Roll No: EE18B067 Name:Abhishek Sekar

Date: October 29, 2021

1. Show that the running time of the merge-sort algorithm on n-element sequence is $\mathcal{O}$(nlogn), even when n is not a power of 2.

Solution:
We can easily see that merge sort is $\mathcal{O}$(nlogn) when n is a power of 2 with the divide and conquer based recurrence relation of $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, where O(n) is the cost incurred in the merging the n elements.
Let's prove that the time complexity doesn't change even when n is not a power of 2.
Proof by Induction:
For any $n \in \mathbb{N}$,we have, the recurrence $T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + \mathcal{O}(n)$ for merge sort.
Thus, from asymptotic analysis, we have, $T(n) \leq T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + C \cdot n$ for a sufficiently large positive constant C.
Let us frame our inductive hypothesis,
Inductive Hypothesis:

$$\begin{aligned} T(n) &= \mathcal{O}(n \log(n)) \forall n \in \mathbb{N} \\ &\leq C \cdot n \log(n) \qquad \text{For a sufficiently large positive constant C} \end{aligned}$$

Base Case:
Our base case is n = 1. For n = 1, T(1) = 0, as there is no computational cost to sort a single element.
Inductive Step:

- Assume that the Inductive Hypothesis holds till some k $\in \mathbb{N}$ such that k<n, i.e., $T(k) \leq C \cdot k \log(k)$
- Using this, try proving that the Inductive Hypothesis holds for n.

$$\begin{aligned} T(n) &\leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + C \cdot (n) && \text{(From recurrence for merge sort)} \\ &\leq C \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C \cdot \left\lceil \frac{n}{2} \right\rceil \cdot \log\left(\left\lceil \frac{n}{2} \right\rceil\right) + C \cdot (n) && \text{(As } \left\lfloor \tfrac{n}{2} \right\rfloor \text{ and } \left\lceil \tfrac{n}{2} \right\rceil < n\text{)} \\ &\leq C \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \log\left(\left\lceil \frac{n}{2} \right\rceil\right) + C \cdot \left\lceil \frac{n}{2} \right\rceil \cdot \log\left(\left\lceil \frac{n}{2} \right\rceil\right) + C \cdot (n) && \text{(As } \left\lfloor \tfrac{n}{2} \right\rfloor \leq \left\lceil \tfrac{n}{2} \right\rceil \text{ and as log(n) is increasing for } n \in \mathbb{N}\text{)} \\ &\leq C \cdot n \cdot \log\left(\left\lceil \frac{n}{2} \right\rceil\right) + C \cdot (n) && \text{(As } \left\lfloor \tfrac{n}{2} \right\rfloor + \left\lceil \tfrac{n}{2} \right\rceil = \text{n}, \forall n \in \mathbb{N}\text{)} \\ &\leq C \cdot n \cdot \log\left(\frac{2n}{3}\right) + C \cdot (n) && \text{(As } \left\lceil \tfrac{n}{2} \right\rceil \leq \tfrac{2n}{3} \ \forall n \geq 2\text{)} \end{aligned}$$

The above statement is easily verified by looking at the below graph too,

Figure 1: Visualisation of $\left\lceil \frac{n}{2} \right\rceil \leq \frac{2n}{3}$

$$\begin{aligned} T(n) &\leq C \cdot n \left(\log(n) + \log\left(\frac{2}{3}\right)\right) + C \cdot (n) \\ &\leq C \cdot n \log(n) + C \cdot n \left(1 - \log\left(\frac{3}{2}\right)\right) \end{aligned}$$

Now, analyzing $\left(1-\log\left(\frac{3}{2}\right)\right)$, on taking log with respect to base 2, we see that, $\left(1-\log\left(\frac{3}{2}\right)\right) = 2-\log 3 < 1 \leq \log(n)\ \forall n \geq 2$.
Thus, $C \cdot n\log(n) + C \cdot n\left(1-\log\left(\frac{3}{2}\right)\right) \leq 2C \cdot \log(n)$.
Therefore, from this, $T(n) \leq 2C \cdot \log(n)$ and as 2C is just another constant, the Inductive Hypothesis holds for n.

