1)

Code for quick select:-

def quickselect(arr,low,high,k):

index = partition(arr,low,high)

if(index == k-1):

return arr[index]

if(index > k-1):

low = low

high = index-1

return quickselect(arr,low,index-1,k)

else:

low = index+1

high = high

return quickselect(arr,index+1,high,k)

def partition(arr,low,high):

pivot = arr[high]

i=low-1

for j in range(low,high):

if arr[j]<=pivot:

i+=1

arr[i],arr[j]=arr[j],arr[i]

arr[i+1],arr[high]=arr[high],arr[i+1]

return i+1

arr=[389,45,2,0,32,453,23]

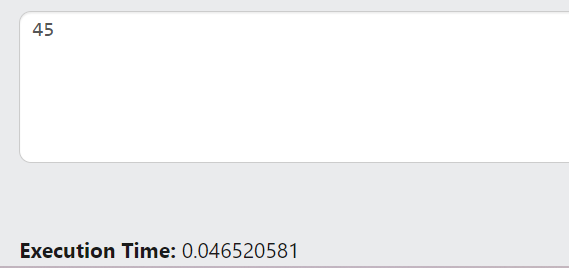
m = quickselect(arr,0,len(arr)-1,5)

print(m)

input 1:-

arr = [389,45,2,0,32,453,23]

k = 5 (5th smallest elements)



Input 2:-

Arr=[1,2,3,4,5,6]

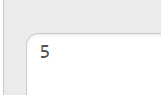
K=5



Input 3:-

Arr=[6,5,4,3,2,1]

K=5



Code for quick sort:-

def quickselect(arr,low,high):

if low<high:

index = partition(arr,low,high)

quickselect(arr,low,index-1)

quickselect(arr,index+1,high)

def partition(arr,low,high):

pivot = arr[high]

i=low-1

for j in range(low,high):

if arr[j]<=pivot:

i+=1

arr[i],arr[j]=arr[j],arr[i]

arr[i+1],arr[high]=arr[high],arr[i+1]

return i+1

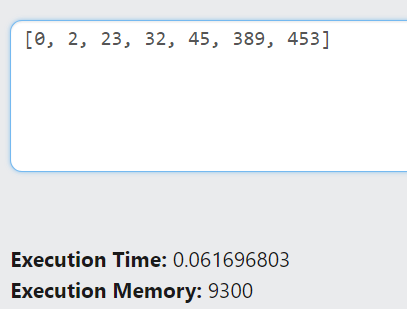
arr=[389,45,2,0,32,453,23]

m = quickselect(arr,0,len(arr)-1)

print(arr)

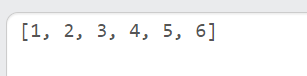
test case-1

input:-389,45,2,0,32,453,23



Test case-2

Input = [6,5,4,3,2,1]



a)

difference between quick sort and quick select:-

purpose:-

quick sort:- it Is mainly used for sorting a given unsorted array it sorts the given list in ascending or descending the order

quick select :- it is mainly used for finding the kth smallest or kth largest element without necessarily need of sorting the entire list

difference in dealing sub arrays:-

quick sort:- in quick sort we recursively split an array into 2 sub arrays based on the index returned by the partition function. And then we sort the left array and right subarray and this continues till the entire array is sorted

quick select :- in quick select we divide the entire array into 2 sub arrays but based on the conditions we either check in left sub array or right sub array or stop the process if the element is found

b)

Time complexity and recurrence relations

Quick sort:-

Best case and average case:- O(NlogN)

Worst case :- O(n^2)

