Merge sort:

1) break list of n elements into two halves (time O(1))
2) Recursively sort each half (time $T(n/2) + T(n/2)$)
3) merge the two sorted lists (time $O(n)$)

Total time: $T(n) = 2T(n/2) + O(n)$
$T(n) = O(n \log n)$

Master method! (powerful) can solve recurrence relations

This algorithm is basically a comparison based sorting algorithm
- → divides entire dataset into groups of at most two
- → compares each # one at a time, moving the smallest # to left of the pair
- → once all pairs sorted it then compares left most elements of the two leftmost pairs creating a sorted group of four with the smallest # on the left and the largest ones on the right
- → process is repeated until there is only one set

This algorithm divides all data into as small possible sets then compares them.

Best - $O(n)$

Average - $O(n \log n)$

worst - $O(n \log n)$

Example (description):

Mergesort (array[], len, r)

IF r > 1

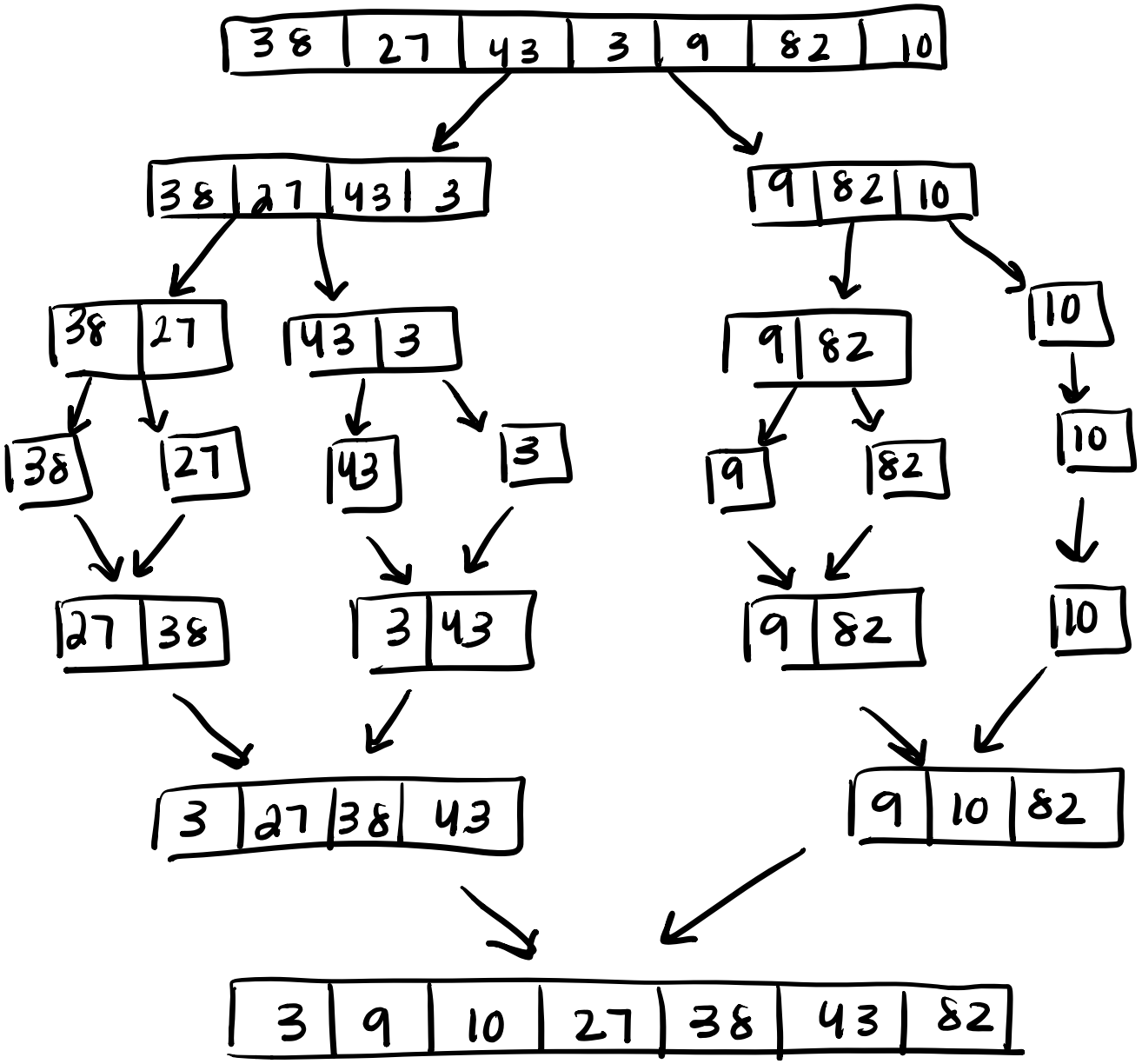
1) Find the middle point to divide the array into two halves:
   middle m = (len + r) / 2

2) Call mergesort for first half:
   call Mergesort (array, len, m)

3) Call Mergesort for second half:
   call Mergesort (array, m+1, r)

