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.

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

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) # will be sorted based on length.
Print(l)        o/p — ['gfg', 'python', 'course']

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

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

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

eg7
```
d = {10: "gfg", 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.

J = 0 | J = 1 | J = 2

10	8	20	5
8	10	20	5
8	10	20	5
8	10	5	20

I^st Pass i = 0

J = 0 | J = 1

8	10	5	20
8	10	5	20
8	5	10	20

IInd Pass i = 1

J = 0

8	5	10	20
5	8	10	20

IIIrd 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
```

→ It is stable algorithm.

→ 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 — $O[n^2]$ — Avg, worst case

$O[n]$ — Best case [already sorted]

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

Selection sort

→ TC - $O(n^2)$ [All cases], SC - $O(1)$

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.

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

→ | 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 |

{ 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]
    return l
```

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

→ initialize with 0 index element

→ (n-1) comparison we are doing.

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 parts. Values from the unsorted part are picked and placed at the correct position in the sorted part.

It will take $O(n^2)$ — Avergge 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 to 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]

0 ... $i-1$	i ... $n-1$
Sorted	unsorted

TC = Insertion sort = Bubble sort

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

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 — $(m+n) \times \log(m+n)$ ✓

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]
low = 0
high = 4
mid = 2

o/p – a = [10, 11, 13, 15, 20]

eg a = [10, 15, 20 40, 8, 11, 55]
low = 0
high = 6
mid = 3

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

TC = O(m+n)

bracket में जो दिया है वो Count inversion function में extra add होगा Only

Merge Sort Algorithms.

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

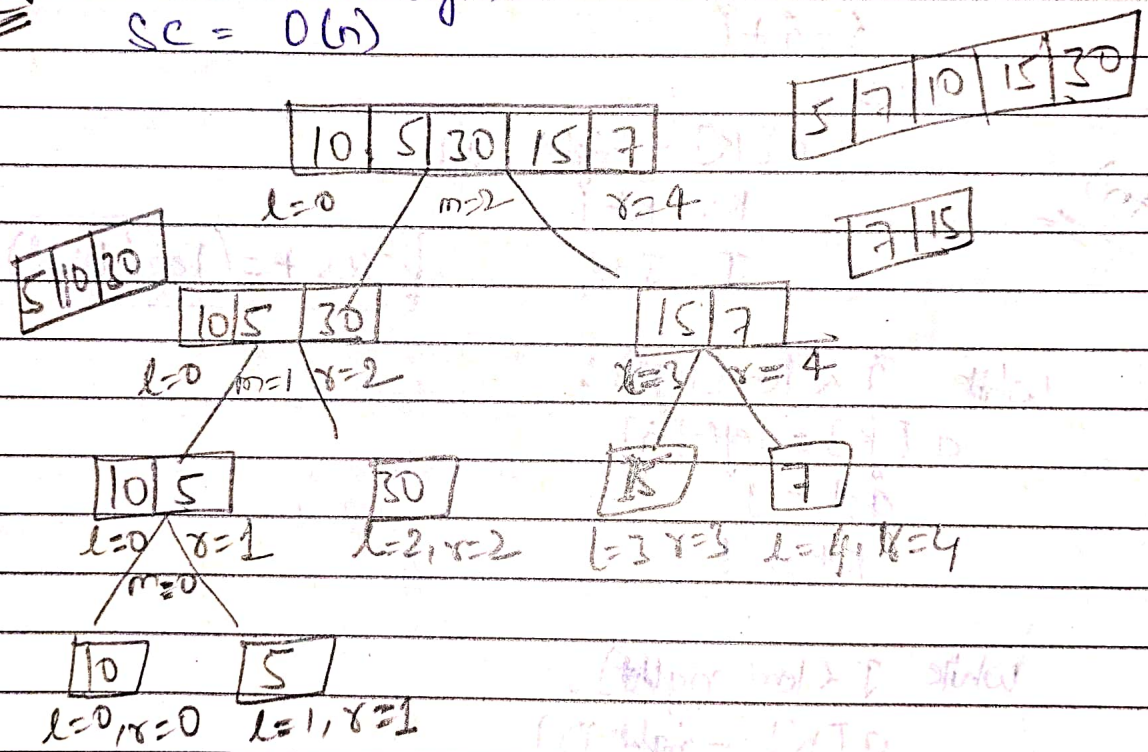
→ If the element is even, you have to take even/2.

→ 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.

→ Analysis of Merge sort :-

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

$T(n) = 2T(n/2) + \Theta(n)$ → Recurrence Relation.

Auxillary 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 = " ")
```