- Therefore, by the principle of mathematical induction, as the Inductive Hypothesis holds for n = 1 and for n > k for some $k \in \mathbb{N}$, the result holds $\forall n \in \mathbb{N}$

Thus, as $T(n) \leq 2C \cdot \log(n)$, we have T(n) = $\mathcal{O}(n\log n)$ even when n is not a power of 2.

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

Solution:
Best and Worst Case
The speed of the quick sort algorithm is determined by how good the partition is with respect to the pivots final position. Notably, the best case scenario for quick sort is when the pivot's final position is the same as it's initial position, which is $\lfloor \frac{n}{2} \rfloor$ here.This results in the partitions being equal.
Thus, the worst case scenario for quick sort would be when the pivot's final position is the farthest away from its initial position. This happens when the final position of the pivot element is either the leftmost or the rightmost index of the sorted array. In either case, the partitions generated are of length n-1 and 0.
Recurrence Relation:
Thus, the recurrence relation for the worst case of quick sort can be written as $T(n) = T(n-1) + O(n-1)$ as we've to make n-1 comparisons before the next partitioning step.
Now, let us prove that this recurrence relation results in T(n) = $\Omega(n^2)$.
Proof:

$$
\begin{aligned}
T(n) &= T(n-1) + O(n-1) \\
&\Rightarrow T(n-1) + k \cdot (n-1) && \text{(For an appropriate positive constant k)} \\
&\Rightarrow T(n-2) + k \cdot (n-2) + k \cdot (n-1) && \text{(Substituting n = n-1 in the above recurrence relation)} \\
&\vdots \\
&\Rightarrow T(1) + k \cdot (2+3+4+\ldots+(n-2)+(n-1)) && \text{(Repeating the above step several times)} \\
&\Rightarrow 0 + k \cdot \left(\sum_{k=2}^{n-1}(k)\right) && \text{(As there is no computational cost in sorting 1 element)} \\
&\Rightarrow -k + k \cdot \left(\sum_{k=1}^{n-1}(k)\right) && \text{(Adding and subtracting k)} \\
&\Rightarrow -k + k \cdot \frac{(n)(n-1)}{2}
\end{aligned}
$$

From this, it is clear that T(n) = $\Omega(n^2)$.
Sequence
As shown by the proof, this occurs for the worst case scenario for quick sort when the pivot's final position is farthest away from its initial position.
As we are fixing the index $\lfloor \frac{n}{2} \rfloor$ as the pivot, it follows from the worst case scenario that the pivot has to be the largest or the smallest element in the array.
More specifically, we see that it has to be the largest element of the array.
If n is even, then, $\lfloor \frac{n}{2} \rfloor$ is just $\frac{n}{2}$ and placing the largest element at the pivot would mean that it is $\frac{n}{2}$ away from its final position whereas the smallest element would be 1 step closer to its position if it were on the pivot(assuming the array is indexed from 1).

If n is odd, then, $\lfloor \frac{n}{2} \rfloor$ is to the left of the mid-point of the array thus making the largest element the obvious candidate for the pivot.
Hence, the kind of sequence that will cause quick sort to operate in such a fashion is one, where the largest element of the sequence sits at the pivot position, i.e., the $\lfloor \frac{n}{2} \rfloor$ index of the array. Quick sort will have the required time complexity if we select the largest element as the pivot for the partitions as well.

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

Solution:
The Process:

- First, we will have to sort the original sequence A. This can be done using merge sort or quick sort in roughly $\mathcal{O}$(nlog(n)) time.
- Next, we iterate through the sorted array in a sequential fashion and append all the unique elements to another array. This takes $\mathcal{O}$(n) time as we're just iterating throughout the array.
- Lastly, we return this array containing the resulting elements.

We can include an additional step before returning the output should the ordering of the array A be retained. In this case, we'll start off with another array which is just an array with A in it's first row and it's indices in the second. Now, the sorting will take place based on the first row of this new array and hence, in the penultimate step, the original ordering can be preserved by sorting the array again, but now across the row of indices.

Time Complexity:
Thus, from our description of the algorithm, we see that it involves one or two steps comprising a sort and one step which involves traversing the whole array.
Thus, our net time complexity will be $\mathcal{O}$(sort) +$\mathcal{O}$(traversal) = $\mathcal{O}$(nlog(n)) + $\mathcal{O}$(n) as there are n elements in the array.
Thus, the net time complexity will be of the order, $\mathcal{O}$(nlog(n)).

