Intro to Algorithms HW2

Jiayao Wu jwu86

February 12, 2016

1 Problem 1

Algorithm:

1. Perform merge sort on A and B.

2. Now A and B are sorted. Create two variable called i and j which are both initialized to 0. Repeat the following process until $i = k$.

Start from the first number in A and we call it $A[j]$ where j is from 0 to n-1, and perform binary search of $A[j]$ in array B. If found this number, j++, i++ and move to the next number in A and do the same process. If not found this number in B, then j++ and directly move to the next number in A.

3. Now, we have reached the situation when $i = k$. This means the k-th smallest entry of their intersection is A[j-1].

Pseudo code for this algorithm

Algorithm: findKthSmallest(A,B)
    Perform MergeSort(A) and MergeSort(B)
    $i = 0$ and $j = 0$;
    while (i != k) {
        if(BinarySearch(B, $A[j]$) is true) i++; j++;
        else j++;
    }
    return A[j-1]
    where BinarySearch(B, $A[j]$) is returning if B contains $A[j]$.

Prove correctness:

We know merge sort will give us sorted arrays as proven in class. Then we just need to prove second and third steps are correct.

Initialization:

i is used to update how many times we have found $A[j]$ in B and j is used to update where in the array A we are currently on.

Middle process:

Binary search will give us the result if $A[j]$ is in B or not. If it is in B, then we increment the number of times we have found $A[j]$ in B which is just i++,

and increment where we are in A which is just j++. Also, at this case, we know $A[j] \in A \cap B$. If it is not in B, then we only increment where we are in A which is just j++. In addition, since A and B do not contain repeated values, the values in $A \cap B$ are distinct. Since A and B are already sorted, we know that $A[j+1] > A[j]$, so each time when we increment i, we know the corresponding $A[j]$ represents the i-th smallest number of the intersection.
Termination:

When $i = k$, we know that we have already found common numbers in both A and B k times because we only increment i when we have found $A[j]$ in array B. So i has been incremented for k times. Since j was already incremented in the last time, we need $j - 1$ in order to find the k-th smallest number.
Therefore, A[j-1] is the k-th smallest entry of $A \cap B$.

Running time analysis:

We know that merge sort is $O(n \log n)$. Then each binary search is $O(\log n)$ and we would at most perform n times because we can assume $n \leq k$ and there are only n numbers in A and B, which gives us $O(n \log n)$. Therefore, the whole algorithm's running time is $O(n \log n) + O(n \log n) = O(n \log n)$.

2 Problem 2

Algorithm:

Basically, it utilizes modified binary search. We assume there is no repetitive elements in A and B.

1. First, if $A[k] < B[0]$, then $A[k]$ is the answer. If $B[k] > A[0]$, then $B[k]$ is the answer.

2. Then, let's consider more complex situation. We select $A[\frac{n}{2}]$ whose index is i. We then choose $B[k\text{-}i]$ whose index is j.

3. Then, we compare them. If $B[j] < A[i] < B[j+1]$ or $A[i] < B[j] < A[i+1]$, then we know $A[i]$ is the smallest k-th number or $B[j]$ is the smallest k-th number.

4. If the above condition does not satisfy, we keep using binary search. If $A[i]$<$B[j]$, then this means the smallest k-th number could be in the sub-array of $A[i+1]...A[n]$. So we perform recursion on this sub-array part. On the other hand, if $A[i]$>$B[j]$, this means the smallest k-th number could be in the sub-array of $A[0]...A[i]$. So we perform recursion on this sub-array part. Keep in mind that, each time when we do recursion, we need to change the value of j so that $i + j = k$.

Here is the pseudo code:

Algorithm: FindKthSmallest(A, B, k)
  if($A[k] < B[0]$) return $A[k]$;
  if($B[k] > A[0]$) return $B[k]$;
  return Search(A,B,0,n)

Algorithm: Search(A, B, p, q)
  i = $\frac{p+q}{2}$, j = k-i;
  if ($B[j] < A[i] < B[j+1]$) return $A[i]$;
  if ($A[i] < B[j] < A[i+1]$) return $B[j]$;
  if ($A[i]$<$B[j]$) return Search(A,B, $\frac{p+q}{2}$+1, q);
  if ($A[i]$>$B[j]$) return Search(A,B, p, $\frac{p+q}{2}$)

Correctness proof:

The first two cases are intuitive. If $A[k] < B[0]$, this means there isn't any number in B that is smaller than $A[k]$ since B is in ascending order. This means $A[k]$ is the k-th smallest since it's the k-th number in A. The same proof can be applied to the case of $B[k] > A[0]$.

