Name - Yogesh Agarwala

EE19B130

Assigment - 3

① Show that the running time of the merge-sort algorithm on n-element sequenc is $O(n \log n)$, even when n is not a power of 2

Algorithm :

ⓐ Input : Array $A[1 \ldots n]$

30	10	18	3	2	16	50

ⓑ Divide into subarrays $A[1 \ldots m]$ and $A[m+1, \ldots n]$ where $m = \lceil n/2 \rceil$

30	10	18	3

2	16	50

ⓒ Recursively Mergesort $A[1 \ldots m]$ and $A[m+1, \ldots n]$

3	10	18	30

2	16	50

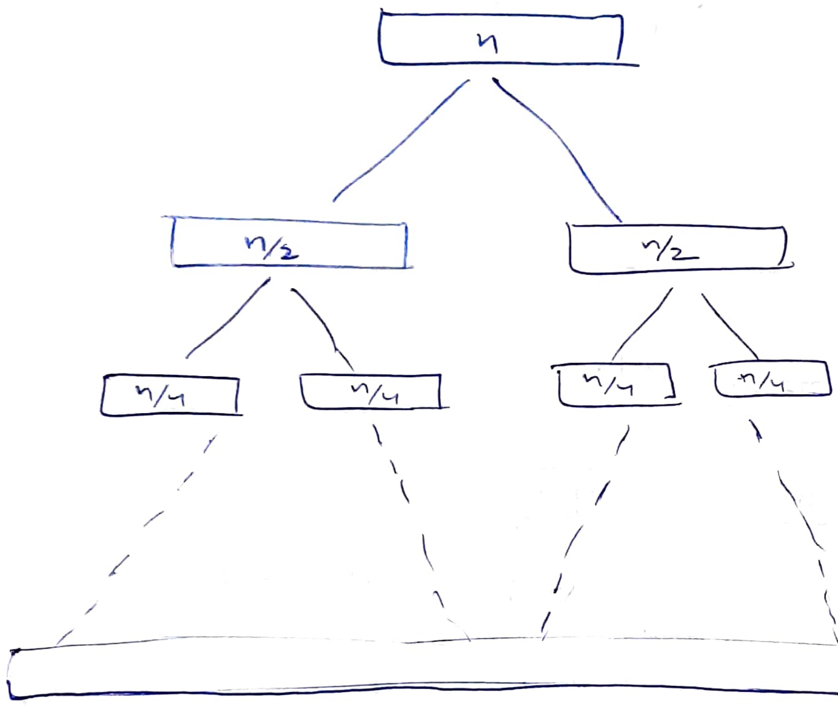
ⓓ Merge the sorted arrays

2	3	10	16	18	30	50

Time- complexity: (valid for both case when n is power of 2 or not)

$$T(n) = \underbrace{T(\lfloor n/2 \rfloor)}_{\text{Sorting left - part}} + \underbrace{T(\lceil n/2 \rceil)}_{\text{sorting right - part}} + \underbrace{cn}_{\text{merging}}$$

When n is a power of 2:



Level 0:

Level 1: No. of operation for merging = cn

Level 2: no. of operation for merging = cn

Level $\log_2 n$: No. of operation for merging = cn

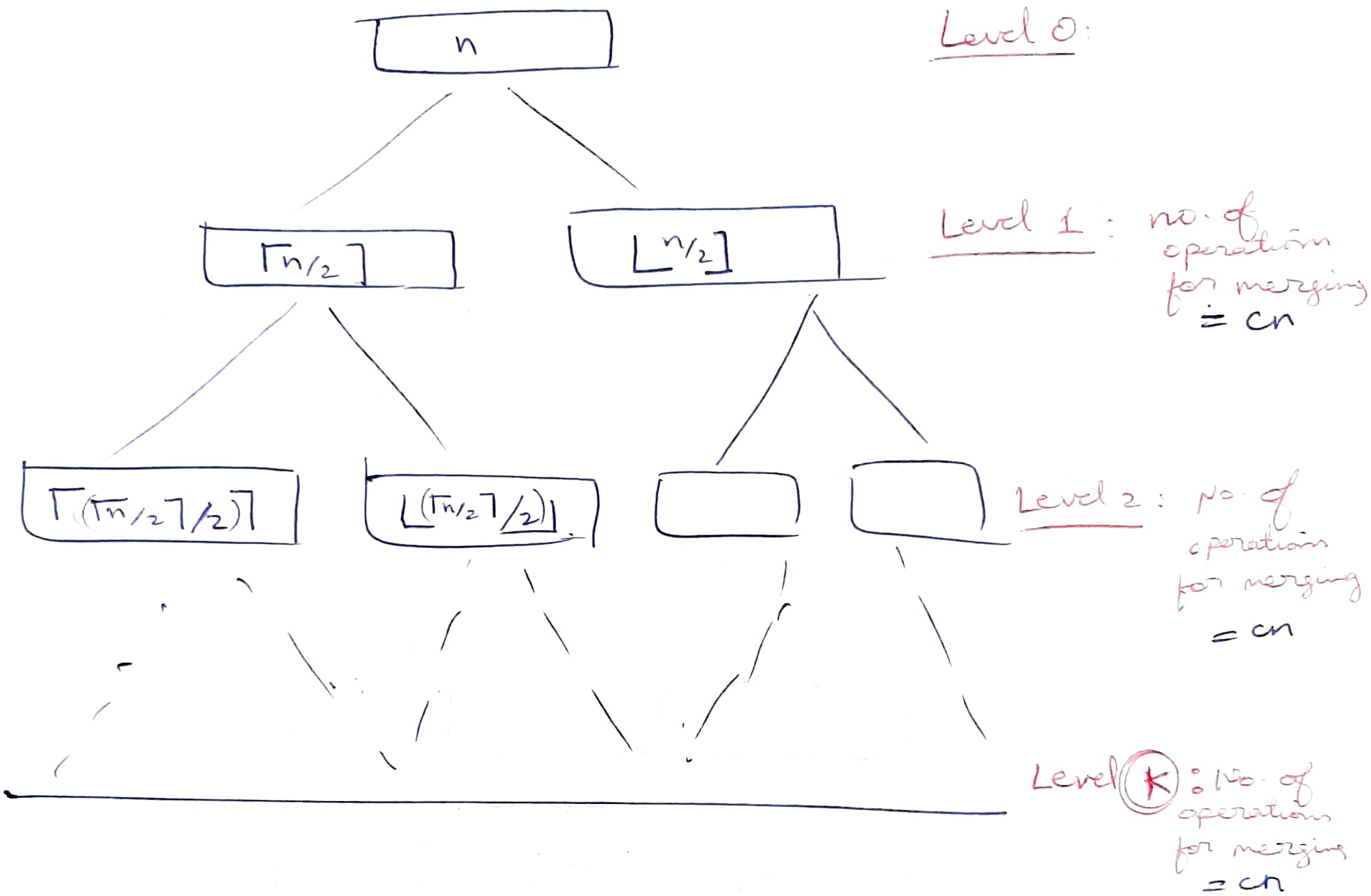
So no. of levels $= \log_2 n$

Work at each level (i.e no. of operations for merging the sorted arrays) $= cn$

$\therefore$ Total work $= (cn)(\log_2 n)$

$= O(n \log n)$

when n is not a power of 2 :



Here no. of levels $(k) = \lceil \log_2 n \rceil$

Work at each level is same as perevious case $= cn$

(Since $\lceil n/2 \rceil + \lfloor n/2 \rfloor = n$)

$\therefore$ Total work $= (cn)(\lceil \log_2 n \rceil)$

$= O(n \log_2 n)$

(2) Consider a modification of the deterministic version of the quick-sort algorithm where we choose the element at index $\lfloor n/2 \rfloor$ as our pivot. Describe the kind of sequence that would cause this version of quick-sort to run in $\Omega(n^2)$ time.

Solution:

If this quicksort is ran on a array, where on every recursive call, the pivot $\lfloor n/2 \rfloor$ is the largest element of its subarray, then the number of comparisons will be

$$(n) + (n-1) + (n-2) + \cdots + 1$$

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

$$= \Omega(n^2)$$

Example:

e.g.

| 1 | 3 | 5 | $\boxed{6}$ | 4 | 2 |

index $= \lfloor 6/2 \rfloor$
$= 3$

no. of comparison $= n$

| 1 | 3 | $\boxed{5}$ | 4 | 2 | $\boxed{6}$

index $= \lfloor 5/2 \rfloor$
$= 2$

no. of comparison $= n-1$

| 1 | 3 | $\boxed{4}$ | 2 | $\boxed{5}$ $\boxed{6}$

index $= \lfloor 4/2 \rfloor$
$= 2$

no. of comparison $= n-2$

⋮

we can see on each recursive call, the pivot $\lfloor n/2 \rfloor$ turns out to be the largest element and thus no. of comparison $= (n) + (n-1) + \cdots$
$= \Omega(n^2)$

3. Describe and analyze an efficient method for removing all duplicates from a collection A of n-elements.

- **Step 1: Time = $O(n \log n)$**

  Sort the array using mergesort/quicksort

  Step 2: Time = $O(n)$

  Traverse the array to find the duplicates, and remove them.

