Assignment - 3

Topic: Sorting

Nikilesh.B
EE17B112

1) Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

From the merge sort algorithm , we can see that:

```
sort( array of N elements )

will call:

sort( array of N/2 elements );  // Sort left  HALF of array
sort( array of N/2 elements );  // Sort right HALF of array

merge( array of N elements );   // Takes N statements
```

Therefore:

Amount of work done by sort(array of size N) =
Amount of work done by sort(array of size N)
+ Amount of work done by sort(array of size N) + N

Or: $T(N) = T(N/2) + T(N/2) + N$

Furthermore: $T(1) = 1$ // Time to execute the if-statement

The performance of merge sort is then given by this recurrence relation:

$T(N) = 2*T(N/2) + N$ ..... (1)
$T(1) = 1$ ..... (2)

Solving the *recurrence relation* of Merge Sort analysis

- We can use the expansion method to find a solution:

T(N) = 2*T(N/2) + N (T(N/2) = 2*T(N/4) + N/2)

= 2*(2*T(N/4) + N/2) + N

= 4*T(N/4) + N + N

= 4*T(N/4) + 2*N

T(N) = 4*T(N/4) + 2*N (T(N/4) = 2*T(N/8) + N/4)

= 4*(2*T(N/8) + N/4) + 2*N

= 8*T(N/8) + N + 2*N

= 8*T(N/8) + 3*N

T(N) = 8*T(N/8) + 3*N (T(N/8) = 2*T(N/16) + N/8)

= 8*(2*T(N/16) + N/8) + 3*N

= 16*T(N/16) + N + 3*N

= 16*T(N/16) + 4*N

And so on....

T(N) = 2*T(N/2) + N

= 4*T(N/4) + 2*N = 2^2*T(N/(2^2)) + 2*N

= 8*T(N/8) + 3*N = 2^3*T(N/(2^3)) + 3*N

In general:

T(N) = 2^k*T(N/(2^k)) + k*N

And: T(1) = 1 (from Equation (2))

For *simplicity*, let us assume that *N* is equal to *some* power of 2

For example: If N = 1024 (2^{10})

T(N) = 2*T(N/2) + N

= 4*T(N/4) + 2*N

= 8*T(N/8) + 3*N

= 16*T(N/16) + 4*N

= 32*T(N/32) + 5* = ...

= 1024*T(N/1024) + 10*N (N = 1024)

= N*T(1) + 10*N (T(1) = 1 !)

= N + 10*N

= 11*N

In general, if $N = 2^k$, we can solve T(N) *exactly* as follows:

If $N = 2^k$

T(N) =2*T(N/2) + N

= 4*T(N/4) + 2*N

= 8*T(N/8) + 3*N

= 16*T(N/16) + 4*N

= 32*T(N/32) + 5*N = ...

= (2^k)*T(N/(2^k)) + k*N ($N = 2^k$, so: N/(2^k) = 1)

= (2^k)*T(1) + k*N (2^k= N)

= N*T(1) + k*N (T(1) = 1)

= N + k*N

= (k+1)*N ($N = 2^k \Rightarrow k = \lg(N)$)

= (lg(N)+1)*N

= (log(N)+1)*lg(N)

If $N \neq 2^k$, we will use an *upper bound*:

If $N \neq 2^k$, then:

Sorting an array of N takes at most the amount of time to sort an array of 2^k
(because this array has *more* elements than the one you are sorting !!!!)

Since sorting an array of size $N = 2^k$ will take:

running time of merge sort = (log(N)+1)*N

then sorting an array of size $N < 2^k$ will take at most:

running time of merge sort ≤ (log(N)+1)*N

Therefore, the running time of merge sort to sort an array of size N is bounded by:

$$(log(N)+1) \times N = \mathbf{O(N \times log(N))}$$

2) In early versions of Quick Sort where the leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

1) Array is already sorted in the same order.

2) Array is already sorted in reverse order.

3) All elements are same (special case of case 1 and 2)

-- Since these cases are very common use cases, the problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot.

-- With these modifications, the worst case of Quick sort has less chances to occur, but the worst case can still occur.

-- Since you're not guaranteed anything about the order of the input, it's possible that the **n/2 position has the smallest/largest element of the input.**

-- Then quicksort will proceed and put everything else on one side of the pivot.

-- In this worst case the middle element is placed at the starting or ending of the list after partition procedure.

-- Then the recursive equation will be like $T(n)=T(n-1)+o(n)$ which results in time complexity as $\mathbf{O(n^2)}$.

3) For removing all the duplicates from the list first we will sort the list and pass it to the function which returns the size of the new array with no duplicates.

```
def removeDuplicates(arr, n):
      if n == 0 or n == 1:
      return n

      # To store index of next unique element
      j = 0

      # traversing through the array
      for i in range(0, n-1):
      if arr[i] != arr[i+1]:
            arr[j] = arr[i]
            j += 1

      arr[j] = arr[n-1]
      j += 1
      return j

arr = [1, 3, 2, 4, 5, 2, 4, 5, 4] #example input
arr.sort() #to sort the array
n = len(arr)

# removeDuplicates() returns
# new size of array.
n = removeDuplicates(arr, n)

# Print updated array
for i in range(0, n):
      print (" %d "%(arr[i]), end = " ")

#example output
1 2 3 4 5
```

4) Given an array of numbers of size n. It is also given that the array elements are in range from 0 to $n^2 – 1$. So our aim is to sort the given array in linear time.