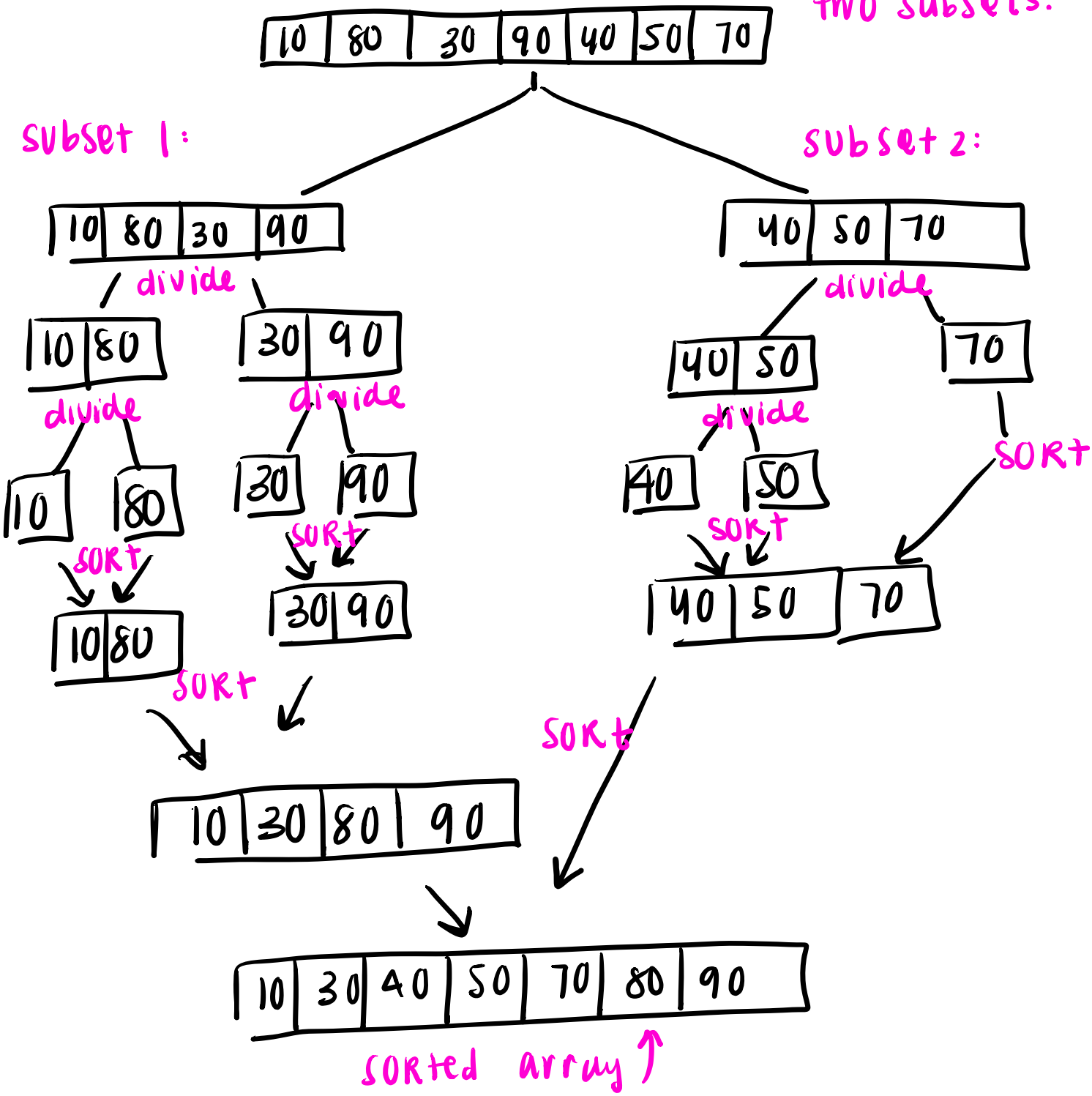
4) merge the two halves sorted in step 2 and 3:
call merge (array, len, m, r)

Example (with numbers):



Example (on my own)

divide enter set into two subsets.



mergesort code:

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2 #Finding the mid of the array
        L = arr[:mid] # Dividing the array elements
        R = arr[mid:] # into 2 halves

        mergeSort(L) # Sorting the first half
        mergeSort(R) # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1

        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
```

QUICKSORT

"dual" version of merge sort

1) un-merge the array
   1) use partition method
   2) break into "halves" of small elements and large elements
   3) note: may not be same size

2) sort each side recursively

3) put the two sides together

Partition = $O(n)$

worst case = $O(n^2)$

avg. case = $O(n \lg n)$

PARTITION Algorithm:

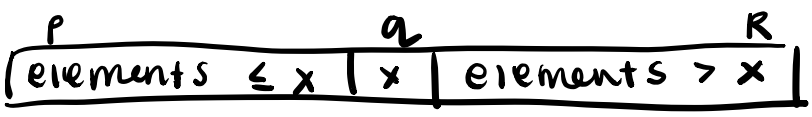
```
Partion ( array , int p , int r )

x = Array [r]
i = p - 1

for j = p to r - 1

    do if (Array [j] ≤ x )

        then i = i + 1
             exchange (A[i] , A[j])

exchange ( A [i + 1] , A[r])

return (i + 1)
```

observations:

in place operation:
merge requires workspace
then copy items back into array

single pass

can choose other elements for pivot
- middle of first, last, middle elements
- choose random location in [p, r] use elements there as pivot
  → Randomized-partition (good)



the sizes of the left and right sides depend on the value of x

example algorithm:

```
Quicksort (p, r)
if p < r then
    q = partition (p, r)
    Quicksort (p, q-1)
    Quicksort (q+1, r)
```

worst case:
partition = cn
let T(n) be the time for quicksort on n elements
worst case, pivot element is the largest (or smallest) remaining element
so, $T(n) = cn + T(n-1)$

expanding $T(n) = cn + T(n-1)$
$= cn + c(n+1) + T(n-2)$
$= cn + c(n+1) + c(n-2) + T(n-3)$
$= c(n + (n-1) + (n-2) + \dots + 1) + T(0)$
$= cn(n+1)/2 + T(0)$
$= O(n^2)$

A comparison based sorting algorithm
- → Divides entire dataset in half by selecting the average element and putting the smaller elements to the left of the average
- → it repeats this process on the left side until it is comparing only two elements at which point to the left side is sorted.
- → when the left side is finished sorting it performs the same operation on the right side

* computer architecture favors the quicksort process

while it has the same Big O (or worse) as many other sorting algorithms, it is often faster in practice than many other sorting algorithms, such as merge sort

it halves the data set by the average continuously until all the information is sorted.

Best case = $O(n)$

Average case = $O(n \log n)$

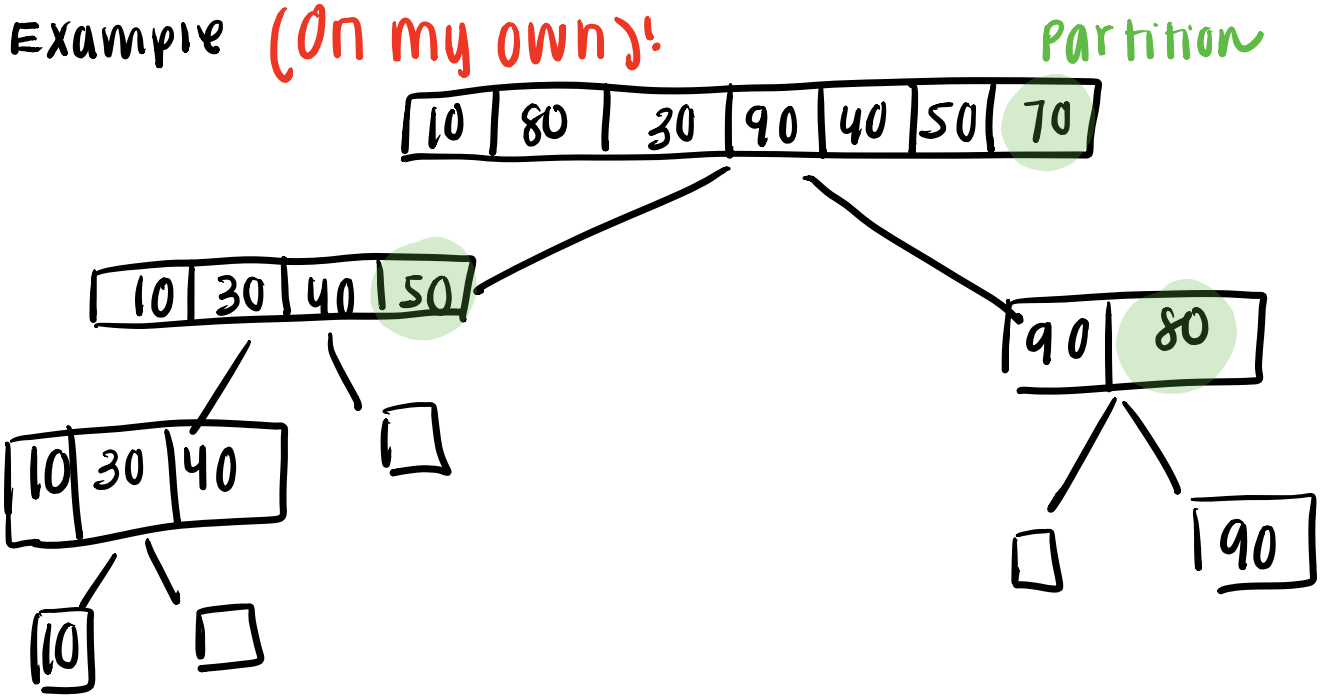
Worst case = $O(n^2)$

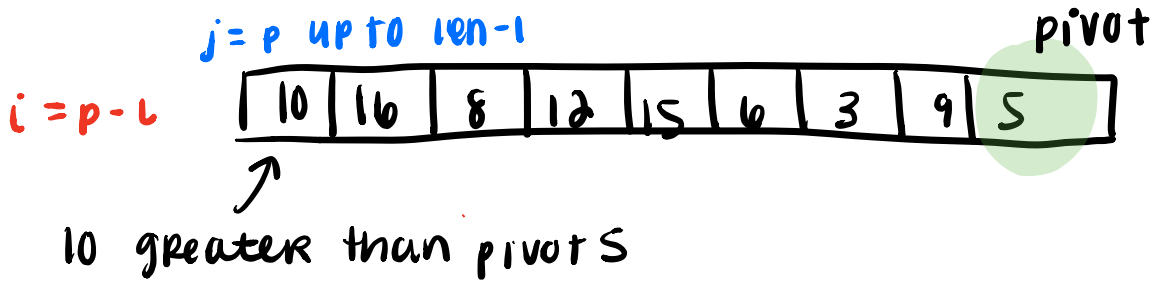
Example (description):

// low = starting index, high = ending index, pi = partition index