- **Time complexity** = mergesort + linear traversal of the sorted array

  $= O(n \log n) + O(n)$

  $= O(n \log n)$

- **Python Implementation** → Next Page

Python Implementation:

In [1]:
```
"""Part 1: Sorting the array using mergesort"""
def merge_sort(A):
    n = len(A)

    # base-case of the recurssion
    if n==1:
        return A

    # sort the left and right halves of the array recursively
    mid = n//2
    L = merge_sort(A[:mid])
    R = merge_sort(A[mid:])
    merged_array = merge(L,R)
    return merged_array

def merge(L,R):
    i = 0
    j = 0
    answer = []

    # comparing the elements of the left and right sorted arrays,
and adding them to a new array (merged_array)
    while i<len(L) and j<len(R):
        if L[i]<=R[j]:
            answer.append(L[i])
            i += 1
        else:
            answer.append(R[j])
            j += 1

    # while comparing the left and right sorted arrays, it may ha
ppen that one of them is completly added to the merged_array
    # but in the other array still there are some elements left,
& since those are already sorted, add them to the merged_array
    if i<len(L):
            answer.extend(L[i:])
    if j<len(R):
            answer.extend(R[j:])

    # return the merged_array
    return answer
```

In [2]:
```
"""Part 2: Removing duplicates from the sorted array"""
def remove_duplicates (sorted_array):
    arr = []
    i=0
    j=0
    n=len(sorted_array)

    while (i<len(sorted_array)):
        # copying elements to the output array from the sorted ar
ray
        arr.append(sorted_array[i])

        # don't copy the element to the output_array if it is a d
uplicate of the previous element
        # in while loop the condition i!=len(sorted_array)-1 is a
dded, to make sure
        # that sorted_array[i]==sorted_array[i+1] is not checked
for the last element
        while (i!=len(sorted_array)-1 and sorted_array[i] == sorte
d_array[i+1]):
            i+=1

        i+=1

    # sorted list with no duplicates
    return arr
```

In [3]:
```
"""main function"""
input = [2,3,1,3,6,2,1,3]
sorted_array = merge_sort(input)
output = remove_duplicates(sorted_array)
print("Array without duplicates = ",output)
```

```
Array without duplicates =  [1, 2, 3, 6]
```

(4) Given an array A of n integers in the range $[0, n^2-1]$ describe a simple method for sorting A in $O(n)$ time.

- Comparison based sorting algorithms like Merge sort, quicksort, heapsort cannot do better than $n \log n$

- Counting sort is a linear time sorting algorithm but in this case it would be $O(n^2)$

Since →

when an array of n integers, with each integer in the range from 1 to K, is sorted using counting sort, it takes

$O(n+K)$

So if elements range is from 1 to n^2, then it will take $O(n+n^2) = O(n^2)$ which is even worse than comparison based sorting

- Radix sort can solve the above problem in $O(n)$ if implemented properly.

i.e if radix sort is applied after representing all the elements of array, in (base n)

(explained after 2 page)

e.g if $n=100$ and one of the element $=(749)_{10}$ in decimal system (base 10) then convert the number to base 100, similarly convert all other elements of the array to base 100, before using radix sort.

Radix Sort:

1. First sort the elements based on the last digit (the least significant digit)

2. These results are again sorted by 2nd last digit (the digit next to least significant)

3. Continue this process for all digits until we reach the most significant digit

Sort must be stable:
that is, if the digits compared are same, then preserve the previous order

e.g if 2703
& 2313
& 1702
are compared then
1702
2703
2313

counting sort can be used for this purpose (as it is a stable sort)

NOTE: Radix sort uses count sort as a subroutine to sort.

Running time

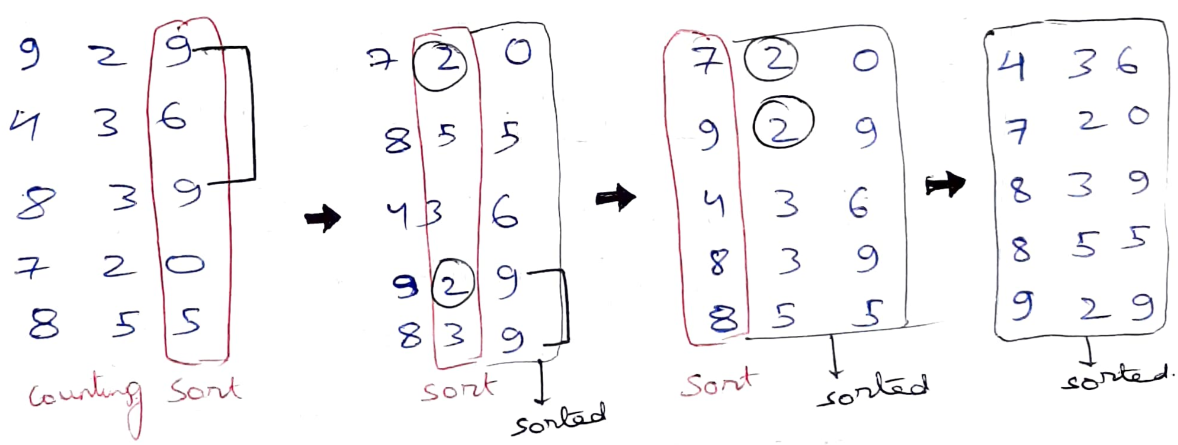
Let there be max. d digits in the input integers