- If we use Counting Sort, it would take O(n^2) time as the given range is of size n^2. Using any comparison based sorting like Merge Sort,Heap sort.. etc would take O(nLogn) time.
- So we will use the Radix Sort algorithm.

Following is standard Radix Sort algorithm.

- Do following for each digit i where i varies from least significant digit to the most significant digits)
- Sort input array using counting sort (or any stable sort) according to the i'th digit

Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for the decimal system, b is 10.

Since n^2-1 is the maximum possible value, the value of d would be $O(\log_b(n))$.

So overall time complexity is $O((n+b)*O(\log_b(n))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k.

The idea is to change base b. If we set b as n, the value of $O(\log_b(n))$ becomes O(1) and overall time complexity becomes O(n).

Example:

arr[] = {0, 10, 13, 12, 7}

Let us consider the elements in base 5. For example 13 in base 5 is 23, and 7 in base 5 is 12.

arr[] = {00(0), 20(10), 23(13), 22(12), 12(7)}

After the first iteration (Sorting according to the last digit in base 5), we get.

arr[] = {0<u>0</u>(0), 2<u>0</u>(10), 1<u>2</u>(7), 2<u>2</u>(12), 2<u>3</u>(13)}

After second iteration, we get

arr[] = {<u>0</u>0(0), <u>1</u>2(7), <u>2</u>0(10), <u>2</u>2(12), <u>2</u>3(13)}

Following is the implementation to sort an array of size n where elements are in range from 0 to $n^2 – 1$ in python:

```
# A function to do counting sort of arr[]
# according to the digit represented by exp.
def countSort(arr, n, exp):
    output = [0] * n # output array
    count = [0] * n
    for i in range(n):
        count[i] = 0   # Store count of occurrences in count[]
    for i in range(n):
        count[ (arr[i] // exp) % n ] += 1
 # Change count[i] so that count[i] now contains actual position of this
digit in output[]
    for i in range(1, n):
        count[i] += count[i - 1]

    for i in range(n - 1, -1, -1): # Build the output array
        output[count[ (arr[i] // exp) % n] - 1] = arr[i]
        count[(arr[i] // exp) % n] -= 1

    # Copy the output array to arr[], so that
    # arr[] now contains sorted numbers according
    # to current digit
    for i in range(n):
        arr[i] = output[i]

# The main function to that sorts arr[] of
# size n using Radix Sort
def sort(arr, n) :

    countSort(arr, n, 1)  # Do counting sort for first digit in base n

    countSort(arr, n, n)  # Do counting sort for second digit in base n.

Example:

    # Since array size is 7, elements should
    # be from 0 to 48
    arr = [40, 12, 45, 32, 33, 1, 22]
    n = len(arr)
    sort(arr, n)

    print("Sorted array is")
    print(*arr)
```

5) Running time analysis of Quick Sort:

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as pivot.
2. Always pick the last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition().

Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

All this should be done in linear time.

The running time of the Quick Sort Algorithm depends on how the partition() method will split the input array.

Since the partition() method depends on the pivot, we conclude that the value of the pivot will affect the performance of Quick Sort.

The Quick Sort algorithm performs the *best* when the pivot is the *middle* value in the array

- In the best case, the input array is divided into 2 arrays each containing approximately *n/2* element
- Summary of the execution:

```
QuickSort( input array with n elements )
 {   ....
     partition( a (array with n elements) );

     QuickSort( array with n/2 elements );
     QuickSort( array with n/2 elements );
 }
```

Amount of work done by QuickSort(array of size n)
= Amount of work done by partition(array of n elements)
+ Amount of work done by QuickSort(array of size n/2)
+ Amount of work done by QuickSort(array of size n/2)

Or:

$T(n) = partition(n) + T(n/2) + T(n/2)$

= partition(n) + 2*T(n/2)

Where: T(n) = Amount of work done by QuickSort(array of size n)

Amount of work done by partition(array of n elements): From previous result---- running time of partition(array of n elements) is equal to:

- = (*n − 1*) (while loop) + 1 (copying the pivot back)
- = *n*

Therefore: *best case* performance of Quick Sort

T(n) = partition(n) + 2*T(n/2) // partition(n) = n
= n + 2*T(n/2)
= 2*T(n/2) + n

T(1) = 1 // For the if-check operation

Solving the recurrence relation:

T(n) = 2*T(n/2) + n // T(n/2) = 2*T(n/4) + (n/2)
= 2*[2*T(n/4) + n/2] + n
= 2^2*T(n/4) + n + n
= 2^2*T(n/4) + 2n // T(n/4) = 2*T(n/8) + (n/4)
= 2^2*[2*T(n/8) + (n/4)] + 2n
= 2^3*T(n/8) + 2^2*(n/4) + 2n
= 2^3*T(n/8) + n + 2n
= 2^3*T(n/8) + 3n
= 2^4*T(n/16) + 4n and so on....

= **2^k*T(n/(2^k)) + k*n** // Keep going until: n/(2^k) = 1 <==> n = 2^k

= 2^k*T(1) + k*n
= 2^k*1 + k*n
= 2^k + k*n // n = 2^k
= n + k*n
= n + (lg(n))*n
= n*(lg(n) + 1)
~= n*lg(n))

Thus**, T(n) = n*log(n).**