sorted() function.

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

⇒ Both use Tim sort and both are stable.

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

eg Stable sort — Bubble sort, - Insertion sort, merge sort
eg un stable sort - Selection sort, Quick sort, Heap sort.

APRIL
05
FRIDAY
'24
14th Week
096-270

Stable sort:— It preserved the natural (or) same order in sorted output as they appear in the input data set.

Stable — bubble sort, insertion sort, merge sort
unstable — Quick sort, Heap sort, Selection sort

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

eg4
```
def fun(s)
    return len(s)
l = ['ggg', 'course', 'Python']
l.sort(key = fun) # will be sorted based on length.
Print(l)        o/p — ['ggg', 'Python', 'course']
l.sort(key = fun, reverse = True)
Print(l)        o/p — ['course', 'Python', 'ggg']
```

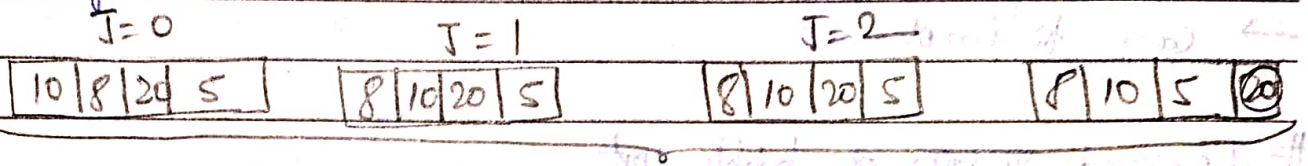
eg5
```
t = (10,2,5,1)
Print(sorted(t)) # [1,2,5,10]
s = {"ggg", "courses", "Python"}
Print(sorted(s)) # ["courses", 'ggg', 'python']
```

eg6
```
s = "ggg"
Print(sorted(s)) # ['g', 'g', 'g']
```

eg7
```
d = {10: "ggg", 15: 'de', 5: 'courses'}
Print(sorted(d)) # [5,10,15]
```

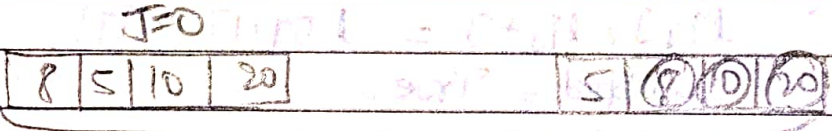
Bubble sort

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.



I^st Pass i=0

II^nd Pass i=1



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]
    return l
```

APRIL
07
'24
15th Week
098-268
SUNDAY

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				

→ It is stable algorithms.
→ Just change '<' for desc order.

When to use?
→ When the input is already sorted
→ Space is concern
→ easy to implement

When to Not?
→ Average time complexity is poor.

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 [already sorted]

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

Selection sort

→ Tc - O(n²) [All case], SC - O(1)

09 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

10 → Not Stable

Basic idea for heap sort-

When to use?

When we have insufficient memory, easy to implement

When not to Not?

When time is concern.

12

Eg: 0 1 2 3 4 5

| 10 | 5 | 8 | 20 | 2 | 18 |

i = 0, min_idx = 4

| 2 | 5 | 8 | 20 | 10 | 18 |

i = 1, min_idx = 1

| 2 | 5 | 8 | 20 | 10 | 18 |

i = 2, min_idx = 2

03 → | 2 | 5 | 8 | 10 | 20 | 18 |

i = 3, min_idx = 4

| 2 | 5 | 8 | 10 | 20 | 18 |

i = 4, min_idx = 5

| 2 | 5 | 8 | 10 | 18 | 20 |

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

```
06 def ss(l):
       n = len(l)
07     for i in range(n-1):
           min_idx = i  → initialize with 0 index element
           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]
       return l
```

In ascending order sign - [<]
" descending " sign - [>]

→ (n-1) Comparison we are doing.

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 Intra 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.

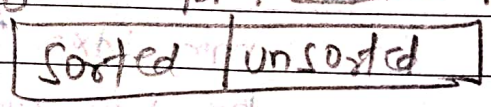
When to use?

→ 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

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



TC = Insertion sort = Bubble sort

```
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. [All Case]
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
THURSDAY
'24
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 = $\Theta(m+n)$

a = [10,15]
b = [5,6,6,30,40]
res = [], i = 0, j = 0
res = [5], i = 0, j = 1
res = [5,6], i = 0, j = 2
res = [5,6,6], i = 0, j = 3
res = [5,6,6,10], i = 1, j = 3
res = [5,6,6,10,15], i = 2, j = 3
res = [5,6,6,10,15,30], j = 4
res = [5,6,6,10,15,30,40], j = 5

rest element of 1st element.

rest element of 2nd element

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

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

low = 0
high = 6
mid = 3

```
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
```

~~TC = O(m+n)~~

bracket में जो दिया है वो count inversion function में extra add होगा only

APRIL
13
'24
SATURDAY
15th Week
104-262

Merge Sort Algorithms.

```
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)
```