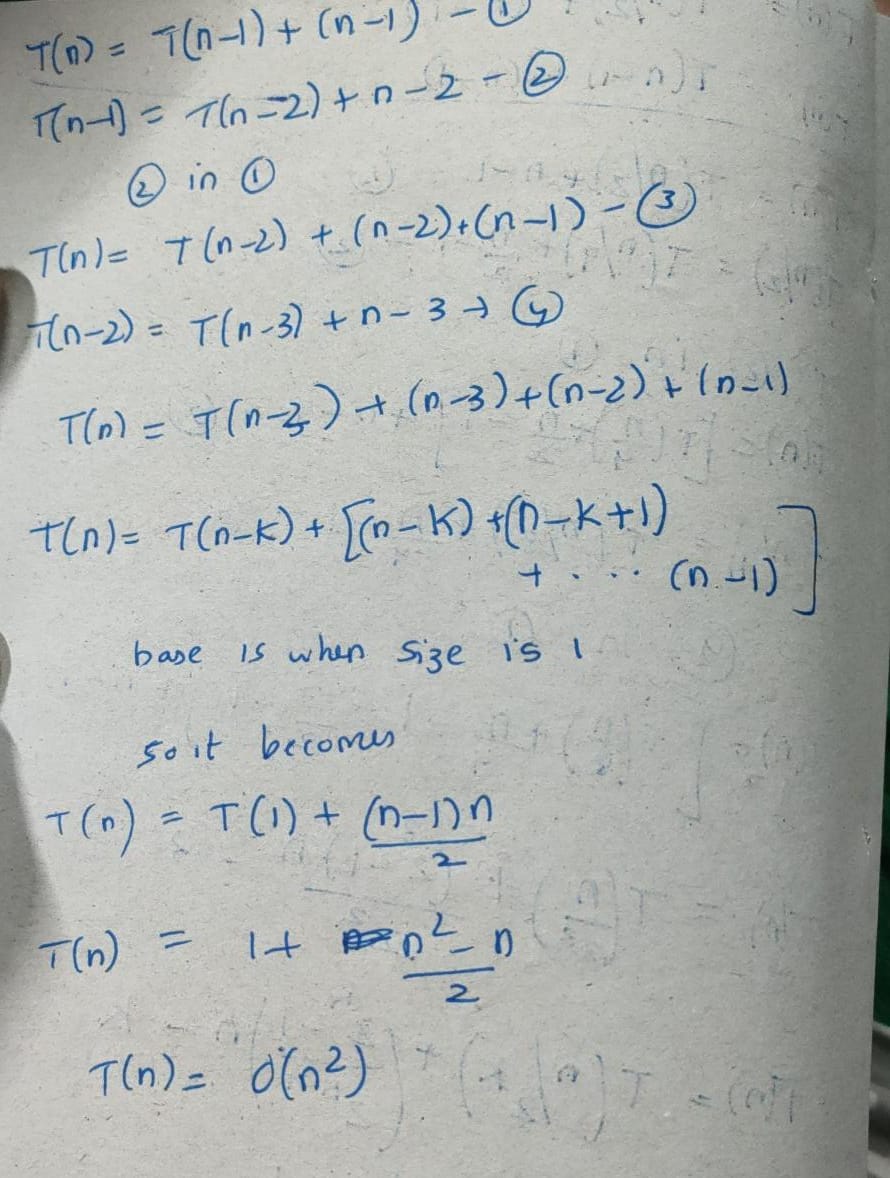
Recurrence relation for the best case scenario:-

T(n) = T(n-1)+(n-1)

Explanation:- the worst case occur when the array is sorted in ascending or descending order. Then if a pivot element is chosen as the first or the last element then to the left or right of the there will be the whole remaining array. So now the array is divided into 1+(n-1) so now next time we need to check n-1 elements in partition function so (n-1) is added

Base case of a quick sort algorithm is when array has just one element.

Solving recurrence relation:-

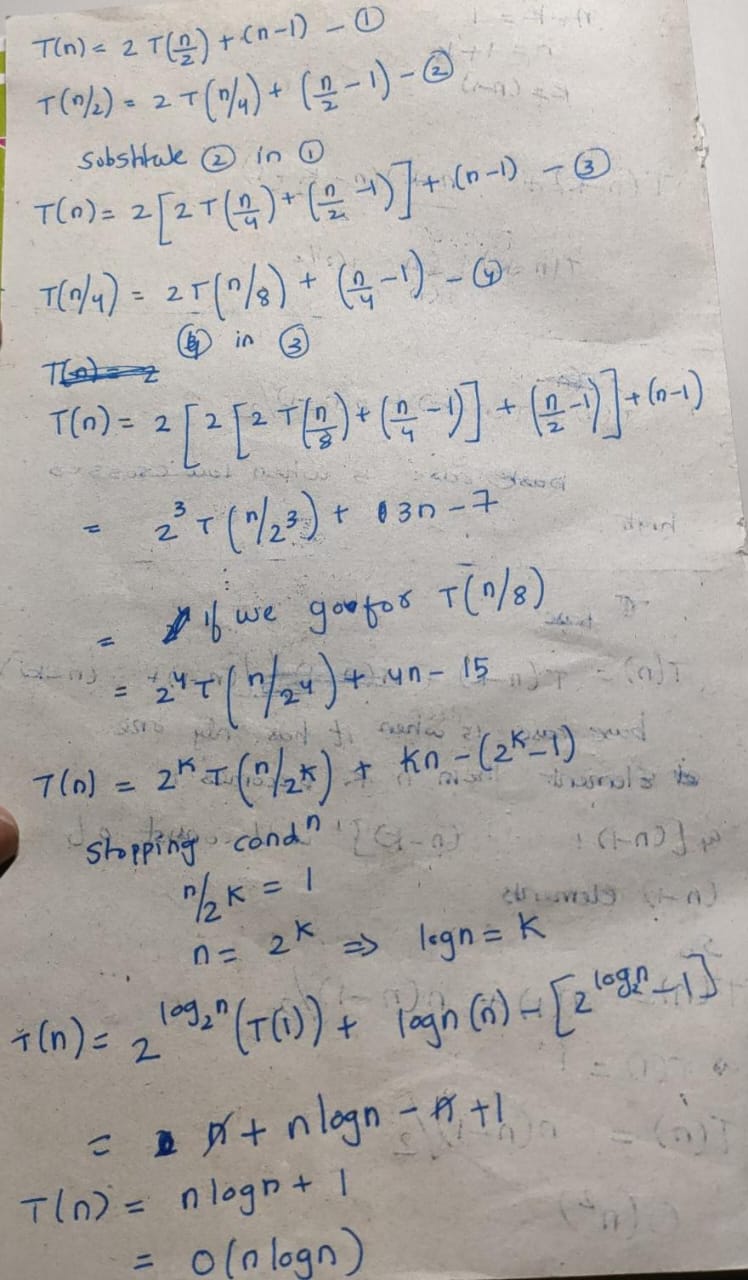


Quick sort recurrence relation for best case or the average case:-

T(n) – 2T(n/2)+(n-1)

So here on an average we split the array into 2 subarrays of n/2 size and we will compare all the elements (or traverse) except the pivot so there is (n-1). Base case of a quick sort algorithm is when array has just one element.