Then, let's look at the next two cases. Because we set i+j=k and there are no repetitive elements in A and B, we know that at any time, all the elements in $A[0]..A[i]$ and $B[0]..B[j]$ are distinct. Also, we know there are k numbers in total in $A[0]..A[i]$ and $B[0]..B[j]$.

If ($B[j] < A[i] < B[j+1]$), we know that $A[i]$ is the k-th element. Here is the reason. $A[i]$ is the last and largest number of the portion of A we are considering and $B[j]$ is the last and largest number of the portion of B we are considering. These two portions have k numbers in total. Also, since $A[i]$ is in the middle of $B[j]$ and $B[j+1]$, this means $A[i]$ is the largest number of all k numbers because it is already larger than the largest number in the part of B we are considering, which gives us the k-th smallest number of the union. It's the same thing for the case when $(A[i] < B[j] < A[i+1])$ where $B[j]$ is just the largest number of all k numbers.

Then let's consider the case of $A[i]<B[j]$. This means the largest number in $A[0]...A[i]$ is smaller than the largest number in $B[0]...B[j]$, which shows that there is still potential to find a even smaller number from $A[i+1]...A[n]$. That's why we do recursion on this part of the array.

On the other hand, let's consider the case of $A[i]>B[j]$, which shows that there is still potential to find a even smaller number from $A[0]...A[i]$. That's why we do recursion on this part of the array.

Running time analysis:

We can use recurrence to find $T(n)$.

$T(n) = T(\frac{n}{2})+C=T(\frac{n}{4})+2C=T(\frac{n}{8})+3C...=T(1) + C\log n$.

Therefore, the running time for this algorithm is $O(\log n)$.

3 Problem 3

I chose the one with $\frac{3}{2}n$ comparison.
Algorithm:

1. Create array B and C both of size $\frac{n}{2}$. Iterate through array A and each time compare $A[i]$ with $A[i+1]$ where 0 ≤i≤n-1. The larger one will be added into B and smaller one will be added into array C. Then increment i by 2. Keep doing so until reaching the end of A.

2. Set variable max to $B[0]$. Iterate through B and find the largest number by replacing max with any number that's greater than it.

3. Set variable min to $C[0]$. Iterate through C and find the smallest number by replacing min with any number that's smaller than it.

4. We have found max and min, so

$$r = \max_{1 \leq i,j \leq n} |a_i - a_j| = max - min$$

Correctness proof:

All we need to do for this problem is to find the max and min in array A. As long as they are correct, the result is correct because $max - min$ gives the maximum difference. Array B contains all the larger numbers in the comparisons and array C contains all the smaller numbers in the comparisons. So we need to prove that max and min are found correctly.

Each time when we compare $A[i]$ with $A[i+1]$, if $A[i]$<$A[i+1]$, this means $A[i]$ can't be the max because there is already some number larger than it but it can be a potential min; $A[i+1]$ can't be the min because there is already some number smaller than it but it can be a potential max. Then we would put $A[i]$ into B and $A[i+1]$ into C. In the end, we know B contains all the possible max values and C contains all the possible min values, so we iterate through each to find max and min. Therefore, max and min are found correctly.

Running time proof:

Step 1 is $\frac{n}{2}$ comparisons because we look at 2 numbers at a time. Step 2 and 3 are each $\frac{n}{2}$ comparisons since both B and C have $\frac{n}{2}$ numbers. Therefore, summing it up will give up $\frac{3}{2}n$.

4 Problem 4

1.
Here is the pseudo code for insertion sort:
Algorithm: Insertion Sort(A)

```
For j = 2 to n
    key = A[j]
    i = j − 1
    while ((i > 0) and (A[i] >key))
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1]=key
```

The insertion sort algorithm iterates through array A which takes $O(n)$ which is demonstrated by the for loop. In each iteration, we swap $A[i]$ with $A[i+1]$ if $A[i] > A[i+1]$. Based on the definition of inversion, as long as $i < j$ and $A[i] > A[j]$, we count it as an inversion. This means we only swap if there is an inversion because we know $i + 1 > i$ but $A[i] > A[i + 1]$. Therefore, the maximum number of swaps when traversing is m because we know there are at most m inversions in array A. Denote this as:
$\sum$(All swaps)=m.
Therefore, this algorithm will at most take number of times of the traversal + total number of swaps = $O(n)$+$O(m)$=$O(n + m)$ steps.

2.
The maximum possible number of inversions is $\frac{n*(n-1)}{2}$. The worst case is reserved sorted array. For the first number, there can be n-1 inversions because it can be inversion with any other number. For the second number, there can be n-2 inversions. It's one less than the first number's inversions because we can't double count that first and second are inversions. So, the summation would be n-1 + n-2 +...+1 which gives us $\frac{n*(n-1)}{2}$.