EE4371 - Assignment 3

Om Shri Prasath, EE17B113

Q1. Show that the running time of the merge-sort algorithm on n-element sequence is $O(n \log n)$, even when n is not a power of 2.

Solution :

The steps followed for merge-sort algorithm is as follows:

1. If the list is of length 0 or 1, then it is already sorted. Otherwise:
2. Divide the unsorted list into two sublists of about half the size.
3. Sort each sublist recursively by re-applying merge sort.
4. Merge the two sublists back into one sorted list.

Let the running time complexity of merge-sort be $T(n)$. From the description of the algorithm given above, steps 1 and 2 take around constant time so it can be ignored. Step 3 is the recursion part which takes $T\left(\frac{n}{2}\right)$ for each sublist, thus $2T\left(\frac{n}{2}\right)$ overall. Step 4 is of $\Theta(n) \approx n$ since we need to pass through the array once for merging the sublist.

Now, we have

$$T\left(n\right) = 2T\left(\frac{n}{2}\right) + n$$

If we recursively substitute the expression for $T(n)$ in the above equation k times, we get

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Now, we have to apply the stopping condition for the recursion, which is when the length of the array becomes 1, i.e. $\dfrac{n}{2^k} = 1$, since $T(1) = 1$ from our assumption above. Now if n was an power of 2, we can get $k = \log n$, but if n is not a power of 2 what should we do? Here we need to understand what does k actually denote. k denotes the **depth of the recursion** we need to go such that we split the problem into single element arrays. For an array with length not a power of 2, the depth will be $\lceil \log(n) \rceil \leq \log(n) + 1$. Thus for analysis of arrays of size not a power of 2, we can use $k = \log n + 1$.

$$T\left(n\right) = 2^{\log(n)+1} + (\log n + 1)n = 3n + n\log n$$

We know that $3n + n\log n = O(n\log n)$, since the higher order term $n \log n$ dominates over $3n$.

Thus, we have proved that the running time of merge sort is $O(n \log n)$

Q2. Consider a modification of the deterministic version of the quick-sort algorithm where we choose the element at index $\left\lfloor \frac{n}{2} \right\rfloor$ as our pivot. Describe the kind of sequence that would cause this version of quick-sort to run in $\Omega(n^2)$ time.

Solution :

The steps followed for quick-sort algorithm is as follows:

1. We pick an element called the *pivot*, around which we sort the array
2. We move the elements which are smaller than pivot to the left of the pivot and elements greater than the pivot to the right of the pivot. After this partitioning the pivot is placed in its correct position.
3. We sort the two sublists of elements greater and less than the pivots recursively. Base case consists of either one or zero elements, in which case the array is sorted

In the given case we see that the pivot is chosen as always the element at $\left\lfloor \frac{n}{2} \right\rfloor$.

The time complexity is given by the following recurrence :

$$T(n) = T(|L|) + T(|R|) + \Theta(n)$$

where $|L|$ is the size of the elements smaller than the pivot and $|R|$ is the size of the elements greater than the pivot. Ideally the value of both $|L|$ and $|R| \sim \frac{n}{2}$, which gives us the recurrence as

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

which gives us the time complexity of $T(n) = O(n \log n$ from the above question. But in some cases one of either $|L|$ or $|R|$ becomes 0 and the other one becomes $n-1$, which can happen if the pivot is chosen as either the smallest element or the greatest element. In this case, we can use $\Omega(n)$, since this is the worst case time complexity and $\Theta(n) \implies \Omega(n)$. The recurrence equation now becomes

$$T(n) = T(n-1) + \Omega(n)$$

on recursively substituting for $T(n)$, we get

$$T(n) = T(n-k) + k\Omega(n)$$

The stopping condition is $T(0) = 0$, for which $n - k = 0 \implies n = k$. Thus, we get

$$T(n) = T(0) + n\Omega(n) = \Omega(n^2)$$

Thus, in the worst case of always choosing either the greatest element or the least element in the array as the pivot, we get the worst case time complexity of $\Omega(n^2)$

Now, to describe the condition of the array in which this occurs, the array should be created such that the middle element always contains either the smallest or the greatest element while partitioning, which will become the pivot.

Thus, the condition where the pivot is always either the greatest or the smallest array is created by taking a sorted array and rotating it either clockwise or anticlockwise by $\left\lfloor \frac{n}{2} \right\rfloor$

E.g. Consider the array $[1, 2, 3, 4, 5]$, we rotate it by $\left\lfloor \frac{5}{2} \right\rfloor = 2 \implies [4, 5, 1, 2, 3]$

1. Pivot position = $\left\lfloor \frac{5}{2} \right\rfloor = 2 \implies$ Pivot = 1. Thus $L = []$, $R = [4, 5, 2, 3]$. No. of comparisons = 4
2. Pivot position = $\left\lfloor \frac{4}{2} \right\rfloor = 2 \implies$ Pivot = 2. Thus $L = []$, $R = [4, 5, 3]$. No of comparisons = 3
3. Pivot position = $\left\lfloor \frac{3}{2} \right\rfloor = 1 \implies$ Pivot = 5. Thus $L = [4, 3]$, $R = []$. No of comparisons = 2
4. Pivot position = $\left\lfloor \frac{2}{2} \right\rfloor = 1 \implies$ Pivot = 3. Thus $L = [4]$, $R = []$. No. of comparisons = 1

Thus, in the above case either one of the L or R are empty and the other one consists of $n-1$ elements. The number of comparisons is of order $4 + 3 + 2 + 1 = 10 = \frac{(5 \times 4)}{2} = \frac{n(n-1)}{2} = \Omega(n^2)$

Q3. Describe and analyze an efficient method for removing all duplicates from a collection A of n elements.