$T_C = O((m+n) \times \log(m+n))$

M2 // a.union(b)

$T_C = \Theta(m+n)$

$= \Theta(a+b)$

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)

$T_C = \Theta(n \times m)$

$S_C = \Theta(1)$

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

M1

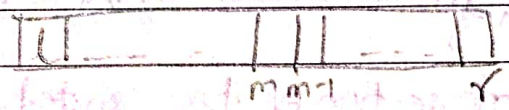
```
def cntinv(arr,n):
    count = 0
    for i in range(n):
        for j in range(i+1, n):
            if arr[i] > arr[j]:
                count += 1
    return count
```

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 += 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)
    for x in arr:
        if x > arr[n-1]:
            temp.append(x)
    for i in range(len(arr)):
        arr[i] = temp[i]
```

T.C = O(n)

Stable

pivot = 70

<pivot	>=pivot	...

l i J unprocess element

Lomuto Partition. l = 0 h = 6

10	80	30	90	50	70

→ This algorithm works by assumming 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[hi]. 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:
            i = i+1
            arr[i], arr[J] = arr[J], arr[i]
    arr[i+1], arr[h] = arr[h], arr[i+1]
    return i+1
```

T.C = O(n)
S.C = O(1)

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, l, h):
    pivot = arr[l]
    i = l-1
    J = h+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 in table

Quick Sort

Like merge sort, quick sort is a divide & 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

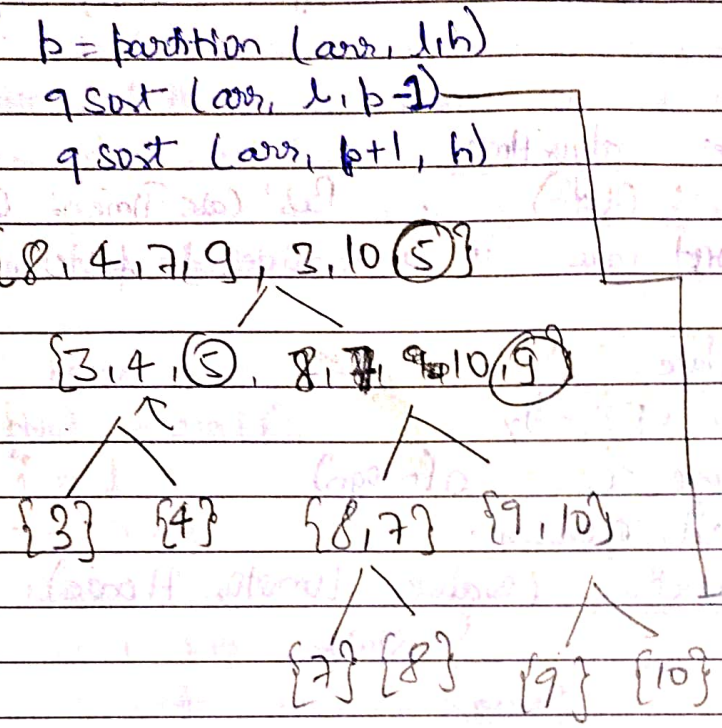
```
def qsort(arr,l,r):
    while l<r:
        p=partition(arr,l,r)
        qsort(arr, l,p)
        l= p+1
```



```
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}



Hoare's partition Quick sort

```
qsort(arr, l,p)
```

Kth smallest Element

M1

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

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