```
QuickSort (array, low, high)
  if (low < high):
      pi = partition (array, low, high)
      quicksort (array, low, pi-1)
      quicksort (array, pi+1, high)
```

Example (On my own)!

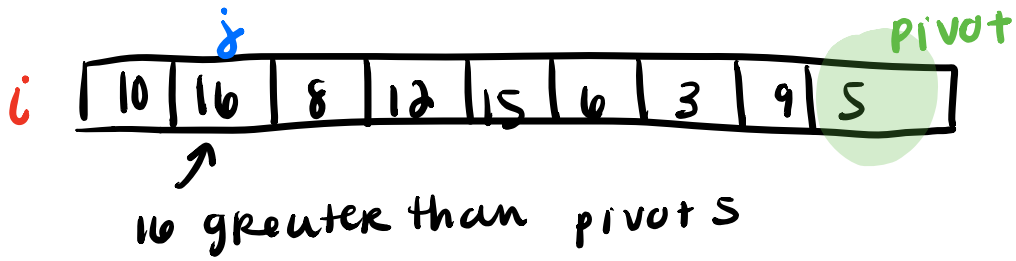




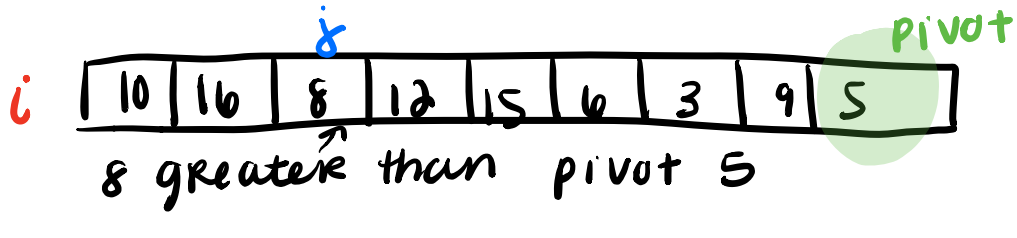
10 greater than pivot 5

if j is larger than pivot, increment j

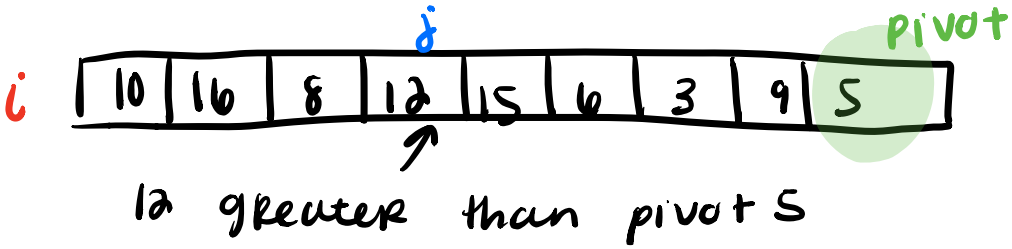
if j is smaller than pivot, increment i and swap i and j



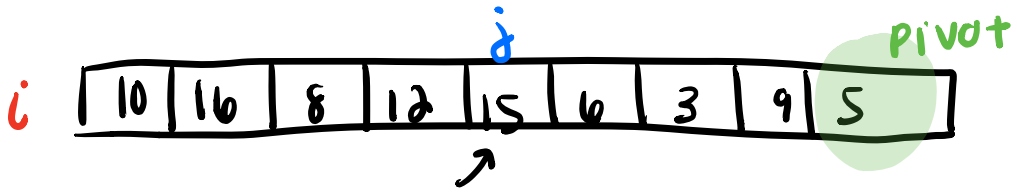
16 greater than pivot 5



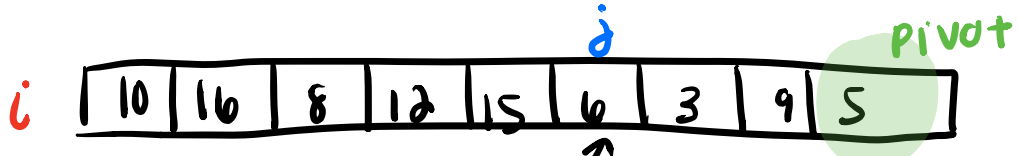
8 greater than pivot 5



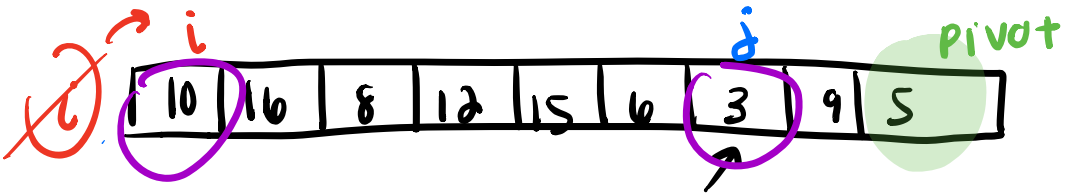
12 greater than pivot 5



is greater than pivot 5

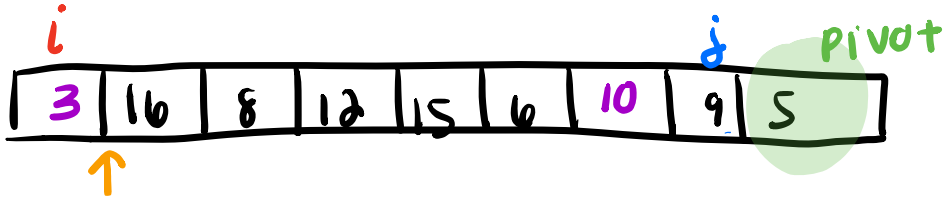


6 greater than pivot 5



3 less than pivot!

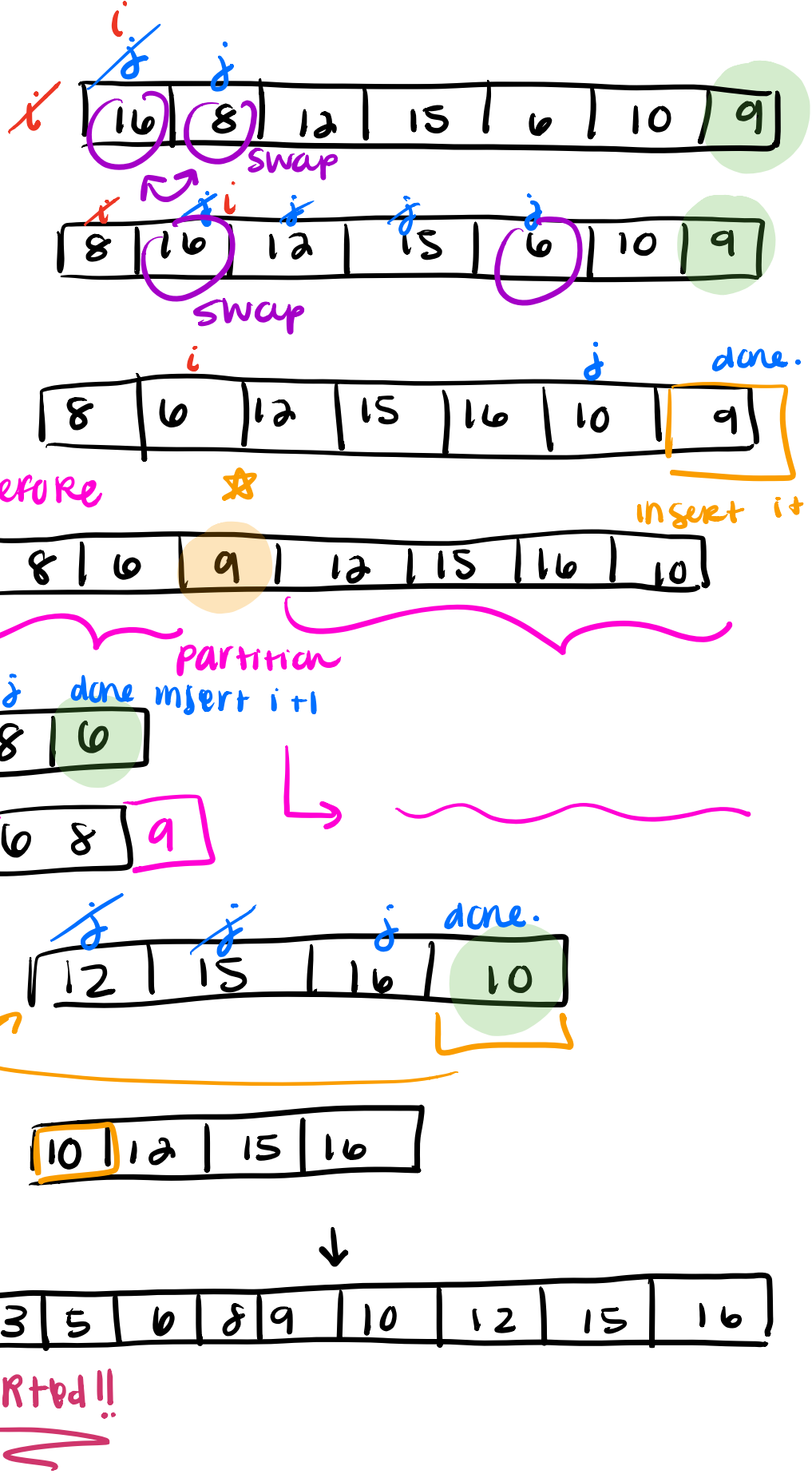
swap i and j



then when you finish, put it at i+1

partition

recursive call on both partitions...


i
j
16 8 12 15 6 10 9
swap
8 16 12 15 6 10 9
swap
i
j
done.
8 6 12 15 16 10 9
FROM before
insert i+1
3 5 8 6 9 12 15 16 10
partition
j done insert i+1
8 6
3 5 6 8 9
i
done.
12 15 16 10
10 12 15 16
3 5 6 8 9 10 12 15 16
SORTed!!

Code for Quick sort:

```
# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
def partition(arr,low,high):
    i = ( low-1 )         # index of smaller element
    pivot = arr[high]     # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if   arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

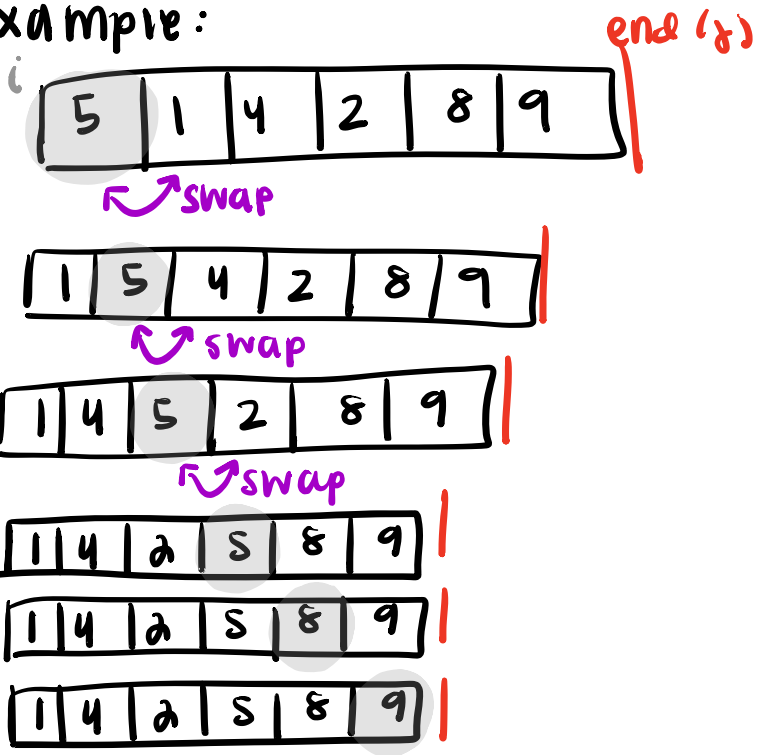
Bubble SORt

A comparison based sorting algorithm
- → iterates left to right comparing every couplet, moving the smaller element to the left
- → Repeats this process until it no longer moves and element to the left

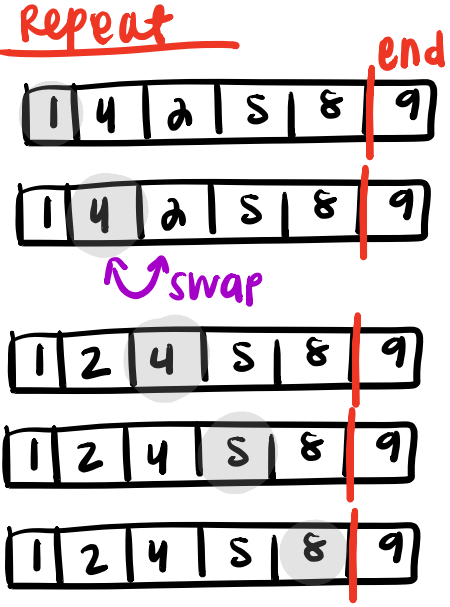
- simple to implement, least efficient
- it moves one space to the right comparing two elements at a time and moving the smaller on to left.