The Algorithm:
The algorithm is described below with *pythonic* syntax.

```
#Code for merge sort

def merge(L,R): # function to merge the left part and the right part into a sorted array
    Result = [] #initialising the result array
    i = 0
    j = 0
    n1 = len(L)
    n2 = len(R)

    while (i < n1 and j < n2):
        if (L[i] <= R[j]):
            Result.append(L[i]) #append the smaller element
            i += 1              #update i
        else:
            Result.append(R[j])
```

```
            j += 1

    #Copy the remaining elements of L, should there be any
    while (i < n1):
        Result.append(L[i])
        i += 1

    #Copy the remaining elements of R, should there be any
    while (j < n2):
        Result.append(R[j])
        j += 1

    return Result

def mergesort(A): #function that performs merge sort on an array A
    n = len(A)

    if(n <= 1):
        return A

    else:

        #The divide and conquer step, where we're splitting A into L and R portions
        L = mergesort(A[:n//2])
        R = mergesort(A[n//2:])

    return merge(L,R)


def unique(A): #removes all duplicates from array A
    temp = []
    A = mergesort(A) #O(nlogn) step
    ele = A[0]
    temp.append(A[0])
    for i in range(1,len(A)):
        if(A[i] != ele):
            ele = A[i]  #make ele the new element to check in the future steps
            temp.append(A[i]) #append every unique element

    return temp
```

Sample Input: [68,66,44,44,3,3,2,3,4,5,6,7,8,9,2,3,5,1,0,-1,-5,-6.7,-8]
Sample Output: [-8, -6.7, -5, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 44, 66, 68]
As can be seen, all the duplicate elements are removed when we look at the output array. Thus, this is an efficient method to remove duplicate elements from an array.
Alternative approach
We can exploit the properties of data structures present in a coding language to come up with yet another efficient approach. The duplicates can be removed in a different fashion by making use of the set data structure in Python.
We can create a data structure which behaves like the set of all elements in A.
We therefore iterate throughout A adding each element of A to the set. However, the property of a set is that, adding an element which is already present in the set, doesn't change the set.
Therefore, finally the set will only contain all the unique elements present in the array A.
Algorithm for the alternate approach

```
def unique_2(A): #O(n) approach to get rid of duplicates
    temp = set()
    for i in range(len(A)):
        temp.add(A[i])
    return list(temp)
```

As we just have to traverse the array A, the time complexity of this approach is just $\mathcal{O}$(n).

4. Show that quick-sort's best case running time is Ω(nlogn).

Solution:
From the solution in the second question, we'd intuitively expect the best case to occur when the chosen pivot element is the median or close to the median of the array we're trying to sort.
Recurrence Relation
The general recurrence relation for quick sort, if we choose the i^{th} index as the pivot is as follows,

$$T(n) = \min_{1 \le k < n} \left(T(k) + T(n-k-1)\right) + \mathcal{O}(n)$$

As there is a *best* case to the quick sort, W.L.O.G we can assume that the recurrence relation results in T(n) having a time complexity which is a convex function, i.e., there is a unique minimizing index for the above recurrence relation.
Let us differentiate the above equation with respect to k to find this unique minimizing index.

$$\frac{dT(n)}{dk} = \frac{d\left(\left(T(k) + T(n-k-1)\right) + \mathcal{O}(n)\right)}{dk}$$
$$0 = \frac{dT(k)}{dk} + -1 \cdot \frac{dT(n-k-1)}{dk}$$
$$\frac{dT(k)}{dk} = \frac{dT(n-k-1)}{dk}$$

Therefore, from this, one critical point is k = n-k-1 which by our assumption leads to the best case time complexity.
Therefore, the minimizing index is k = $\frac{n-1}{2}$. If n is even, then k is appropriately the floor or the ceiling of $\frac{n-1}{2}$ which is where our intuition led us.
Thus, with this, our recurrence relation becomes,

$$T(n) = T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + \mathcal{O}(n)$$