Solving the recurrence relation



Quick select algorithm:-

Best case and average case:- O(n)

Worst case:- O(n^2)

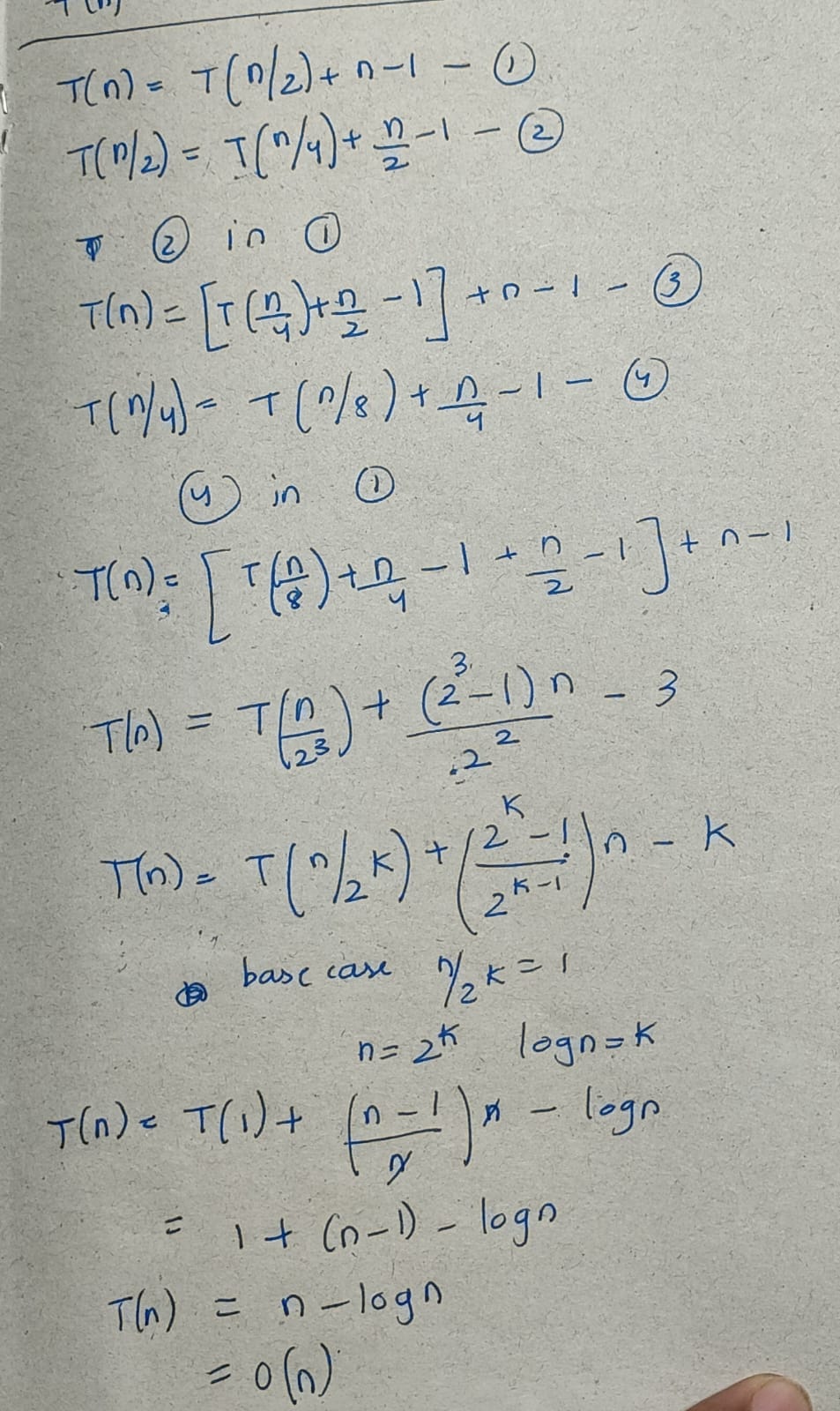
The worst case of quick select is similar to quick sort if we are given an sorted array and asked to find the first minimum element , by choosing the last element as pivot it would take o(n^2) time.

Same in the opposite case given an sorted array choosing first element as pivot and asked to find the (len-1)th smallest element or the largest element gives us the worst case.