```
def partition(arr, l,r):
    n= arr[r]
    i = l
    for J in range( l,r):
        if (arr[J] <= n):
            arr[i], arr[J] =arr[J], arr[i]
            i +=1
    arr[i], arr[r] = arr[r], arr[i]
    return i
```

M-2

```
def kth small (arr,k):
    l=0
    r= len(arr) -1
    while l<=r :
        p = partition (arr, l,r)
        if p==k -1:
            return p
        elif p> k-1 :
            r= p-1
        else:
            l= p+1
    return -1
```

→ **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 :- $O(n \log n)$

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

(d) Auxilary space at worst case : $\Theta(n)$

(e) " " " best / Average case : $\Theta(\log n)$

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 : $O(n \log n)$ → Auxilary space : $O(1)$

→ Not stable.

→ It used in hybrid sorting algorithms like Introsort.

Applications of Heap sort

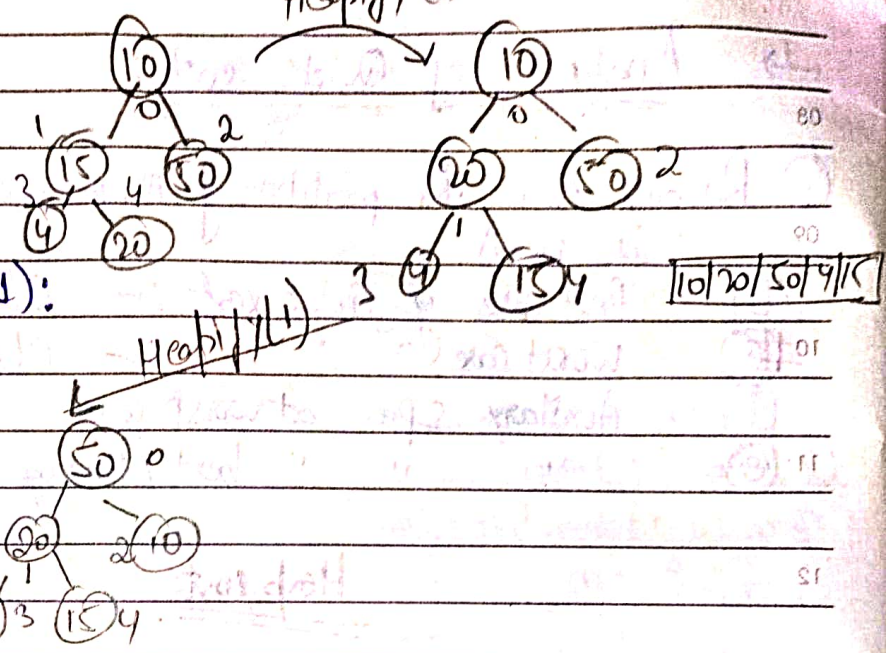
→ Priority Queue.

→ Resource scheduling.

→ Shortest path algorithms.

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], arr[i]
        maxHeapify(arr, i, 0)
```

Overview of Sorting Algorithms.

1. Binary array. → Partition Algo of Quick sort (Lomuto, Hoare)
2. Array with three values.
3. Array of size n and small ranged values. → Counting sort
4. Array of size n and range is of size n^2 (or) n^3 or closer. → Radix sort
5. Array of uniformly distributed data over a range. → Bucket sort.
6. When memory writes are costly. → Selection sort, cycle sort.
7. When adjacent swaps are allowed. → Bubble sort, cocktail sort.
8. When array size is small. → Insertion sort, selection sort.
9. When available extra memory is less. → Shell sort.

General Purpose Sorting Algorithms. — $\Theta(n \log n)$

① merge sort [D&Q] ② Heap sort ③ Quick sort. [D&Q]

08

Hybrid Algorithms (used in libraries).

09 ① Tim sort. (Insertion & merge sort) ② Intro sort. (Quick + Heap + Insertion sort)

10

Counting sort

11

It is a sorting technique based on keys b/w a specific range. It works by counting
12 the number of object having distinct key value (kind of hashing). Then do some arithmetic to calculate the position of each object in the output sequence.

