Submission for Problem Set 3 (due Thursday, October 11, 9:59 PM)

Name: Deepanshu Parihar

1. (4 + 4 = 8 points) Multi-Merge

Suppose you would like to merge k sorted arrays, each of length n, into one sorted array of length nk.

(a) One strategy is to use the merge operation we studied in class to merge the first two arrays to obtain a new sorted array of size $2n$, then merge in the third to obtain a new sorted array of size $3n$, then merge in the fourth, and so on until you obtain the desired sorted array of length nk.

Formalize the above algorithm in pseudocode and analyze its time complexity, in terms of n and k.

Answer:
Algorithm-
Not coding for merge as Merge function defined in Merge Sort algo is being used as it is.

a=Merge(Array 1 , Array 2)
$while(n \geq 3\&\&n \leq k):$
a=Merge(a , array(n))
n++

Time Complexity-

To merge two arrays of size n it takes 2n time, on merging the new array with another array of n size it takes 3n time.
So this will continue till we add k arrays of n size.
We get,

$T(n) \leq n + 3n + 4n + 5n + 6n + ... + kn$
$\leq (2 + 3 + 4 + 5 + 6 + k)$
$\leq ((k(k + 1)/2) - 1)$
$\leq \; (nk^2 + nk - 2n)/2$
$T(n) = \Theta \; (nk^2)$

(b) Give a more efficient algorithm to this problem, using divide and conquer. Analyze its time complexity, in terms of n and k.

Answer:
Using divide and conquer

Suppose k is even

Now , for a more efficient algorithm, consider taking two arrays at a time till we reach the kth array and merging them at each level.
When each array is of size n then we get a worst case complexity of 2n(k/2) as there are k/2 pairs and combining them will take a worst case running time of 2n

Now combing the arrays of size 2n will take a time of 4n in worst case and for k/4 pairs it results in 4n(k/4)

This continues till we get an array of size nk.

The work being done at each level for merging is nk i.e. no of elements , but the number of traversals or levels gets reduced to log k.
So we get a net of O(nklogk)
Hence this is an improvement on the previous stated algorithm.

2. (8 points) Finding widgets of majority type using pairwise testing

You have n widgets, each of which is of a certain *type*. You want to determine whether there is a *majority type*; i.e., if there is a type t such that the number of widgets of type t is *greater than* $n/2$. For instance, the set of 7 widgets with types A, A, B, C, A, C, A, respectively has a majority type (A) since there are 4 widgets of that type. On the other hand, the set of 6 widgets with types A, A, B, C, D, A has no majority type.

Unfortunately, you are unable to determine the type of any given widget. Instead, the only operation available to you is to call a subroutine EqualityTest that takes two widgets as input and returns $True$ if both are of the same type and $False$ otherwise.

Give a divide-and-conquer algorithm for determining if there is a majority type in a given set of n widgets. The running time of your algorithm is the number of calls made by your algorithm to EqualityTest. Analyze the running time of your algorithm.

Ideally, the running time of your algorithm should be $O(n \log n)$. You will receive extra credit if the running time of your algorithm is $O(n)$.

Answer:

Here I have implemented a fuction MaxWidget that using divide and conquer to divide the set recursively into two arrays.Then it checks and counts which widget is in majority and stores it in variables maxl and maxr.This is done at each level and if the sum of count of one type is gretaer than the other then that is returned.

Maxwidget(A, p, q) a = $\lfloor (p+q)/2 \rfloor$
ArrayLeft = MaxWidget(A, p, a)
ArrayRight = MaxWidget(A, a+1, q)

maxl = total number of maximum type of widgets in array on left
maxr = total number of maximum type of widgets in array on right
if(EqualityTest(ArrayLeft, ArrayRight) == True)

  return ArrayLeft

  if(maxl > a/2)

    return ArrayLeft

  if(maxr > a/2)

    return ArrayRight

  else

    There is no major element

Here for counting maxl and maxr the complexity is a total of n at each level and in worst case we have to divide by 2 throughout which gives a total no of steps of log n with base 2 as we recursively call MaxWidget function to calculate ArrayLeft and ArrayRight functions.
So we get a total of nlog(n)
Hence Proved.
(Collaborated with Viral Pandey)

3. (4 points) Finding the end of an infinite array?

Suppose you are given a very long array A, whose first n elements are integers (in arbitrary order) and the remaining elements are the special symbol ∞. You can access any position i in A by referring to $A[i]$. However, you do not know n.

Give an algorithm for determining n. Analyze the number of accesses to A made by your algorithm, in terms of n. For full credit, your algorithm must make $O(\log n)$ accesses to A.

Answer:
Algorithm-
Assuming array index to start from 1
First we set the bound in which the last integer element lies-

x=0
y=1
while(a[y] != infinity)
x=y
y= y*2

So now we have the range to be A[x] to A[y]
Now implementing the search function to find the number
It takes three inputs the array A , the lower bound and the upper bound

Search(A,x,y)

```
b=(x+y)/2
if(A[b]==infinity  A[b-1] != infinity  A[b+1] == infinity)
return b - 1
else if (A[b]==infinity  A[b+1] == infinity)
Search(A,x,b-1)
else
Search(A,b+1,y)
```

So here b-1 is the value of n.

Time Complexity-
The code for finding the range will take a worst case time of log(n) as we are incrementing by two so the base is 2 and the range where the last integer element exists might be inn the last range of the value of 2n. Therefore its log(n)

The code for the search function is a tweaking of binary search and here again the only work being done is in finding the middle of the array. So this is also log(n)
Hence we get a net time complexity of 2log(n)
Therefore , its O(logn)
Hence , proved.