Solution : $O(n \log n)$ Time Complexity

Given an collection of n elements, we need to remove duplicates from the collection. Assuming we do not need to keep the order, the operation can be done as follows :

1. We sort the array, using either quick sort or merge sort to sort the array
2. Once the array is sorted, we do the following
   A. If the length of the array is 1, we just return it. Else:
   B. We keep two iterators i and j which are initialized to 0 and 1
   C. If element at position i is not equal to j, we increment i and write the element present at i into j and then increment j too
   D. Otherwise we just increment j
   E. We continue the loop until j reaches the end of the array
   F. We return the elements present from 0:i+1 as the non-duplicates array output.

The above algorithm has $O(n \log n)$ time complexity for sorting and $O(n)$ time complexity for removing the duplicates, so the overall time complexity is $O(n \log n)$

In [1]:

```python
"""
Python code to implement removal of duplicates from collection
"""

"""
Function to remove duplicates from arr
"""


def remove_duplicates(arr):

    n = len(arr)

    # Returning if only one element is present
    if n == 1:
        return arr

    # Sorting the array
    arr.sort()

    i = 0
    j = 1

    # Loop to remove the duplicates
    while j < n:

        # If element do not match, write it a previous position
        if arr[i] != arr[j]:
            i += 1
            arr[i] = arr[j]

        j += 1

    # Return the non-duplicates part of array
    return arr[0 : i + 1]


# Testing the code

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

print(remove_duplicates(arr))
```

```
[1, 2, 3, 4, 5, 6]
```

Q4. Show that quick-sort's best-case running time is $\Omega(n \log n)$.

Solution :

The recurrence relation for quick-sort algorithm is given as :

$$T(n) = T(k) + T(n-k-1) + \Theta(n)$$

where k is the number of elements less than the pivot.

Now, we have to prove that for a fixed c and n_o, $T(n) \geq cf(n) \,\forall\, n > n_o$, where $f(n) = n \log n$ (condition for $\Omega(n)$)

We need to find the minimum possible bound of $T(n)$, and show that it is of the order $\Omega(n \log n)$. For this we will assume that the above asymptotic relation is true and show that our assumption is true afterwards. Thus,

$$\min_k T(n) = \min_k T(k) + T(n-k-1) + \Theta(n) \geq \min_k k\log(k) + (n-k-1)\log(n-k-1) + \Theta(n)$$

Differentiating w.r.t k, we get

$$\log(k) - \log(n-k-1) = 0 \implies k = \frac{n-1}{2}$$

$$\implies \min_k T(n) \geq \frac{n-1}{2}\log\frac{n-1}{2} + \frac{n-1}{2}\log\frac{n-1}{2} + \Theta(n) \geq (n-1)\log\frac{(n-1)}{2}$$

We see that for $n_o = 2$ and $c = 0.5$, $T(n) \geq (n-1)\log\frac{(n-1)}{2} \geq cn\log n$. Thus the quick-sort's best-case running time is $\Omega(n \log n)$.

One thing to note that is we get this best case running time when $k = \frac{n}{2}$, i.e. we choose the **median** of the array as the pivot in all of the steps.

Q5. 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 : $O(n \log n)$ Time Complexity

We use the merge sort algorithm which we had discussed before in an bottom up manner as follows :

- We first create n queues which each contain one item intially.
- Until the number of queues become 1, we do the following :
  - We merge sort consecutive queues to create a new queue, thus reducing the number of queues by $\frac{n}{2}$
  - If the number of elements is odd, we keep the last queue as it is

In [2]:

```python
"""
Python code to merge sort in bottom-up manner

"""


"""
Queue implementation

push()  -> Pushes element into the queue
pop()   -> Pops element from the queue
size()  -> Returns the size of the queue
"""


class Queue:
    def __init__(self):
        # Initializing the queue as a list
        self.queue = []
        # Variable to hold size
        self.size = 0

    def push(self, elem):
        # Push element at the end
        self.queue.append(elem)
        # Increase the size
        self.size += 1

    def pop(self):
        if self.size > 0:
            # Pop element at the front
            t = self.queue.pop(0)
            # Decrese the size
            self.size -= 1

            return t

        else:
            # Return None if queue is empty
            return None

    def size(self):
        # Return size of array
        return self.size


"""
Function to merge two queues in a sorted manner

Input -> Two queues
Output-> Sorted queue containing elements of both queues
"""


def merge(x, y):

    # Resulting queue
    res = Queue()

    # Pop the values from the queue
    a = x.pop()
    b = y.pop()

    # We append the smaller element to res
    while a and b:
        if a <= b:
            res.push(a)
            a = x.pop()
        else:
            res.push(b)
            b = y.pop()

    # Append the leftover elements
    while a:
        res.push(a)
        a = x.pop()

    while b:
        res.push(b)
        b = y.pop()

    # Return the sorted merged queue
    return res


"""
Function to merge sort elements

Input  -> Array of elements
Output -> Queue of sorted elements
"""


def merge_sort(l):

    # Array to store the queues
    s = []

    # Create the queues of elements
    for i in l:
        k = Queue()
        k.push(i)
        s.append(k)

    while len(s) > 1:
        # New array for holding merged queues
        s_temp = []

        n = len(s)
        i = 0

        while i < n:
            if i < n - 1:
                # Merging and appending to array
                s_temp.append(merge(s[i], s[i + 1]))
            else:
                # Append lone queue without merging
                s_temp.append(s[i])
            i += 2

        # Assign the new array to old array
        s = s_temp

    # Return the sorted queue
    return s[0]


# Testing the function
q = merge_sort([3, 2, 1, 4, 6, 5])
print(q.queue)
```

```
[1, 2, 3, 4, 5, 6]
```