Best case	$O(n)$
Avg. case	$O(n^2)$
Worst case	$O(n^2)$

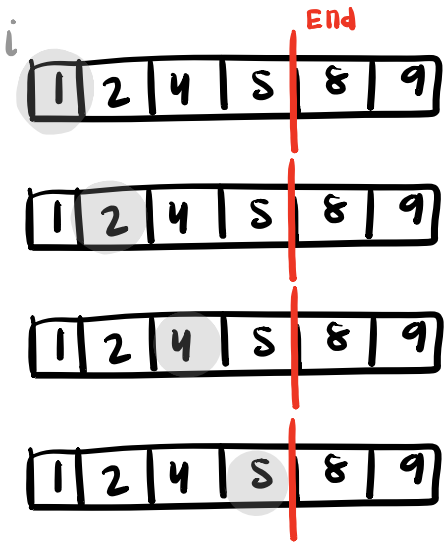
Example:



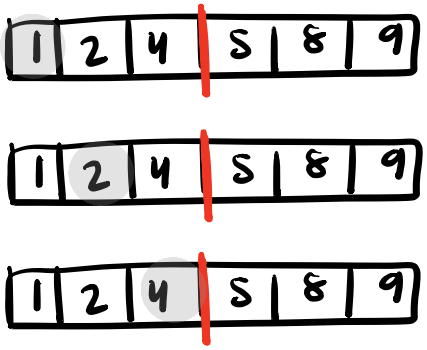
* Repeat



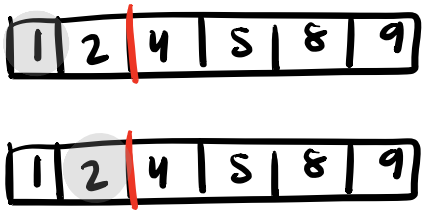
* Repeat



*Repeat:



* Repeat:



done!

Code for bubble sort

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

```
# Python3 Optimized implementation
# of Bubble sort
```

```
# An optimized version of Bubble Sort
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already
        #  in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to
            # n-i-1. Swap if the element
            # found is greater than the
            # next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # IF no two elements were swapped
        # by inner loop, then break
        if swapped == False:
            break
```

Linear time sorts

Counting sort & Radix sort

Counting sort :

We sort n elements, each in the range 0 to K (K fixed)
- sometimes $K = n$
- use element as array index

simple = count the # of items with value i
FOR $0 \leq i \leq K$

Use i as index to array

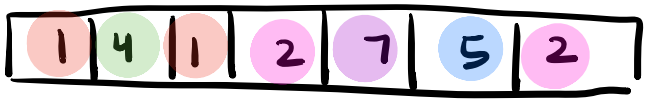
This algorithm is based on keys between a specific range.

→ count the # of objects having distinct key values

→ then calculate the position of each object in the output sequence

Example (with numbers)

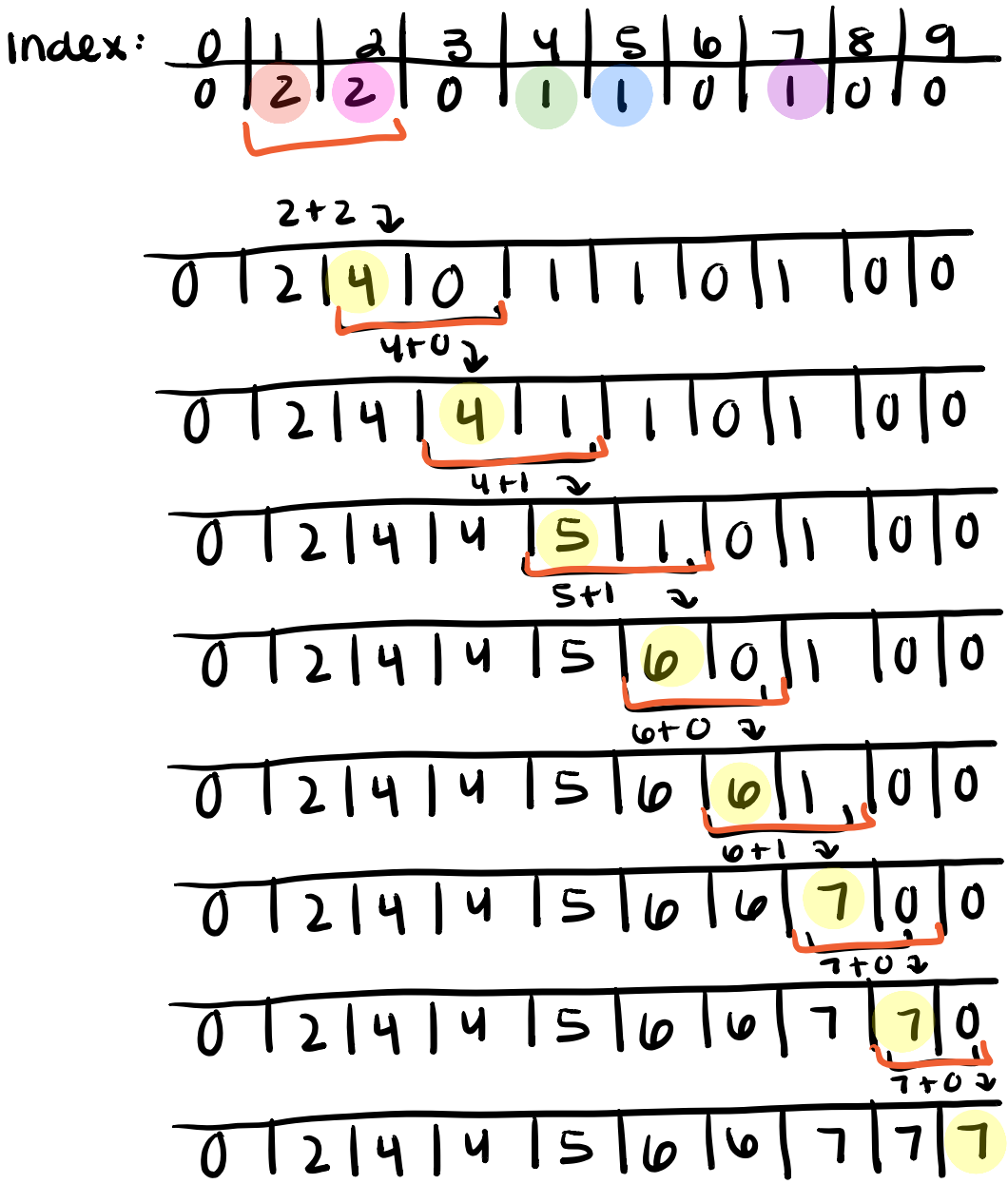
range of 0 to 9 — ARRAY



Create a count array to store the count of each unique object

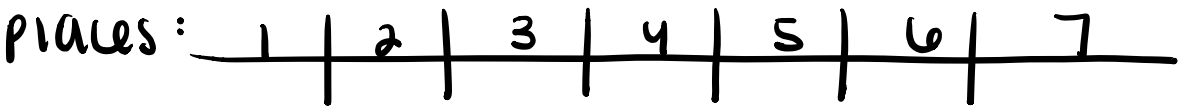
index:	0	1	2	3	4	5	6	7	8	9
	0	2	2	0	1	1	0	1	0	0

Modify count array by adding the previous counts.