01

M-1

```
def counting (arr, K):
    count = [0] * K
    for x in arr:
        count [x] += 1
    index = 0
    for i in range (K):
        for j in range (count [i]):
            arr [index] = i
            index += i
    Print (arr)
```

Problem with this implementation is It would not work for an array of object like an array of student to be sorted by marks.

M-2

```
def counting (arr, K):
    O = [0] * len(arr)
    C = [0] * K
    for x in arr:
        C[x] += 1
    for i in range(1, K):
        C[i] += C[i-1]
    for i in range (len(arr)):
        arr[i] = O[i]
    for x in reversed(arr):
        O[ [x] - 1] = x
        C [x] -= 1
    Print( arr)
```

02

03

04

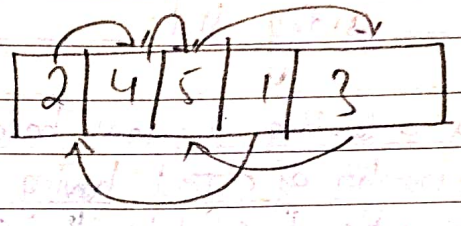
05

06

07 → Not a Comparison Based sorting Algorithms.
→ $\Theta(n+K)$ Time Complexity.
→ $\Theta(n+K)$ Auxilary space.
→ Stable
→ Used as a Subroutine in radix sort.

Cycle sort

→ It is based on the idea that the array to be sorted can be divided into cycles. Cycles can be visualized as a graph. We have n nodes and an edge directed from node i to node j. if the element at i^{th} index must be present at j^{th} index in sorted array. Cycle in arr [] = {2, 4, 5, 1, 3}



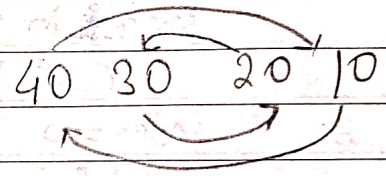
→ It Does minimum memory writes and can be useful for cases where memory write is costly.

→ A worst case $O(n^2)$ sorting algorithms.

→ In place and Not stable.

→ useful to solve question like find minimum swaps required to sort an array.

M-1




```
def cycle (arr):
    w=0
    for cs in range (len(arr) -1):
        item = arr[cs]
        pos = cs
        for i in range (cs+1, len(arr)):
            if arr[i] < item:
                pos = pos + 1
        if pos == cs:
            continue
        while item == arr[pos]: pos += 1
            arr[pos], item = item , arr[pos]
            w += 1
        while pos != cs: pos = cs
            for i in range (cs+1, len(arr)): if arr[i] < item: pos += 1
            while item == arr[pos]: pos += 1
            arr[pos], item = item , arr[pos]
            w += 1
```

Radix sort

→ Comparison based sorting algorithms (Merge sort, Heap sort, Quick sort) lower bound of TC = $\Theta(n \log n)$.

→ linear time sorting algorithm with TC = $\Theta(n+k)$ when element range 1 to k.

→ Sorts digit by digit, starting with the least significant digit, using Counting sort as a subroutine.

eg Sorting by 1s Place — 170, 90, 802, 2, 24, 45, 75, 66

" " 10s Place — 802, 2, 24, 45, 66, 170, 75, 90

" " 100s Place — 2, 24, 45, 66, 75, 90, 170, 802.

```
def counting sort (arr, exp):
    O = [0]* len(arr)
    C = [0]* 10
    for i in range(0, len(arr)):
        index = arr[i] // exp) %10
        C[index] += 1
    for i in range(1, 10):
        C[i] += C[i-1]
    i = len(arr) - 1
    while i >= 0:
        index = (arr[i] // exp) %10
        O[C[index]-1] = arr[i]
        C[index] -= 1
        i -= 1
    for i in range (0, len(arr)):
        arr[i] = O[i]
```

TC = $\Theta(d \times (n+b))$ → total no. of digit.

Applications

(I) Efficient in sorting records keyed by multiple fields.