Recurrence relation in avg/worst case:-

T(n):- T(n/2) + (n-1)

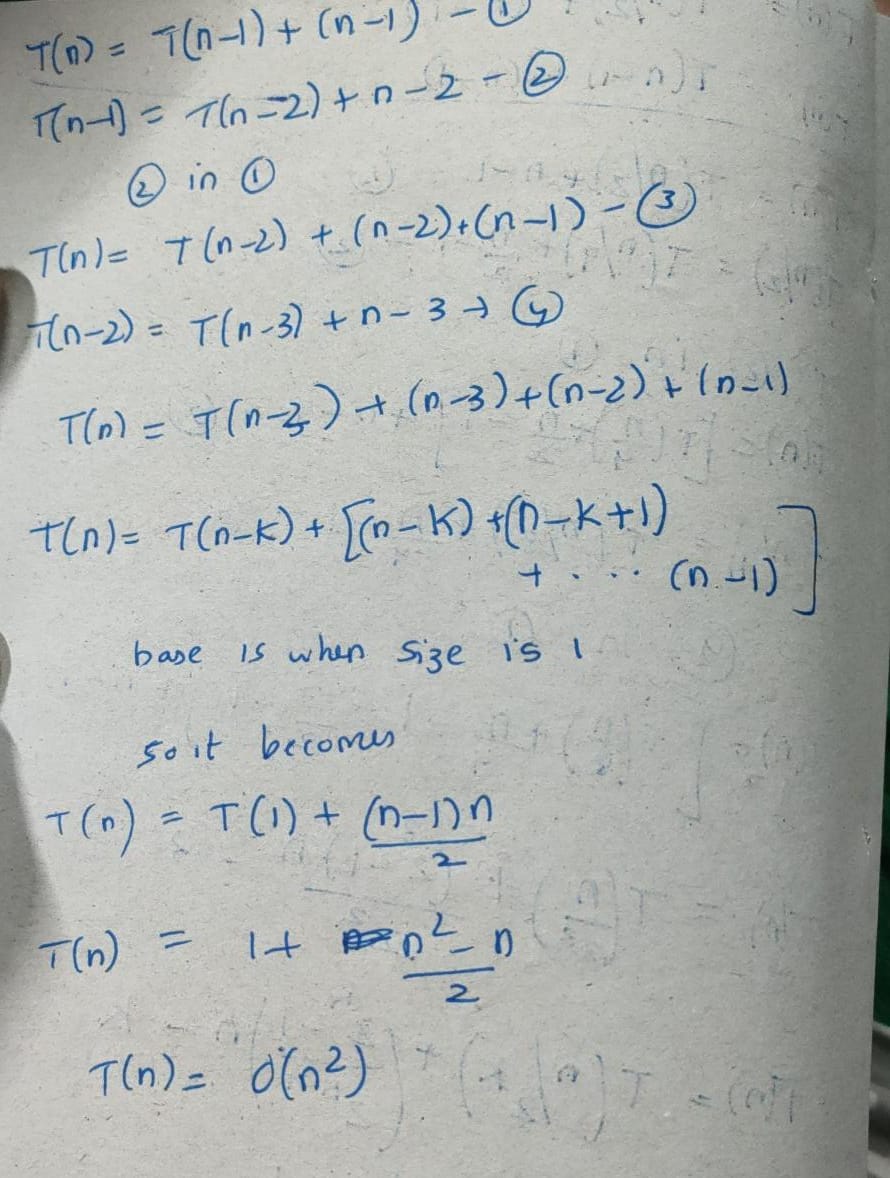
There is no 2 as we only search or select in one of the sub arrays so there is no 2 unlike the quick sort where we sort both the sub arrays



Recurrence relation for worst case:-

T(n)=T(n-1)+(n-1)

Base case is when input size is 1



The worst case recurrence relations are same for both the algorithms

c)

use cases of quick sort:-

1. Quick sort pivot selection process and partitioning process is used In many famous algorithms like quick select
2. It is used in various visualization techniques to correctly arrange and display the data points
3. It is majorly used in file systems to sort the files based on size making it easier to access files.
4. Same even e-commerce website use it sort them by ratings , price ,popularity etc
5. Mainly used in DBMS used where we need larger datasets
6. Basically in use cases where worst cases are a rarity quick sort is widely used as it doesnot occupy extra memory as it is a in-place sort

Use cases of quick select:-

1. It is used in median finding. We can do it even by sorting the entire array but it takes O(nlogn) time but using this approach averagely it takes O(n) time.
2. It is mainly used in order statistics where we need to search the kth highest selling product. And It is also said that it is used in finding the kth nearest neighbours in ML algorithms
3. it is used in image processing finding the darkest or brightest pixel or the kth brightest pixel
4. it is also used in data base queries where we usually find the kth max or the kth min
5. can be used in graph algorithms to find the kth shortest path from one node to other node

d) what are the different pivot selection methods

1) choosing the first element or last element as pivot – but these choosing may lead us to worst case scenarios for example in cases where a sorted array or a mostly sorted array is given to us , if we use this pivot selection these may consume O(n^2) time. This is same scenario even for the quick select algorithm.

2)choosing any random element from the array – this selection will reduce the chance of getting the worst case considerably.

3) choosing median of three is an another approach where we select the median value of the first , middle and last element of the array , By choosing the median of three elements (the first, middle, and last elements), you increase the chances of selecting a pivot that roughly divides the data into two relatively equal partitions. This helps avoid worst-case scenarios and significantly improves quick sort performance on nearly sorted data.