Since we have input 7, we create an array with 7 places.

Corresponding values represent the places in the count array



We place the objects in the correct positions and decrease the count by one.

ARRay:

1	4	1	2	7	5	2

Index:

0	1	2	3	4	5	6	7	8	9
0	2 (1 0)	4 (3 2)	4	5 (4 5)	6 (5 5)	6	7 (6 7)	7	7

places:

1	2	3	4	5	6	7
1	1	2	2	4	5	7

look at the array, and we need to place the value 1, then look at index array with the value 2, so put the value 1 in place 2.

then decrease the value at index 2, and move to the next element in the array (4) and go to the 4th index which holds 5 so put 4 in place 5

Repeat for all the values in the array...

1	1	2	2	4	5	7

time complexity: $O(n + k)$

n = # elements

k = range of input

Code for Counting Sort:

```
# Python program for counting sort

# The main function that sort the given string arr[] in
# alphabetical order
def countSort(arr):

    # The output character array that will have sorted arr
    output = [0 for i in range(256)]

    # Create a count array to store count of inidividul
    # characters and initialize count array as 0
    count = [0 for i in range(256)]

    # For storing the resulting answer since the
    # string is immutable
    ans = ["" for _ in arr]

    # Store count of each character
    for i in arr:
        count[ord(i)] += 1

    # Change count[i] so that count[i] now contains actual
    # position of this character in output array
    for i in range(256):
        count[i] += count[i-1]

    # Build the output character array
    for i in range(len(arr)):
        output[count[ord(arr[i])]-1] = arr[i]
        count[ord(arr[i])] -= 1

    # Copy the output array to arr, so that arr now
    # contains sorted characters
    for i in range(len(arr)):
        ans[i] = output[i]
    return ans
```

Radix Sort - only sorts number

sort on last digit (stable sort)

sort on next to last digit, etc

Idea:

- break #s into groups of r bits
- view it as consisting of b/r digits of r bits each
- each digit has value 0 to 2^r
- sort on each digit is $O(n + 2^r)$
- done b/r times

Example:

→ sort numbers from least significant digit to most significant digit

↪ use counting sort in this algorithm

input array:

170	45	75	90	802	24	2	66

first consider the one's place and sort according to only that #, and if there is the same value (170 and 90) the one that was first in the array, remains in front of the other.

170	90	802	2	24	45	75	66

now look at the 10's place

802	2	24	45	66	170	75	90

now look at the 100's place

2	24	45	66	75	90	170	802

now you would look at the 1000 place, but since none of the elements have a value here, you know its sorted!

Lemma 8.3:

Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these #s in $O(d(n+k))$ time if the stable SORT takes $O(n+k)$ times.

→ this makes sense because lets say one digit was 3098, that means you have to go up to the 1000's place, so you do the SORT which is $O(n+k)$ 3 times to get there which is dependent on the last digit placement.

lemma 8.3
Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in O(d(n+k)) time if the stable sort it uses takes O(n+k) time.

Lemma 8.4: go over this!

Given n b-bit numbers and any positive integer $r \leq b$, RADIX-SORT CORRECTLY SORTS these #s in $O(\frac{b}{r})(n+2^r)$ time if the stable SORT it uses takes time $O(n+k)$ time for inputs in the range 0 to k

lemma 8.4
Given n b-bit numbers and any positive integer r<=b, RADIX-SORT correctly sorts these numbers in $O((b/r)(n+2^r))$ time if the stable sort it uses takes time O(n+k) time for inputs in the range 0 to k.

$b = \lg n \lg \lg n$ c) $O\left(\frac{\lg n \lg n}{\lg \lg n}\right)\left(n + 2^{\lg \lg n}\right)$

b) $O\left(\frac{\lg n \lg h}{\lg n}\right)\left(n + 2^{\lg n}\right)$

ط)

```
# Method to do Radix Sort
def radixSort(arr):

    # Find the maximum number to know number of digits
    max1 = max(arr)

    # Do counting sort for every digit. Note that instead
    # of passing digit number, exp is passed. exp is 10^i
    # where i is current digit number
    exp = 1
    while max1/exp > 0:
        countingSort(arr,exp)
        exp *= 10
```

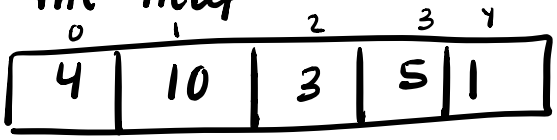
Heap sort

Example :

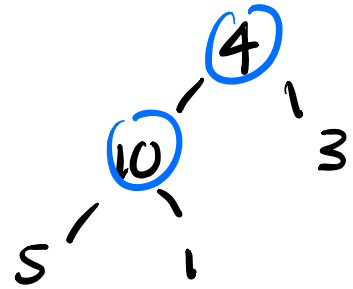
1) First build the heap using the given elements

Create max heap (ascending order)
once created, swap root node with the last node with the last node and delete the last node from the heap.

