CS515 - Algorithms & Data Structures

Practice Assignment 2

Vy Bui - 934370552

Instructor: Professor Glencora Borradaile

The School of Electrical Engineering and Computer Science
Oregon State University

Problem 1
A fixed point of an array A[1..n] is an index i such that A[i] = i. Given a sorted array of distinct integers A[1..n] as input, give a divide-and-conquer algorithm to determine if A has a fixed point that runs in time $O(logn)$.

Description Let $FP(i, j)$ be the function that checks if there exists a fixed point in $A[i, j]$. Imagine $A[k]$ as a discrete function of index k with $i \geq k \geq j$, then the fixed point of A is the intersection between $A[k]$ and the identity function $I[k]$. Because A is a distinct increasing array of integer, the rate of change of A at each index k must be greater than or equal to $I[k]$, which is equal 1 for all k. Therefore, $A[k]$ can only intersect $I[k]$ if $A[i] \leq i$ and $A[j] \geq j$.

Recurrence Let $m = \frac{i+j}{2}$, the recursive formulation can be described as follows

$$FP(i,j) = \begin{cases} False & i > j \\ \begin{cases} True & A[m] = m \\ FP(i,m) & A[m] > m \\ FP(m+1,j) & A[m] < m \end{cases} & otherwise \end{cases}$$

Pseudocode

Algorithm 1 $FP(i, j)$

```
if i > j then
    return False
end if
m ← (i + j)/2
if A[m] = m then
    return True
else if A[m] > m then
    return FP(i, m)
else
    return FP(m + 1, j)
end if
```

Proof of Correctness

Base Case: first, if A only has one element, if there exists a fixed point, then it has to be that element. Second, the left index i greater than the right index j indicates an empty array, which implies that no fixed point exists.

Inductive Hypothesis: $FP(i, m)$ and $FP(m + 1, j)$ correctly determine if $[i, m]$ and $A[m + 1, j]$ contain a fixed point, respectively.

Inductive Step: It is trivial that if at least one of $FP(i, m)$ and $FP(m+1, j)$ is true, then $FP(i, j)$ is true because they use the same indices and values of A.

Runing Time Analysis

On each recursive call, the algorithm splits the problem into two roughly equal halves and only solves one of them. It takes constant time to check if the midpoint is the fixed point. Therefore, the total running time of this algorithm is $T(n) = T(\frac{n}{2}) + O(1) = O(logn)$.

Problem 2
For a sequence of n numbers $a_1, .., a_n$, a *significant inversion* is a pair (a_i, a_j) such that $i < j$ and $a_i > 2a_j$. Assuming each of the numbers a_i is distinct, give an $O(nlogn)$ time algorithm to count the number of significant inversions in a sequence. (Hint: modify merge sort.)

Description We solve this problem by first sort the array, then we iteratively do binary search for the smallest possible significant inversion on the rest of the array. In particular, for each a_i, we search for $2a_i + 1$ on $A[i + 1 : n]$. If we can find such a number $A[k]$, then there are $(n - k + 1)$ elements in that can make a significant inversion with a_i. If such a number does not exist in A, we check the last element that binary search checks. This number must be either the largest number that is less than $2a_i + 1$ or the smallest number that is greater than $2a_i + 1$. If it is the former then all numbers following it in A must be greater than $2a_i + 1$, which can make up a significant pair with a_i. We can modify binary search to return the last number that was checked.

Let $SI(A)$ denotes the function that counts the number of significant inversions in A, and let $BS(i, j, t)$ denote the function that can find the minimum number that is greater than or equal to target t. Assume that A is sorted, for each a_i, we can use BS to search for $2a_i + 1$, and then count the numbers of the following numbers in A. The final result is the sum of all counts.

Pseudocode

Algorithm 2 $CSI(A[1, n])$

```
SortA
count ← 0
for i : [1, n − 1] do
    target = 2A[i] + 1
    k = BS(i + 1, n, target)
    if A[k] >= target then
        count ← count + n − k + 1
    else
        count ← count + n − k
    end if
end for
    return count
```

Proof of Correctness

First, prove that the modified $BS(l, r, t)$ correctly finds t, or the largest number that is less than t, or the smallest number that is greater than t. It returns the index of one of these numbers.

Base case: if the mid point is equal to t, it is obviously the number that we are searching for. If $l == r$, the array has only one number, which must fall in one of the cases listed above. Therefore the algorithm works correcly on the base case.

Inductive Hypothesis: assume that, for all l, r such that $r - l < k$, BS can correctly find one of the numbers of interest.

Inductive Step: we will prove that BS will work on $r - l = k + 1$. Similar to binary search, if $m < t$, then

Algorithm 3 $BS(l, r, t)$

```
if l == r then return i
end if
m ← (l+r)/2
if t = A[m] then
    return m
else if t < A[m] then
    return BS(l, m − 1, t)
else
    return BS(m + 1, r, t)
end if
```

we can rule out left half L of the array because all elements of L are less than the mid point. Then we only need to search for t in R by calling $BS(m+1, r, t)$, which, by the inductive hypothesis, will return the correct index because $r - (m+1) < k$. The case where $m > t$ is analogous. Proof completes!