This is very similar to what we had in the first problem.
Therefore, let us use an inductive approach as we did in the first problem. Let us frame our inductive hypothesis,
Inductive Hypothesis:

$$\begin{aligned} T(n) &= \Omega(n\log(n)) \forall n \in \mathbb{N} \\ &\geq C \cdot n\log(n) \qquad \text{For a sufficiently small positive constant C} \end{aligned}$$

Base Case:
Our base case is n = 1. For n = 1, T(1) = 0, as there is no computational cost to sort a single element.
Inductive Step:

- Assume that the Inductive Hypothesis holds till some k $\in \mathbb{N}$ such that k<n, i.e., $T(k) \geq C \cdot k\log(k)$
- Using this, try proving that the Inductive Hypothesis holds for n (≥ 2).

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + k \cdot (n) \qquad \text{(From derived recurrence, for an appropriate k> 0)} \\ &\geq C \cdot \left\lfloor \frac{n-1}{2} \right\rfloor \cdot \log\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + C \cdot \left\lceil \frac{n-1}{2} \right\rceil \cdot \log\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + k \cdot (n) \qquad \text{(As } \left\lfloor \tfrac{n}{2} \right\rfloor \text{ and } \left\lceil \tfrac{n}{2} \right\rceil < n\text{)} \\ &\geq C \cdot \left\lfloor \frac{n-1}{2} \right\rfloor \cdot \log\left(\frac{n-1}{2}\right) + C \cdot \left\lceil \frac{n-1}{2} \right\rceil \cdot \log\left(\frac{n-1}{2}\right) + k \cdot (n) \\ &\geq C \cdot (n-1) \cdot \log\left(\frac{n-1}{2}\right) + k \cdot (n) \qquad \text{(As } \left\lfloor \tfrac{n}{2} \right\rfloor + \left\lceil \tfrac{n}{2} \right\rceil = \text{n, } \forall n \in \mathbb{N}\text{)} \end{aligned}$$

The above statement is easily verified by looking at the below graph too,

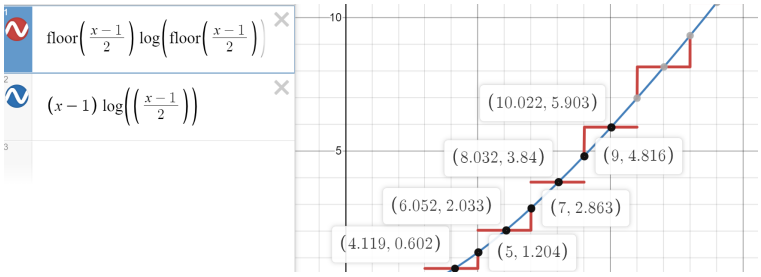

Figure 2: Visualisation of the above inequality

Our inequality is valid, as we're only looking at $n \in \mathbb{N}$ and not arbitrary n.
From this, we can say that,

$$\begin{aligned} T(n) &\geq C \cdot (n-1) \cdot \log(n-1) - C \cdot (n-1) + k \cdot (n) \\ &\geq C \cdot n\log(n-1) - C \cdot \log(n-1) - C \cdot (n-1) + k \cdot (n) \\ &\geq C \cdot n\log(n-1) - C \cdot \log(n-1) + \Theta(n) \\ &\geq C \cdot n\log\left(\frac{n}{2}\right) - C \cdot \log(n-1) + \Theta(n) \qquad \text{(For } n \geq 2, n \in \mathbb{N}\text{)} \\ &\geq C \cdot n\log(n) - C \cdot \log(n-1) - C \cdot n + \Theta(n) \\ &\geq C \cdot n\log(n) \end{aligned}$$

As we can find a sufficiently small positive constant C such that $n\log(n)$ is the dominating term on the R.H.S of the inequality.

- Therefore, by the principle of mathematical induction, as the Inductive Hypothesis holds for n = 1 and for n > k for some $k \in \mathbb{N}$, the result holds $\forall n \in \mathbb{N}$

Therefore, quick sort's best case running time is Ω(nlogn).

5. Implement in python, a bottom-up merge-sort for a collection of items by placing each item in its own queue, and then repeatedly merging pairs of queues until all items are sorted within a single queue.