The same pivot selection methods can be used for quick select algorithm as the same partition function is used for both the algorithms.

Selecting first element as pivot:-

Algorithm/code

def partition\_first\_ele(arr, low, high):

pivot = arr[low]

left = low + 1

right = high

while True:

while left <= right and arr[left] < pivot:

left += 1

while arr[right] >= pivot and right >= left:

right -= 1

if right < left:

break

else:

arr[left], arr[right] = arr[right], arr[left]

arr[low], arr[right] = arr[right], arr[low]

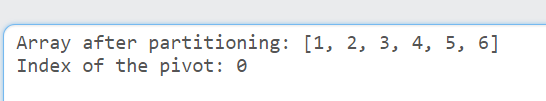
return right

arr = [1,2,3,4,5,6]

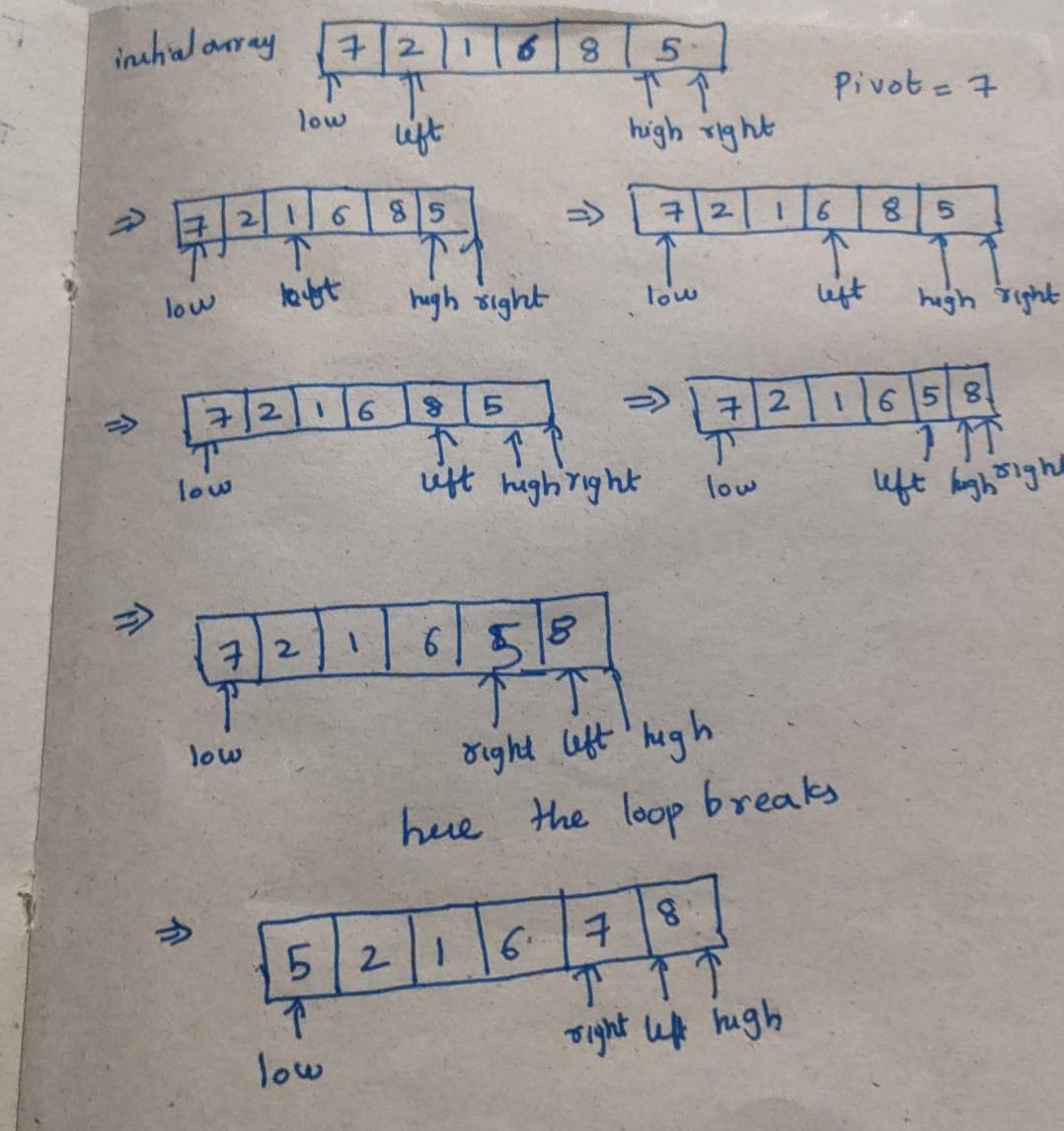
pivot\_index = partition\_first\_ele(arr, 0, len(arr) - 1)

print("Array after partitioning:", arr)

print("Index of the pivot:", pivot\_index)



If we use this pivot selection strategy then if the array is sorted or reversely sorted then this results in worst case of quick sort or select so we use more partitioning methods to reduce the chances of the worst case.



