CS5800 Algorithms *Out: September 20, 2019*

Solutions for Problem Set 2

Name: Instructor's Solutions *Due: September 27, 2019*

Problem 1 (Solving recurrences) 10+5

Solve the recurrence $T(n) = 2T(\sqrt{n}) + 1$ first by directly adding up the work done in each iteration and then using the Master Theorem.

Solution:

Work done in each iteration:
The number of problems at each level is doubling, while the work done in each problem is always equal to 1. Thus the i-th level will have 2^i total work. Since we are reducing our problem size by taking repeated square roots, the depth of the tree will be $\log\log n$ (because we want k s.t. $n^{1/2^k} = 2$)[1]. Thus the total work done is:

$$\sum_{i=0}^{\log\log n} 2^i = 2^{\log\log n+1} - 1 = 2\log n - 1 = \Theta(\log n)$$

Master's Theorem:
Use the substitution $n = 2^m$ to change the recurrence to be $T(2^m) = 2T(2^{m/2}) + 1$. Let us write $S(m) = T(2^m)$ which would give us a new recurrence $S(m) = 2S(m/2) + 1$. We can now use the Master's Theorem with $a = 2, b = 2, f(n) = 1$. This gives us $S(m) = \Theta(m)$ or using the earlier substitution in reverse, $T(n) = \Theta(\log n)$. ■

Problem 2 (Recursion) 5+10

Consider the following recursive function $\ell(n)$:

$$\ell(n) = \begin{cases} 0 & \text{if } n = 0 \\ \ell(n/2) & \text{if n is even} \\ 1 + \ell((n-1)/2) & \text{if n is odd} \end{cases}$$

1. Describe what the function $\ell(n)$ is calculating. Your description should make sense to a computer scientist, but be independent of the recursive definition.

2. Give a recurrence relation for the run time of $\ell(n)$, and solve it to get the run time of $\ell(n)$.

[1]We want the first time that the floor of the repeated square rooting gets to 1, which is the same as the actual square root going to 2 (or some number less than 4). Another way to think of this is that if you are halving the number n, you get to 1 in $\log n$ steps but if you halve the exponent of n then you will get to 1 exponentially faster or in $\log\log n$.

Solution:

1. It is counting the number of 1's in the binary representation of n. If the number is 0, then there are no 1's in its binary representation so $\ell(n) = 0$. Iteratively/recursively you will take the last bit of the binary representation and add that to your count. If the number if odd, then the last bit is 1 and otherwise it is 0. After you have added this, you "remove" that bit by dividing by 2, which is simply a right shift in the binary representation.

2. Let $T(d)$ be the run time of ℓ on an input with d bits. At each stage, we are halving the size of the input while doing constant work. Thus the recurrence relation is $T(d) = T(d-1) + 1$, which simplifies to $T(d) = d$. Since $d = \log n$, the running time of $\ell(n)$ is $\Theta(\log n)$.

$\blacksquare$

Problem 3 (Sorting special arrays) 3+4+10+3

Consider the problem of sorting an array $A[1,\dots,n]$ of integers. We presented an $O(n \log n)$-time algorithm in class and, also, proved a lower bound of $\Omega(n \log n)$ for any comparison-based algorithm.

1. Give an efficient sorting algorithm for a ***boolean***[2] array $B[1,\dots,n]$.

2. Give an efficient sorting algorithm for an array $C[1,\dots,n]$ whose elements are taken from the set $\{1,2,3,4,5\}$.

3. Give an efficient sorting algorithm for an array $D[1,\dots,n]$ whose elements are distinct ($D[i] \neq D[j]$, for every $i \neq j \in \{1,\dots,n\}$) and are taken from the set $\{1,2,\dots,2n\}$.

4. In case you designed linear-time sorting algorithms for the previous subparts, does it mean that the lower bound for sorting of $\Omega(n \log n)$ is wrong? Explain.

Solution:

1. (Boolean Array) Keep two counters c_0 and c_1 and sweep through the array. Whenever you see a 0 or False increment c_0 the counter for 0 and when you see a 1 or True increment c_1 the counter for 1. At the end, in the output array, make the first c_0 entries 0 and the next c_1 entries 1.

   Time complexity: $O(n)$ **Space complexity:** $O(1)$ for the two counters.

[2] In a boolean array $B[1,\dots,n]$, each element $B[i]$ (for $i = 1,\dots,n$) is either 0 or 1.