(II) Historically used in card sorting machines.

```
def radix sort (arr):
    mx = max (arr)
    exp = 1
    while mx/exp > 0 :
        counting sort (arr, exp)
        exp *= 10
```

Bucket sort

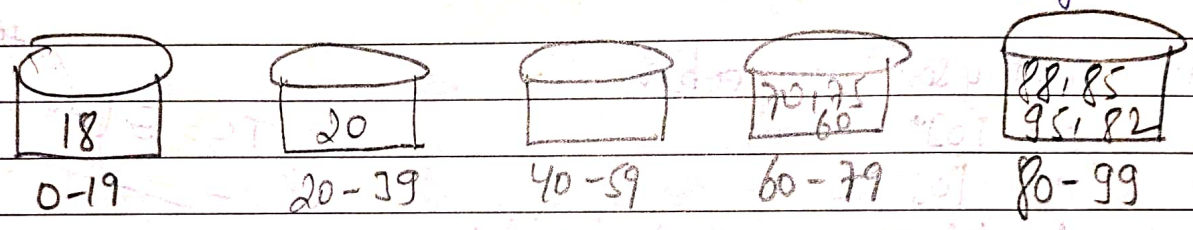
→ 08 Bucket sort is a sorting algorithms useful for sorting number that are uniformly distributed within a specific range.

eg: Sort a large set of floating-point number ranging from 0.0 to 1.0 uniformly

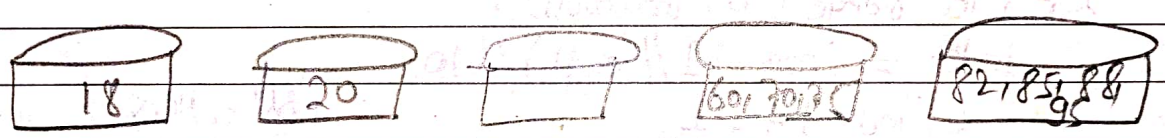
→ 10 Comparison based sorting algo (Merge sort, Heap sort, Quick sort) have a lower bound of $\Omega(n \log n)$.

11 i/p: 20, 88, 70, 85, 75, 95, 18, 82, 60 {Range 0 to 99}

12 Step 1 Scatter



02 Step 2 Sort 03 Bucket



Step 3 Join sorted Bucket

18, 20, 60, 70, 75, 82, 85, 88, 95

05 TC = $\Theta(n)$ [uniformly distributed] → Best case

06 worst case: all item go into a single bucket.
$O(n^2)$ if we use insertion sort
07 $O(n \log n)$ if we use $O(n \log n)$ sorting algo.

```
def bucketsort(arr, k):
    mx = max(arr) + 1
    bkt = [[ ] for i in range(k)]
    for x in arr:
        i = (k*x)//mx
        bkt[i].append(x)
    for i in range(k):
        bkt[i].sort()
    idx = 0
    for i in range(k):
        for j in range(len(bkt[i])):
            arr[idx] = bkt[i][j]
            idx += 1
    for i in range(len(arr)):
        Print(arr[i], end=" ")
```

Minimum Difference in Array

Given an unsorted array, find the minimum difference b/w any pair in given array

i/p: arr[] = [1, 8, 12, 5, 18] 1 8 12 5 18

o/p: 3 min(4)

i/p: arr[] = [8, -1, 0, 3] 1, 8, 12 5, 8 min(3)

o/p: 1

Naive

```
def minDiff(arr):
    res = float("inf")
    for i in range(1, len(arr)):
        for j in range(i):
            res = min(res, abs(arr[i] - arr[j]))
    return res
```

O(n^2)

Efficient

```
def minDiff(arr):
    res = float("inf")
    arr.sort()
    for i in range(1, len(arr)):
        res = min(res, abs(arr[i] - arr[i-1]))
    return res
```

O(n log(n))

① Sort the array
② Compute difference b/w adjacent element.
③ returns the minimum of values found in step2.