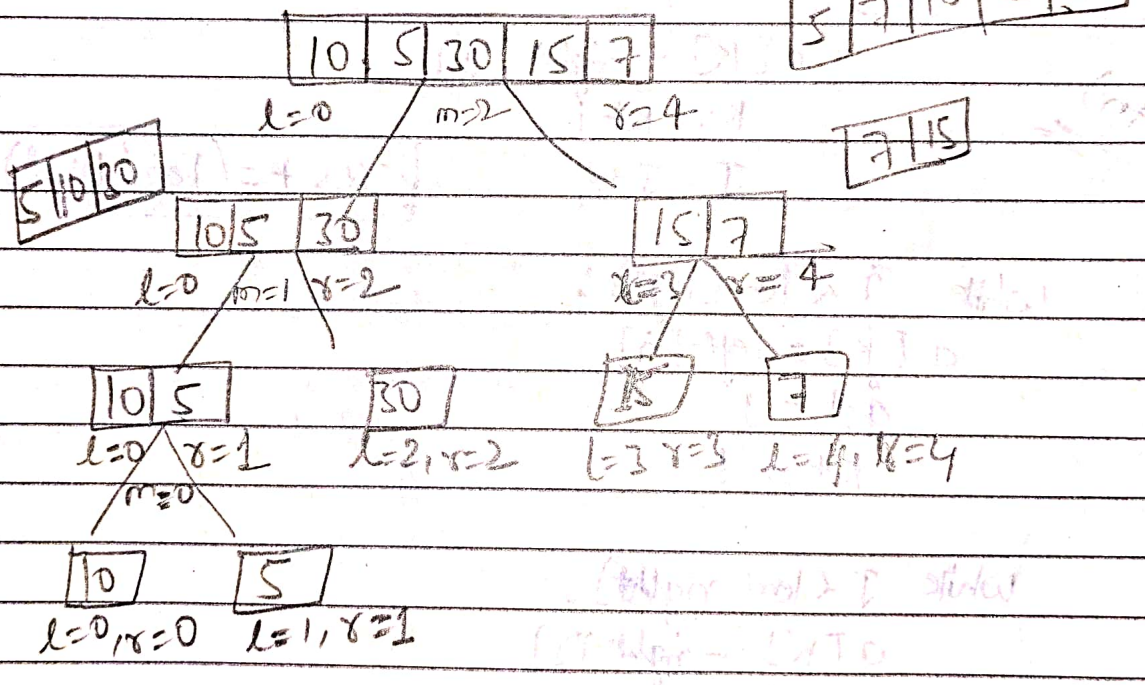
→ If the element is even, you have to take even/2.

→ If the element is odd then, you have to take either odd/2 (or) odd/2 + 1.

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.

The water is running in the river father drink it mother drink it

→ Analysis of Merge sort :-

TC = O(n log n) All cases —

T(n) = 2T(n/2) + Θ(n) → Rewrrence Relation.

Auxiliary Space = O(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]

M1

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

TC = O((m+n) x log(m+n))

M2 // a.union(b)

TC = Θ(m+n)

= Θ(a+b)

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]:
            continue
    for j in range(n):
        if a[i]==b[j]:
            print(a[i], end=' ')
            break
```

// a.intersection(b)

TC = Θ(n*m)

SC = Θ(1)

APRIL
15
MONDAY
'24
16th Week
106-260

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 if array in decreasing order} = $\frac{n(n-1)}{2}$

Method 1

```
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 — $O(1)$

Method 2

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

TC: $O(n \log n)$

m = l+r/2

Every inversion (x, y) where (x > y) has three possibilities:
(a) both x & y lie in the first half.
(b) both x & y lie in the second half.
(c) x in the first half & y in the second half.

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]

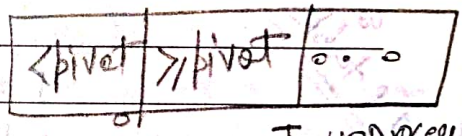
(or)

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,p):
    n = len(arr):
    arr[p], arr[n-1] = arr[n-1], arr[p]
    temp = []
    for x in arr:
        if x <= arr[n-1]:
            temp.append(x)
T = O(n)
    for x in arr:
        if x > arr[n-1]:
            temp.append(x)
Stable
    for i in range(len(arr)):
        arr[i] = temp[i]
```

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]. This 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:
T=O(n)      i = i+1
S=O(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

Lomuto Partition – It places the last element at its correct position then moves all the smaller element on the left of it & greater element on right of it.

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, lb):
    pivot = arr[lb]
    i = lb-1
    J = hb+1
    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]
```

In lomuto we are considering last element as pivot but in Hoare's partition we are considering first element as pivot.

TC = O(n)

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 : $O(n^2)$, Best case Time : $O(n \log n)$

→ 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 $\Theta(n \log n)$

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

Lomuto Quick sort.

→ Always pick the last element as a pivot.