Input :



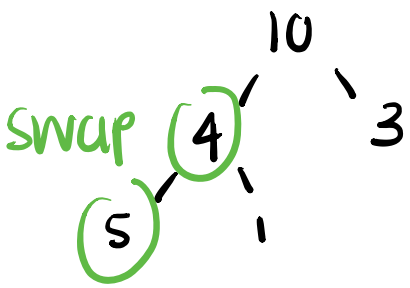
Build Heap

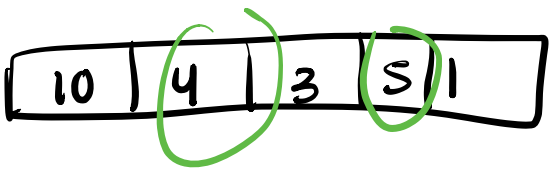


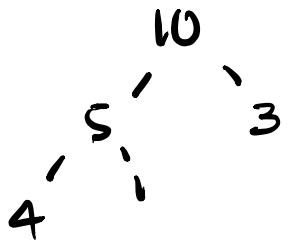
* transform into max-heap

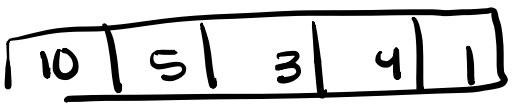
→ parent node should always be greater than or equal to child nodes.

10 greater than 4 so swap

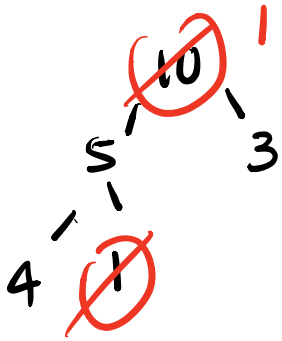


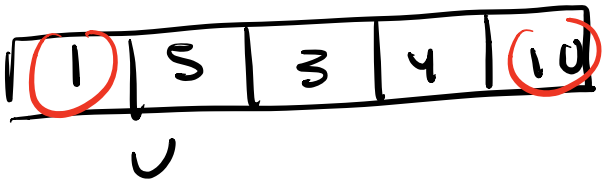




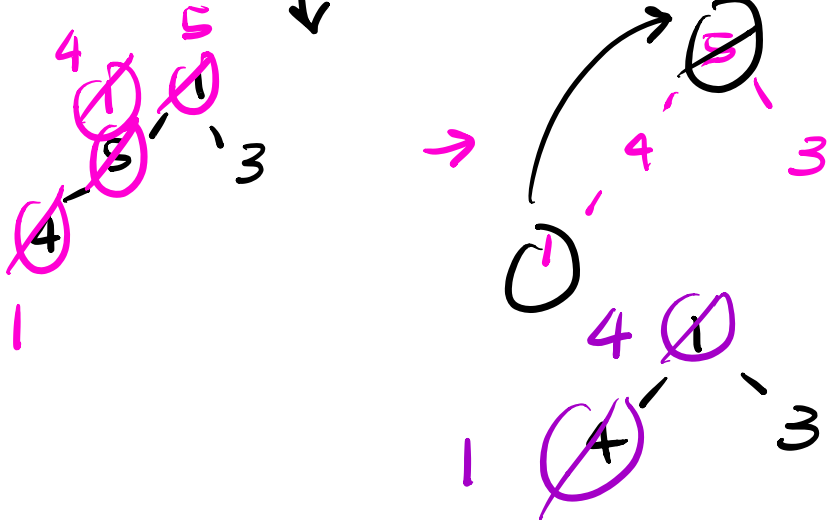


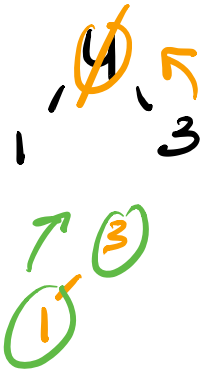
now that we have the max-heap, delete the root and swap with the last element

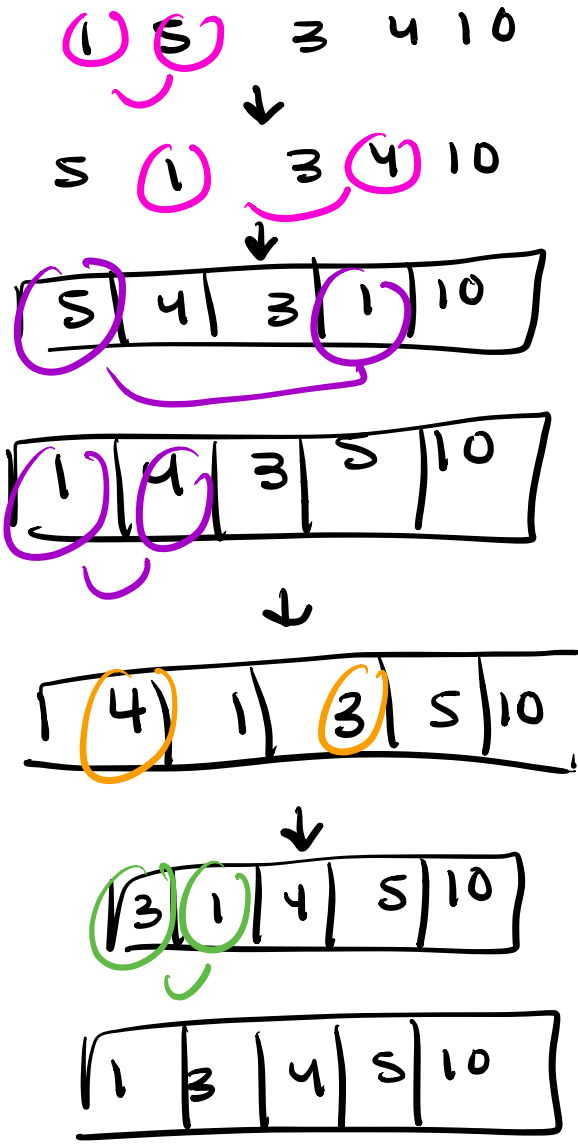

10
1
5
3
4
1


1
5
3
4
10

max-heapify


4
5
1
1
3
4
1
5
4
3
1
4
1
3
1
4


4
1
3
3
1


1 5 3 4 10
5 1 3 4 10
5 4 3 1 10
1 4 3 5 10
4 1 3 5 10
3 1 4 5 10
1 3 4 5 10