Second method : -

Choosing a random pivot

so here what we do is basically choose a pivot and then swap it with either the last element or the first element in the above code I swapped it with the first element. This considerably reduces the chance of getting the worst case. The difference cannot be properly observed for an array of smaller size but may be impactful for a array of a medium or larger size.

Code

import random

def partition\_median\_of\_three\_swap\_last(arr, low, high):

pivot\_index = random.randint(low, high)

arr[low], arr[pivot\_index] = arr[pivot\_index], arr[low]

pivot = arr[low]

left = low + 1

right = high

while True:

while left <= right and arr[left] < pivot:

left += 1

while arr[right] >= pivot and right >= left:

right -= 1

if right < left:

break

else:

arr[left], arr[right] = arr[right], arr[left]

arr[low], arr[right] = arr[right], arr[low]

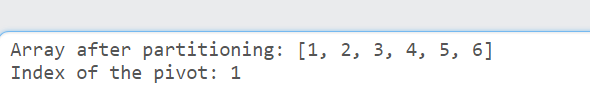
return right

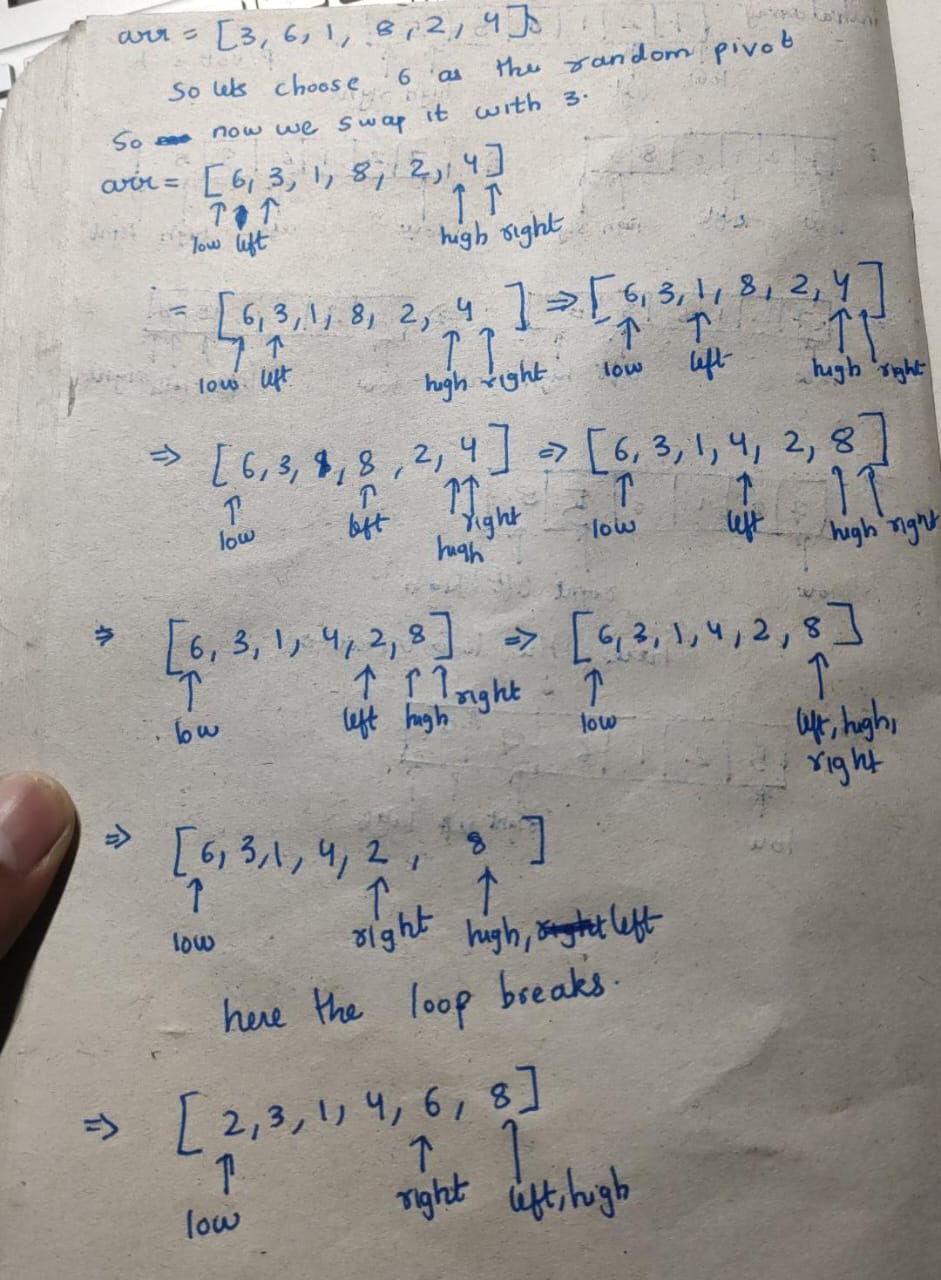
arr = [1,2,3,4,5,6]

pivot\_index = partition\_median\_of\_three\_swap\_last(arr, 0, len(arr) - 1)

print("Array after partitioning:", arr)

print("Index of the pivot (median of three, swapped with last element):", pivot\_index)