Algorithm 1 Sorting a Boolean array using Two Way Count Sort

```
procedure COUNT SORT($B[1, \ldots n]$)
    $counttrue \leftarrow 0, countfalse \leftarrow 0$
    for $i \leftarrow 1$ to $n$ do
        if $B[i] = true$ then
            $counttrue \leftarrow counttrue + 1$
        else
            $countfalse \leftarrow countfalse + 1$
        end if
    end for
    initialize $ret[counttrue + countflase]$
    for $i \leftarrow 1$ to $counttrue$ do
        $ret[i] \leftarrow true$
    end for
    for $i \leftarrow 1$ to $countfalse$ do
        $ret[i + counttrue] \leftarrow false$
    end for
    return $ret$
end procedure
```

2. (Sort when elements are from $\{1, 2, 3, 4, 5\}$) This is very similar to the boolean array algorithm. We keep 5 counters: c_1, c_2, c_3, c_4, c_5 and increment the counter i when you see element i. At the end make the first c_1 positions to be 1, c_2 to be 2 and so on.

   time complexity: $O(n)$ **space complexity:** $O(1)$ for the five counters.

Algorithm 2 Five Way Count Sort

```
procedure COUNT SORT($C[1, \ldots n]$)
    initialize $count[5]$ with all 0
    for $i \leftarrow 1$ to $n$ do
        $count[C[i]] \leftarrow count[C[i]] + 1$
    end for
    initialize $ret[1, \ldots, n]$
    $cur \leftarrow 1$
    for $i \leftarrow 1$ to 5 do
        for $j \leftarrow 1$ to $count[i]$ do
            $ret[cur] \leftarrow i$
            $cur \leftarrow cur + 1$
        end for
    end for
    return $ret$
end procedure
```

3. (Sorting n elements from $\{1, 2, \ldots, 2n\}$) This is similar to the previous problems, but instead of a constant number of possible elements each potentially occurring n times, we have n out of $2n$ elements each occurring at most once. We slightly modify the earlier algorithm and use a boolean

array B of length $2n$ initialized to all False. We then sweep the input and whenever we see element i, we set B_i to be True. At the end, we sweep B from the start and insert whatever element was True into the output array.

time complexity: $O(n)$ **space complexity:** $O(n)$.

Algorithm 3 n Way Count Sort

```
procedure COUNT SORT($E[1, \dots n]$)
    initialize boolean array $count[2n]$ with all $false$
    for $i \leftarrow 1$ to $n$ do
        $count[D[i]] \leftarrow true$
    end for
    initialize $ret[1, \dots, n]$
    $cur \leftarrow 1$
    for $i \leftarrow 1$ to $2n$ do
        if $count[i]$ is $true$ then
            $ret[cur] \leftarrow i$
            $cur \leftarrow cur + 1$
        end if
    end for
    return $ret$
end procedure
```

4. We did find linear time solutions to the previous parts. However, note that we didn't use any comparisons in those sorting algorithms. The lower bound of $\Omega(n \log n)$ is true only when the algorithm is a comparison based sorting algorithm. The lower bound theorem crucially relies on that assumption. Since that assumption is not valid for any of the sorts above, the lower bound theorem cannot be applied for those sorts. You can read up on Counting Sort for more details: `https://en.wikipedia.org/wiki/Counting_sort`

> In computer science, counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm, radix sort, that can handle larger keys more efficiently.
>
> Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the $\Omega(n \log n)$ lower bound for comparison sorting does not apply to it.

■

Problem 4 (Lower Bounds) 10

Show a $\Omega(\log n)$ lower bound for finding elements in a sorted array.

Solution:

Since we are finding a given element, we can assume that we would be using comparisons in our algorithm to make decisions. Hence, we can get a lower bound by modifying the comparison tree technique that was used to show that sorting has a $\Omega(n \log n)$ lower bound.

To be correct the algorithm must return one index corresponding to where the element is. That is, it needs to have one of n different outputs depending on the inputs. So our comparison tree will need n leaves. The height of this tree will be the worst case number of comparisons that any comparison-based algorithm will have to make. Since there are n leaves, we know that the height h will satisfy this inequality: $n \leq 2^h \implies h \geq \log n$. Thus this problem has a $\Omega(\log n)$ lower bound when using any comparison based algorithm. ■

Problem 5 (Programming: Counting threshold inversions) 40

You'll be given an array (of integers) and a threshold value as input, write a program to return the number of threshold inversions in the array. An inversion between indices $i < j$ is a threshold inversion if $a_i > t * a_j$, where t is the threshold value given as input.

Hint: Understanding counting inversions (normal inversions, without a threshold) will be useful to figure out how to do threshold inversions. We will be covering that in the recitations.