$\Rightarrow d = \log_b k$ where

b = base for representing number
e.g decimal system $b = 10$
binary system $b = 2$

k = maximum of possible value of the input integers

each digit $\in \{0, 1, 2, \ldots, b-1\}$



Use counting sort digit-sort

$\Rightarrow \Theta(n+b)$ per digit

$\Rightarrow$ Total time $= \Theta((n+b)d)$

$= \Theta((n+b)\log_b k)$

$= \Theta(n \log_n k)$ minimized when $b = n$

$= \Theta(nc)$ if $k \le n^c$ where c is a constant

Note :

Radix sort running time $= O((n+b)d)$

$= O((n+b)\log_b(k))$

we can sort an array of integers, with a range from 1 to n^c, if the numbers are represented in base n.

So in our question the range is $[0, n^2-1]$

So if we represent all the n integers in base n, then at the max we will have a 2-digit number. So using only 2 calls to counting sort, we can solve the problem. Since there are only 2 calls

$\therefore$ Running time $= O(\boxed{(n+b)}\,\boxed{d})$

max 2-digit if base $= n$ & $K = n^2$

counting sort of each digit.

$= O((n+n)2)$

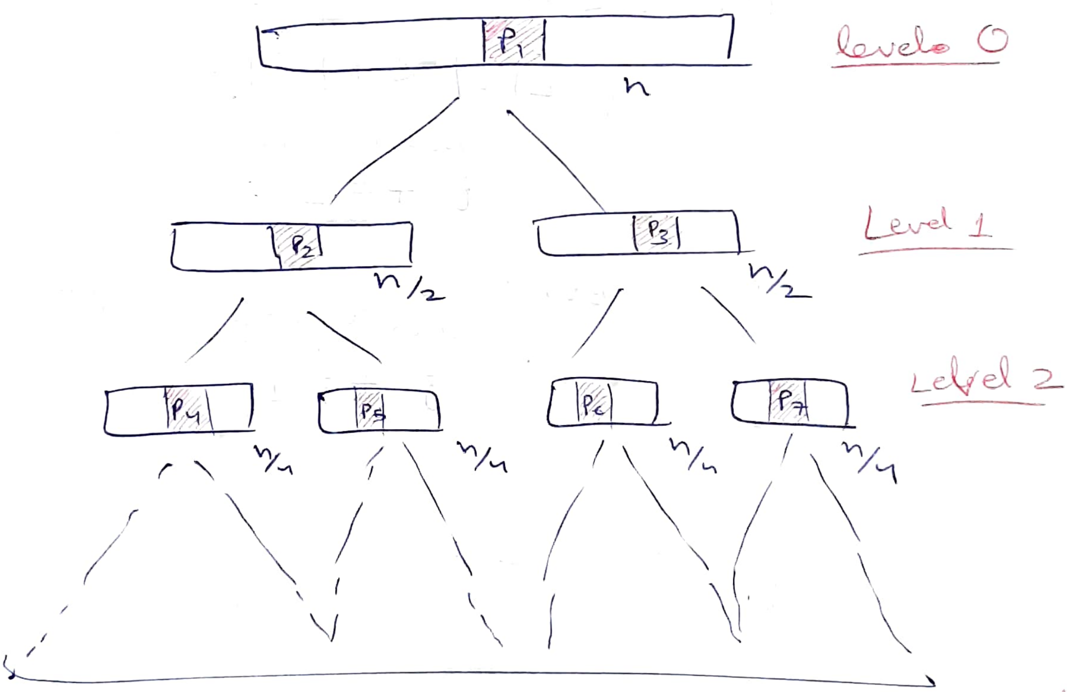
$= O(4n)$

$\approx O(n)$

5. Show that the quicksort's best-case running time is $\Omega(n \log n)$

Best-case

Suppose we run Quicksort on some array, then the best-case occurs, when on each recursive call the pivot choosen is ~~to~~ equal to or close to the median element of its subarray.



No. of levels $\geq \log_2 n$ (equality holds when pivot is equal to median element of each subarray)

Work on each level (i.e rearranging array about the pivot) $= cn$ (if pivot is close to median element)

$\therefore$ Time-complexity (total-no. of comparisons) $\geq (cn)\log_2 n$

$= \Omega(n \log n)$

Lower-bound since this is the best-case