Here the partition is better than that of the 1st case where there were no elements to the left side of the array. The left sub array is 1 and right is 3,4,5,6 but this is also not an near perfect partiion we can achieve it by our next partition method median of three

3rd pivot selection method

median of 3:-

in this we choose the median of high , low and mid value of the array and in the code for a difference swapped it with the element at the last position

code:-

def partition\_median\_of\_three\_swap\_last(arr, low, high):

mid = (low + high) // 2

l= [[arr[low], low], [arr[mid], mid], [arr[high], high]]

l.sort()

median\_value, median\_index = l[1]

arr[high], arr[median\_index] = arr[median\_index], arr[high]

pivot = arr[high]

i=low-1

for j in range(low,high):

if arr[j]<=pivot:

i+=1

arr[i],arr[j]=arr[j],arr[i]

arr[i+1],arr[high]=arr[high],arr[i+1]

return i+1

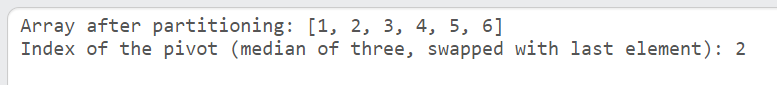
arr = [1,2,3,4,5,6]

pivot\_index = partition\_median\_of\_three\_swap\_last(arr, 0, len(arr) - 1)

print("Array after partitioning:", arr)

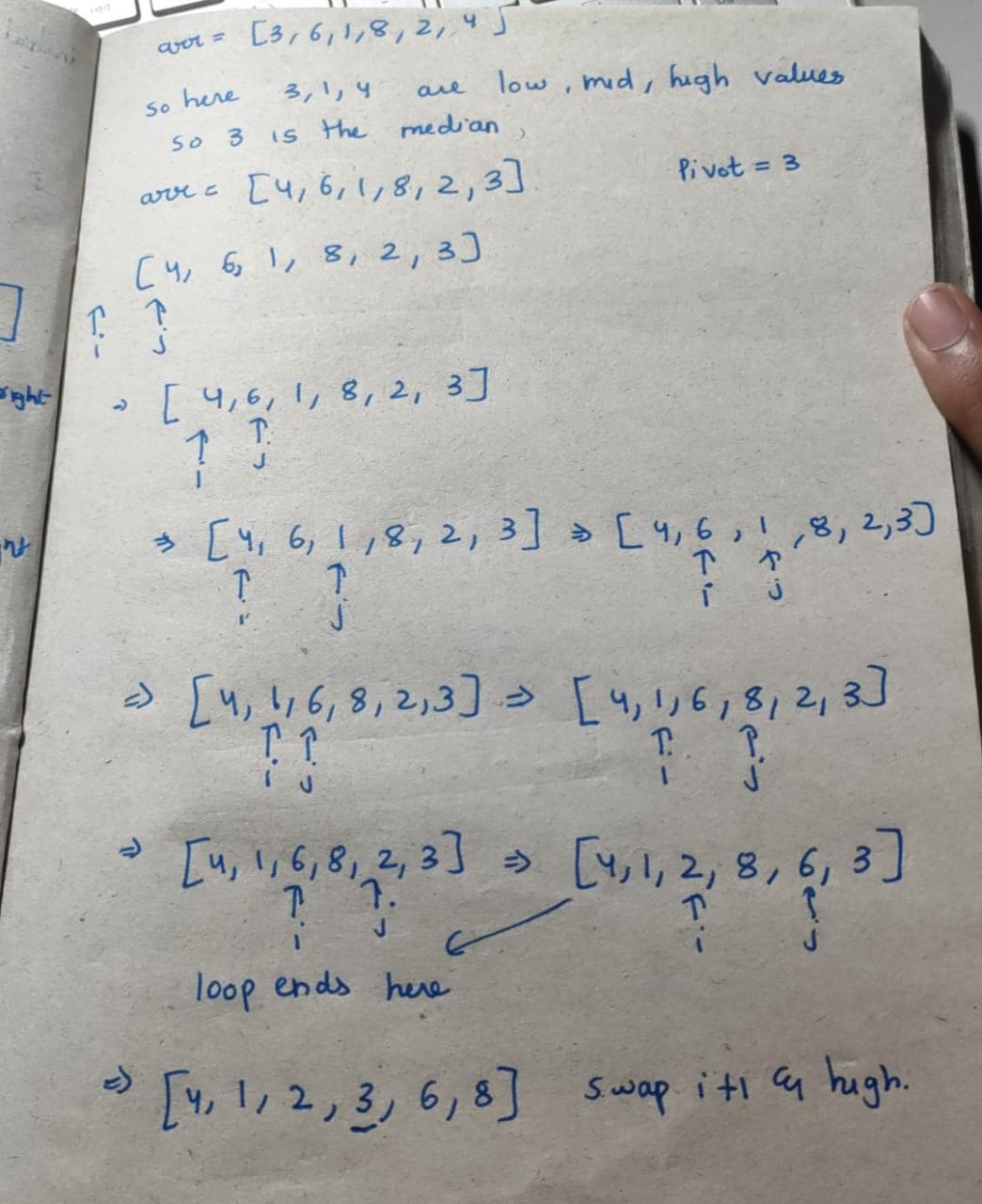
print("Index of the pivot (median of three, swapped with last element):", pivot\_index)

pivot selected is 3



Here the partition is almost the near perfect partition

One more example:-



So in this method most of the times ensures that the there is a relatively balanced split of the arrays or we get arrays relatively of equal portion as we can see in the above example the sub array sizes are 3 and 2 respectively.

e)

Algorithm A takes 10.24 micro seconds to sort the entire unsorted array with 2^10 that is 1024 elements.