Solution:

The modification needed from normal inversion counting is that here the thresholded inversions would have to be counted first and then the actual merging process needs to be done. Refer to the two while loops in the Python solution's `mergeAndCount` procedure.

```
#include <bits/stdc++.h>

using namespace std;

int mergeprocedure(vector <long long int> &a, int astart, int alast, int bstart, int blast,
     int significant) {
    int count = 0, i = astart, j = bstart;
    int alen = alast - astart + 1;
    int blen = blast - bstart + 1;
    while (i <= alast && j <= blast) {
        if (a[i] > significant*a[j]) {
            count += alast - i + 1;
            j++;
        }
        else {
            i++;
        }
    }

    int k = astart, l = bstart;
    vector <int> ans;
    while (k <= alast && l <= blast) {
        if (a[k] < a[l]) {
            ans.push_back(a[k]);
            k++;
        }
        else {
            ans.push_back(a[l]);
            l++;
        }
    }

    while (k <= alast) {
        ans.push_back(a[k]);
        k++;
    }

    while (l <= blast) {
        ans.push_back(a[l]);
        l++;
    }

    copy(ans.begin(), ans.end(), a.begin() + astart);

    return count;
}


int countsiginv(vector <long long int> &a, int start, int last, int significant) {
    int count = 0;
    if (last == start || last - start < 0) {
        return 0;
    }
    int left = 0, right = 0, middle = 0;
    int m = ((start + last) / 2);
    left = countsiginv(a, start, m, significant);
    right = countsiginv(a, m+1, last, significant);
    middle = mergeprocedure(a, start, m, m+1, last, significant);
    return (left + right + middle) % 1000000007;
}

int main () {
    int n;
    int significant;
    cin>>significant;
    cin>>n;
    vector <long long int> a;
    long long int k;
    for (int i = 0;i<n;i++) {
        cin>>k;
        a.push_back(k);
    }

    cout<<"Ans : "<<countsiginv(a, 0, a.size() - 1, significant);
    return 0;

}
```

Listing 1: C/C++ solution

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class SolutionInversion {
```

```java

public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  int significant = sc.nextInt();
  sc.nextLine();
  int n = sc.nextInt();
  sc.nextLine();
  int[] a = new int[n];
  for (int i = 0; i < n; i++) {
    a[i] = sc.nextInt();
    sc.nextLine();
  }
  System.out.println(SolutionInversion.countsiginv(a, 0, a.length - 1, significant));
}

static int countsiginv(int[] a, int start, int last, int significant) {
  int count = 0;
  if (last == start) {
    return 0;
  }

  int left = 0, right = 0, middle = 0;
  int m = ((start + last) / 2);
  left = countsiginv(a, start, m, significant);
  right = countsiginv(a, m + 1, last, significant);
  middle = mergeprocedure(a, start, m, m + 1, last, significant);
  return (left + right + middle) % 1000000007;
}

private static int mergeprocedure(int[] a, int astart, int alast, int bstart, int blast,
    int significant) {
  int count = 0, i = astart, j = bstart;
  int alen = alast - astart + 1;
  int blen = blast - bstart + 1;
  while (i <= alast && j <= blast) {
    if (a[i] > significant * a[j]) {
      count += alast - i + 1;
      j++;
    } else {
      i++;
    }
  }
  int k = astart, l = bstart;
  List<Integer> ans = new ArrayList<>();
  while (k <= alast && l <= blast) {
    if (a[k] < a[l]) {
      ans.add(a[k++]);
    } else {
      ans.add(a[l++]);
    }
  }

  while (k <= alast) {
    ans.add(a[k++]);
  }

  while (l <= blast) {
      ans.add(a[l++]);
    }
    for (int m = 0; m < ans.size(); m++) {
      a[astart + m] = ans.get(m);
    }

    return count;
  }
}
```

Listing 2: Java solution

```python
t = int(input())
n = int(input())
l = list(map(int, input().split()))

def countInversions(l, t):
    if(len(l) <= 1):
        return l, 0
    else:
        mid = int(len(l)/2)
        left, a = countInversions(l[:mid], t)
        right, b = countInversions(l[mid:], t)
        result, c = mergeAndCount(left, right, t)
        return result, (a + b + c)

def mergeAndCount(left, right, t):
    result = []
    count = 0
    i,j = 0,0
    n, m = len(left), len(right)

    while(i < n and j < m):
        if(left[i] > t*right[j]):
            count += n-i
            j += 1
        else:
            i += 1

    i,j = 0,0
    while(i < n and j < m):
        if(left[i] <= right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result, count

_, count = countInversions(l, t)
print(count % (10**9+7))
```

Listing 3: Python solution

■