8	4	7	9	3	10	5

pivot

After partition:-

5<	5	>=5

p = 2

After qsort(arr, 0, 1)

3	4	5	>=5

sorted — will be done

After qsort(arr, 3, 6)

3	4	5	7	8	9	10

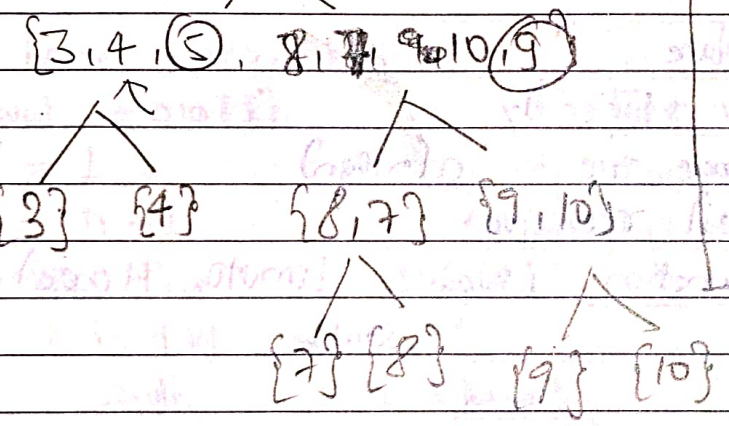
APRIL
19
'24
16th Week
110-256
FRIDAY

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

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

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




```
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
```

Hoare's partition Quick sort

qsort(arr, l, p)

kth smallest element

```
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
```

→ Analysis of Quick sort :- (using Hoare's partition) ⇓ same T.C

(a) Running time for partition of N elements (using Lomuto partition) is O(N)

(b) Best case of Quick sort :- O(nlogn)

(c) Worst case " " " :- $O(n^2)$ [when array is sorted (or) reverse sorted]

(d) Auxilary space at worst case : O(n)

(e) " " " best / Average case : O(logn)

Heap sort

→ Heap sort is a comparison based sorting technique based on binary heap DS.

→ It is an optimization over selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

Two steps: ① Build a max heap.
② Repeatedly swap root with the last node, reduce heap size by 1 and heapify.

→ Time complexity : O(nlogn) → Auxilary space : O(1)

→ Not stable.

→ It used in hybrid sorting algorithms like Introsort.

Applications of Heap sort

→ Priority Queue.

→ Resource scheduling.

→ Shortest path algorithms.

APRIL
29
YSUNDAY
124
17th Week
112-254

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				

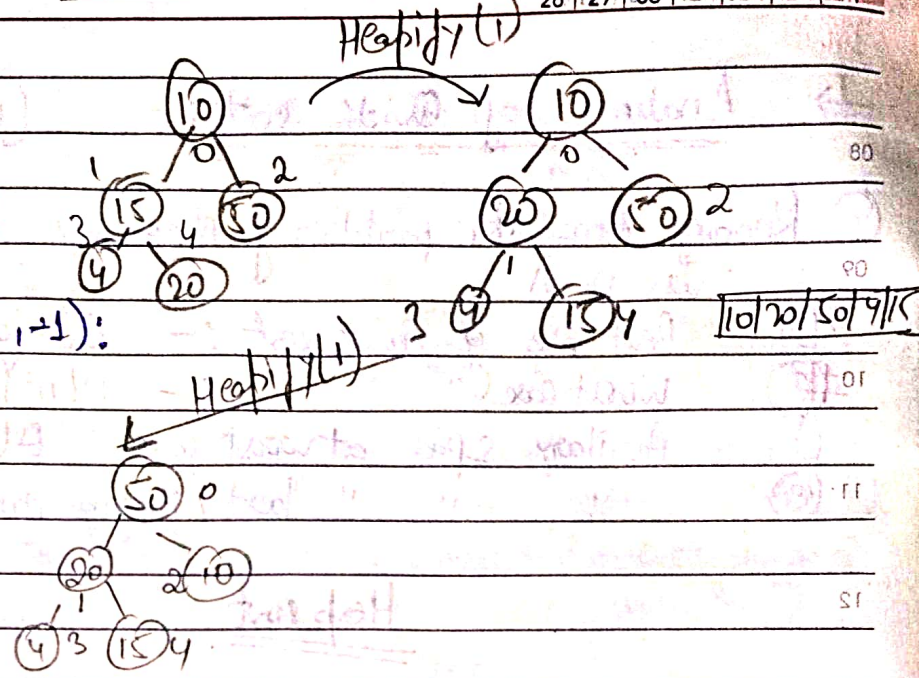
APR 2024

10 | 15 | 50 | 4 | 20

Step 1 Build a max heap.

```
def buildHeap(arr):
    n = len(arr)
    for i in range((n-2)//2, -1, -1):
        maxHeapify(arr, n, i)

def maxHeapify(arr, n, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        maxHeapify(arr, n, largest)
```




```
def heapSort(arr):
    n = len(arr)
    buildHeap(arr)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], ar
        maxHeapify(arr, i, 0)
```