Algorithm B takes 1.024 microseconds to find the ith smallest or ith largest element in an list of 1024 unsorted elements. So in a list of 1024 elements to find k smallest elements we need 1.024 elements.

So acc to this till k=10 means finding the first 10 smallest elements our quick select works better but when we ask the algorithm to find 11 , 12 or the first 100 smallest elements quick sort is better in that case.

Generalized case:-

So we generalize the case for n elements quick sort(algorithm A) takes (10.24)\*(n/1024) or n/100 mirco seconds.

If we genralize it for algorithm B to find ith smallest element it takes (1.024)\*(n/1024) or n/1000 micro seconds. So for finding k smallest/largest elements it takes k\*n/1000.

So if solve the equation

n/100 > = k\*n/1000

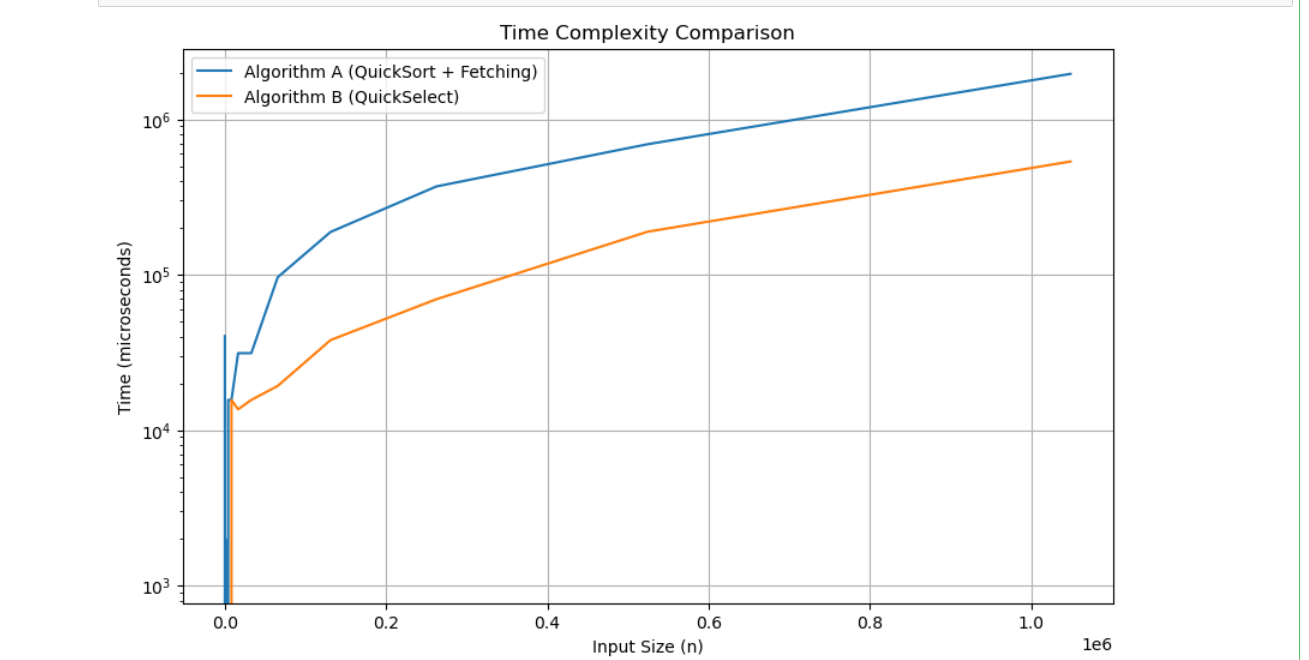
we get that for k<=10 (in the case for 1024 elements it takes 10.24 secs to sort) for any n quick select is faster and for k>10 for any n quick sort(if fetching time is ignored) is faster.

What algorithm, A or B, must be chosen if n = 2^20 and k = 2^9 ≡ 512?

So of input size is n = 2^20 then the algorithm A that is the quick sort algorithm takes n/100 that is 2^20/100 that is around 10496.96 micro seconds

For algorithm B the execution time is (2^9)\*(2^20)/1000 takes 536,870.912 micro seconds

So for this case algorithm A is better.



Plot of the time taken by both algorithm (quicks sort to sort the array and access the ith smallest element which takes unit and quick select algorithm which finds ith smallest element) . so as the input sizes increases the gap between both the algorithms increase so we can prefer quick select over quick sort for selecting ith smallest element.