Chocolate distribution Problem

Given an array of n integers where each value Represent the number of chocolates in a pocket. Each pocket can have a variable number of chocolates. There are m students, the task is to distribute chocolate pocket such that:

- Each student gets one pocket.
- The Difference b/w the no. of chocolates in the pocket with maximum chocolate and the pocket with minimum chocolates given to the students is minimum.

eg i/p: arr[] = {7, 3, 2, 4, 9, 12, 56} , m = 3
o/p: 2.

Explanation: We have 7 Packets of chocolates and we need to pick three packets for 3 students. If we pick 2, 3 and 4, we get the minimum difference b/w maximum & minimum packet sizes.

```
def minDiff(arr, m):
    if (m==0 or len(arr)==0):
        return 0
    if (len(arr) < m):
        return -1
    arr.sort()
    res = arr[m-1] - arr[0]
    for i in range (1, len(arr) - m + 1):
        res = min(res, arr[i+m-1] - arr[i])
    return res.
```

O(n log n)

① Consider every item as minimum one by one.
[7, 9, 2] — 5
[3, 4, 7] — 4
[2, 3, 4] — 2

② Minimum of all the value is 2

Sort an array with two types of elements.

Sort an array containing two types of elements.

① Segregate positive and negative
i/p: arr[] = {15, -3, -2, 18}
o/p: arr[] = {-3, -2, 15, 18}
(−)ve (+)ve

② Segregate Even and odd
i/p: arr[] = {15, 14, 13, 12}
o/p: arr[] = {14, 12, 15, 13}
== % 2 != % 2

③ Sort a Binary Array
i/p: arr[] = {0, 1, 1, 1, 0}
o/p: arr[] = {0, 0, 1, 1, 1}

Naive

```
def sort(arr):
    n = len(arr)
    temp = [0]*n
    i = 0
    for j in range(0,n):
        if arr[j] < 0:
            temp[i] = arr[j]
            i += 1
    for j in range(0,n):
        if arr[j] >= 0:
            temp[i] = arr[j]
            i += 1
    arr[:] = temp
    print(arr)
```

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

Efficient

```
def sort(arr):
    i, j = -1, len(arr)
    while True:
        i += 1
        while arr[i] < 0:
            i += 1
        j -= 1
        while arr[j] >= 0:
            j -= 1
        if i >= j:
            return
        arr[i], arr[j] = arr[j], arr[i]
    return arr
```

TC = $\Theta(n)$
SC = $\Theta(1)$

Meeting Maximum Guests

i/p: arr[] = {900, 940}, exit[] = {1000, 1030}
o/p: 2 [9:40 to 10:00]

i/p: arr[] = {800, 700, 600, 500}, exit[] = {840, 820, 830, 530}
o/p: 3 [8:00 to 8:20]

Consider a big party where a log register for guest's entry and exit times is maintained. Find the time at which there are maximum guests in party. Note that entries in register are not in any order.

Efficient Sol^n

Consider all events in sorted order & count guests

a = [900, 600, 700]
d = [1000, 800, 730]

600	Arrival	1
700	"	2
730	departure	1
800	"	0

900	Arrival	1
1000	departure	0

Guest_count

```
def maxGuest(arr, dep):
    arr.sort()
    dep.sort()
    n = len(arr)
    i, j = 1, 0
    curr, res = 1, 1
    while (i<n and j<n):
        if (arr[i] <= dep[j]):
            curr += 1
            i += 1
        else:
            curr -= 1
            j += 1
    res = max(res, curr)
    return res.
```

TC = O(n log n)

arr [900, 600, 700], dep = [1000, 800, 730]

After sorting: —

arr = [600, 700, 900], dep = [730, 800, 1000]

Initially i = 1, j = 0 curr = 1, res = 1

arr [i] <= dep [j]

i = 2, j = 0, curr = 2, res = 2

arr [i] > dep [j]

i = 2, j = 1, curr = 1, res = 2

arr [i] > dep [j]

i = 2, j = 2, curr = 0, res = 2

arr [i] <= dep [j]

i = 3, j = 2, curr = 1, res = 2

Sort an array with three types

① Sort an array of 0's, 1's and 2's.