Solution:
Code:

```
def merge(A, temp, frm, mid, to):

    k = frm
    i = frm
    j = mid + 1

    #same as the merge step we did before, but we only do this till the mid portion as the
    while (i <= mid and j <= to):
        if (A[i] < A[j]):
            temp[k] = A[i]
            i += 1
        else:
            temp[k] = A[j]
            j += 1

        k += 1

    # copy remaining elements
    while (i < len(A) and i <= mid):
        temp[k] = A[i]
        k += 1
        i += 1

    # copy back to the original list to reflect sorted order
    for i in range(frm, to + 1):
        A[i] = temp[i]


def mergesort(A):

    #create a copy of the array
    temp = A.copy()

    #divide the list into blocks of m
    m = 1
```

```
    while m <= len(A) - 1: #while m is within the length of the array
        for i in range(0, len(A)-1, 2*m): #fragment the array
            frm = i  #index from
            mid = i + m - 1 #the middle index
            to = min(i + 2*m - 1, len(A) - 1) #the index to
            merge(A, temp, frm, mid, to) #perform the merge operation on the array we're co
        m = 2*m #update m
    return A
```

Input and Output

- **Input:** an array A
- **Output:** sorted array A

Sample Input and Sample Output:
input: [2, 3, 4, 1, 5, 7, 6, 8, 9, 4, 5, 6, 7, 3, 4]
output: [1, 2, 3, 3, 4, 4, 4, 5, 5, 6, 6, 7, 7, 8, 9]
Explanation
The code is written in the conventional fashion, i.e., a function to merge the arrays and a function to sort them.

- Merge: The function merge remains mostly the same as a conventional merge sort with just one major difference. It takes in three location parameters as opposed to the left and right part of the array that I had earlier implemented in question 3.However, we can draw parallels to this approach. The left part of the array here, can be thought of as starting from the index from and running till the index mid while the right part of the array runs from the index mid + 1 till the index to.
  There's just one notable difference. When we copy the remaining elements into the array as we did before, we just do so till the index mid. This is because, the other part of the array A would have already been sorted. Finally, the temporary array is copied back into A to reflect the sorted order.
- Mergesort: As instructed in the question, the merge sort first breaks the array A into single element queues.
  There is a while loop which supervises the pairwise merging part also described by the question.
  In the first, iteration, these single element queues are merged in pairs by calling the merge function. Thus, this step ensures that every two element queue is sorted.
  After this, we update m by doubling it. In the next step, we break the array A into the queues of size 2 which were sorted earlier and merge them to produce four element queues which are sorted.
  This step keeps repeating till m goes larger than the length of the array which implies that the array has been sorted.

Example to Illustrate:
The above algorithm is best understood by an example.
Consider the input [3,2,1,4] for this example.
Here, the length of the array is 4.

- In the first iteration of the while loop. For the first merge call, we have from = 0, mid = 0, to = 1. So what this does is, it takes the left part of the array to be [3], a single element queue and the right part of the array to be [2]. Merging results in the array A being modified only till the index to, as required. A will then become [2,3,1,4] as 2 is less than 3.
- For the next merge call, we have from = 2, mid = 2 and to = 3. This takes the left part of the array to be [1] and the right part to be [4]. On merging, we get A to be [2,3,1,4] as 1 is less than 4 which is already maintained.
- Now we update m to 2. Thus in this iteration, we consider the pairs [2,3] and [1,4] for the merge call. The result of this merge call will give us [1,2,3,4] which is the sorted array. The while loop terminates here as m = 4 is greater than 3 which is the domain of the array.

Example Steps in the Algorithm
Type the elements of the input array in a space separated manner
3 2 1 4
Left queue: [3]
Right queue: [2]
Output of merge call: [2, 3, 1, 4]
Left queue: [1]
Right queue: [4]
Output of merge call: [2, 3, 1, 4]
Left queue: [2, 3]
Right queue: [1, 4]
Output of merge call: [1, 2, 3, 4]
Sorted array: [1, 2, 3, 4]
Time Complexity:
The time complexity of the bottom up merge sort will be the same as the divide and conquer based merge sort and this can be seen in the way the algorithm was implemented. Therefore, the time complexity of this algorithm is $\mathcal{O}(\text{nlog(n)})$.