Extra work to compute final solution: for each a_i, CSI counts the maximum significant inversions that a_i can make up. Notice that CSI does not count the elements that it already checks. The sum of all these counts is obviously the final solution.

Runing Time Analysis

We can use merge sort or quick sort to sort A in $O(nlogn)$ time. BS takes $O(logn)$ to search each time. Because it is called $O(n)$ times, these calls take $O(nlogn)$. Therefore, the runtime of the entire algorithm is $O(nlogn)$.

Problem 3
You are given two sorted arrays of size m and n. Give an $O(logm+logn)$ time algorithm for computing the k-th smallest element in the union of the two arrays.

Description

Let us call the two sorted arrays A and B with i and j as their indices, respectively. We can apply binary search on both arrays to find two points $1 \leq i \leq m$ and $1 \leq j \leq n$ such that $i + j = k$. Then the k-th smallest element in the union of the two arrays will be $max(A_i, B_j)$.

To begin with, i and j are the midpoints of the subarrays of interest of A and B. The algorithm starts off considering the entire two arrays and assigns their midpoints to i and j. Given i and j, we know that $G = max(A_i, B_j)$ is the $(i + j)^{th}$ element of the union because G is larger than all elements on the left of A_i and those of B_j. When $i + j > k$, we eliminate the right half of the subarray that contains $max(A_i, B_j)$. On the other hand, if $i + j < k$, we eliminate the left half of the subarray that contains $S = min(A_i, B_j)$ by half.

Pseudocode

Algorithm 4 $FK(A[1 : m], B[1 : n]), k$

```
al, ar, bl, br ← 1, m, 1, n
while al < ar ∨ bl < br do
    i ← (al + ar)/2
    j ← (bl + br)/2
    if i + j == k then return max(A[i], B[j])
    else if i + j < k then
        L ← min(A[i], B[j])
        if L == A[i] then
            al ← i
        else
            bl ← j
        end if
    else
        G ← max(A[i], B[j])
        if G == A[i] then
            ar ← i
        else
            br ← j
        end if
    end if
end while
```

Proof of Correctness

To prove that this algorithm, we will prove that it always correctly eliminates numbers that are not the target. In the first case when $i + j < k$, assume that $A_i < B_j$, then the union $U = A[1 : i] \cup B[1 : j] =$

$..., A_i, ..., B_j....$ Therefore, $A[1 : i-1] < A_i < target$ can be safely removed from consideration. In the second case when $i + j > k$, assume that $A_i < B_j$, then the union $U = A[1 : i] \cup B[1 : j] = ..., A_i, ..., B_j....$ Therefore, $B[j + 1 : m] > B_j > target$ can be safely removed from consideration. The algorithm will terminate because it eliminates half of one of the subarrays at each iteration.

Runing Time Analysis Because the algorithm eliminates half of one of the two arrays at each iteration, it takes $O(logm + logn)$ to eliminate all elements from both arrays.

Problem 4
You are given an $n \times n$ matrix A[1..n, 1..n] where all elements are distinct. We say that an element A[x] is a *local minimum* if it is less than its (at most) four neighbors, i.e. its up, down, left and right neighbors. Give an $O(n)$ time algorithm to find a local minimum of A.

Description The idea is to narrow down the position of the local minimum by elimination. First, We divide the matrix into 4 parts by a cross $A[1:n, n/2]$ and $A[n/2, 1:n]$. Then we search for the smallest element s in the cross. If s is the center of the cross then it is also the local minimum. Otherwise, if both of its neighbors are greater, then s is also a local minimum. Other than those cases, we find the quadrant that contains s's smallest neighbor, and recursively apply the algorithm on it until a local minimum is found.

Pseudocode

Algorithm 5 $LM(A[1:n, 1:n]), k$

```
xmin ← 1
xmax ← n
ymin ← 1
ymax ← n
while xmin < xmax ∨ ymin < ymax do
    xmid ← (xmax − xmin)/2
    ymid ← (ymax − ymin)/2
    minCell ← MAXINT
    minRow ← xmid
    minCol ← ymid
    for A[i, j] ∈ A[m, 1 : n] ∪ A[1 : n, m] do
        if a < minCell then
            minCell ← a
            minRow ← i
            minCol ← j
        end if
    end for
    if minRow == xmid ∧ minCol == ymid then return A[minRow, minCol]
    end if
    minNeighbor ← MinNeighbor(minRow, minCol)
    Update xmin,xmax,ymin,ymax to those of the quadrant that contains minNeighbor
end while
```

Proof of Correctness

The quadrant that contains the minimum neighbor must also contain a local minimum because if we follow a downward path from s to its minimum neighbor and so on, we eventually reach a cell c where we cannot move to another cell because $c < s$, which is smaller than all cells on the cross. In other words, the cross is the boundary of the quadrant. If we recursively drawing crosses to divide the matrix, eventually the local minimum will lie on the cross.

Runing Time Analysis This algorithm takes $2n + 2\frac{n}{2} + 2\frac{n}{4} + ... = O(n)$ (geometric series) time to find the minimum number in the crosses. It takes constant time to check cases and find the smallest neighbor of s. Totally, the algorithm takes $O(n)$.