i/p: arr[] = {0, 1, 0, 2, 1, 2}

o/p: arr[] = {0, 0, 1, 1, 2, 2}

② Three way partitioning.

i/p: arr[] = {2, 1, 2, 20, 10, 20, 1}, pivot = 2

o/p: arr[] = {1, 1, 2, 2, 20, 10, 20}

③ Partition around a range.

i/p: arr[] = {10, 5, 6, 3, 20, 9, 40}, range [5, 10]

o/p: arr[] = {3, 5, 6, 9, 10, 20, 40}

Naive Sol^n

```
def sortA(arr):
    temp = []
    for i in arr:
        if i == 0:
            temp.append(i)
    for i in arr:
        if i == 1:
            temp.append(i)
    for i in arr:
        if i == 2:
            temp.append(i)
    for i in range(len(arr)):
        arr[i] = temp[i]
```

TC = O(n)
SC = O(n)

arr = [0,1,1,2,0,1]
After 1st loop:
temp = [0,0] # Copying 0
After 2nd loop:
temp = [0,0,1,1,1] # Copying 1
After 3rd loop:
temp = [0,0,1,1,1,2] # Copying 2
After fourth loop
arr = [0,0,1,1,1,2] # adding to initial

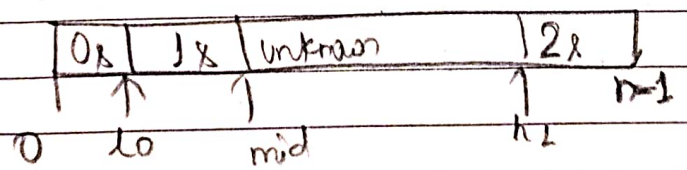
Efficient Sol^n

Dutch National Flag Algorithm

```
def sortA(arr):
    lo, mid, hi = 0,0,(len(arr)-1)
    while mid <= hi:
        if arr[mid] == 0:
            arr[lo], arr[mid] = arr[mid], arr[lo]
            lo += 1
            mid = mid + 1
        elif arr[mid] == 1:
            mid += 1
        else:
            arr[hi], arr[mid] = arr[mid], arr[hi]
            hi -= 1
```

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

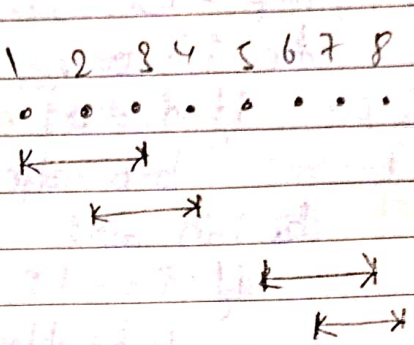
[0,1,2,1,1,2], lo=0, mid=0, hi=5
[0,1,2,1,1,2], lo=1, mid=1, hi=5
[0,1,2,1,1,2], lo=1, mid=2, hi=5
[0,1,2,1,1,2], lo=1, mid=2, hi=4
[0,1,1,1,2,2], lo=1, mid=2, hi=3
[0,1,1,1,1,2,2], lo=1, mid=2, hi=3
[0,1,1,1,2,2], lo=1, mid=4, hi=3



Merge overlapping intervals

i/p: [[1,3], [2,4], [5,7], [6,8]]

o/p: [[1,4], [5,8]]



[1, 3]
[2, 4] > [1, 4]

[5, 7]
[6, 8] > [5, 8]

Efficient Soln

Hints: Two steps

① Sort
② Linear Traversal.

```
def merge(arr):
    arr.sort(key = lambda x: x[0])
    res = 0
    for i in range(1, len(arr)):
        if (arr[res][1] >= arr[i][0]):
            arr[res][1] = max(arr[res][1], arr[i][1])
        else:
            res = res + 1
            arr[res] = arr[i]

    for i in range(res + 1):
        print(arr[i], end = " ")
```

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

Printing result.