CS5800 Algorithms *Out: September 20, 2019*

Solutions to Problem Set 2

Name: Andrew Stefanich, Tushar Sharma, Chris Tozlowski *Due: September 27, 2019*

Group members: **** Write the names of all group members here. Use the submitter's name in the `\name{}` command. ****

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:

Iterative method:
$T(n) = 2T(\sqrt{n}) + 1$

$$T(n^{1/2}) = 2T(n^{1/4}) + 1$$

$T(n^{1/2}) = 2[2T(n^{1/2^2})] + 1$
$T(n^{1/4}) = 4[2T(n^{1/8}) + 1] + 2 + 1$
$T(n^{1/4}) = 8[T(n^{1/8})] + 4 + 2 + 1$

$T(n)$ is a GP series with common ratio of 2 with height of the tree 'k' terms.

We have to assume an input size of n of the form 2^k.
Therefore, T(n) is doing same work as sum of the series: 1+2+4+8+.....+2^k
$T(n) = \frac{1-2^{k+1}}{1-2} = 2.2^k + 1 \sim O(2^k)$ where k is total levels in the recursion tree.
at base case, we need n=2 at k = 1
at the final step of recursion the input will become 2^kroot of n and has to be 2,

$$n^{\frac{1}{2k}} = 2$$

Taking power of 2^k on both sides,

$$n^{\frac{2k}{2k}} = 2^{2^{2k}}$$

Taking $\log_2$on both the sides,

$$log_2 n = 2^k.log_2 2$$

Taking log again,

$$log_2(log_2 n) = k$$

Substituing value of k = $log_2(log_2 n)$back in the recurrence relationship we get,

$$T(n) = 2.2^{log_2(log_2 n)} + 1$$

$$T(n) = 2.log_2 n + 1$$

$$T(n) = \Theta(lgn)$$

Master method:
$T(n) = 2(\sqrt{n}) + 1$
Let $n = 2^k \quad \longrightarrow \quad k = log_2 n$
$T(2^k) = 2T(2^{\frac{k}{2}}) + 1$
Let $S(k) = T(2^k)$
$S(k) = 2S(\frac{k}{2}) + 1$
$a = 2, b = 2, f(k) = 1$
$f(k) = O(n) \quad \longrightarrow \quad S(k) = \Theta(k)$
$S(k) = T(2^k) = T(n)$
$S(k) = \Theta(k) \quad \longrightarrow \quad T(n) = \Theta(lgn)$

■

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.

   Solution:

   This procedure is calculating the number of 1's in the binary form the number 'n' or the hamming weight of the number 'n'.

   ■

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

   Solution:

$$T(n) = T(\frac{n}{2}) + 1$$

$$a = 1, b = 2, f(n) = 1$$

$$n^{log_2 1} = n^0 = 1$$
$$n^0 = \Theta(f(n))$$

$$T(n) = \Theta(n^0 lgn)$$
$$T(n) = \Theta(lgn)$$

■

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

Consider the problem of sorting an array $A[1, ..., n]$ 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***[1] array $B[1, ..., n]$.

Algorithm 1 sort

```
procedure BOOLEANSORT(B)
    varF ← 0                                   ▷ counter for false occurrences
    for i ← 0 to B.length − 1 do
        if B[i] == 0 then
            F = F + 1
        end if
    end for
    for i ← 0toB.length − 1 do
        if i < F then
            B[i] = 0
        else
            B[i] = 1
        end if
    end for
end procedure
```

$n(1 + 1) + n(1 + 1)$
$2n + 2n$
$4n$
$O(n)$

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

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

Algorithm 2 sort

```
procedure COUNTINGSORT(C)
    LET k = 5                                          ▷ range of input set
    LET n = C.length
    LET A =Array of length k
    for i ← 1 to n do                                  ▷ store count of each element
        A[C[i]] = A[C[i]] + 1
    end for
    for i ← 2 to k do                                  ▷ store position for use in output array
        A[i] = A[i] + A[i − 1]
    end for
    LET B = Array of length n                          ▷ build output array
    for i ← (n − 1) down to 0 do
        B[A[C[i]] = C[i]
        A[C[i]] = A[C[i]] − 1
    end for
end procedure
```

$1 + 1 + n + n + k(1 + 1) + n + n(1 + 1)$
$O(n + k)$
$k = 5$ for this instance, countingSort = O(n)

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

   Counting sort runs in linear time when $k = O(n)$, where k is the maximum value in the input array. Distinct elements and not knowing k will result in memory overhead, some of the benefit of counting sort will be compromised. However, since k is 2n, the linear time upper bound holds, we do not consider space efficiency at this time (especially in comparision to similar algorithms like radix sort).

Algorithm 3 sort

```
procedure COUNTINGSORT(D)
    LET n = D.length                          ▷ length of input array
    LET k = 2n
    LET A =Array of length k
    for i ← 1 to n do                         ▷ store count of each element
        A[C[i]] = A[C[i]] + 1
    end for
    for i ← 2 to k do                         ▷ store position for use in output array
        A[i] = A[i] + A[i − 1]
    end for
    LET B = Array of length n                 ▷ build output array
    for i ← (n − 1) down to 0 do
        B[A[C[i]] = C[i]
        A[C[i]] = A[C[i]] − 1
    end for
end procedure
```

$O(n + k)$ (shown in previous problem)
$k = 2n$
$O(n + 2n)$
$O(3n)$
$countingSort = O(n)$

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:

The lower bound for sorting of $\Omega(nlgn)$ only holds true for comparison based sorting algorithms. This is still a lower bound when we can't make any assumptions about the input elements other than they have some sort of total order. When we can make strong assumptions about the data, we can potentially manipulate elements in some way other than just comparisons (e.g. integers), then we no longer have a a lower bound of $\Omega(nlgn)$ but rather can potentially sort in linear time.

■

Problem 4 (Lower Bounds) 10

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

Solution:

If we have a list with 'n' elements, we can make a binary tree with total 'n' leaves. Then we will have to make 1 comparison at every level of the tree.

Assuming 'h' to be the total number of levels or height of the tree we get the following:

Maximum number of leaves = 2^h and we can also say, T(n) will be O(h) since there is 1 comparison at every level.

Using the idea of comparison trees for lower bounds, we can get the relationship between total number of leaves and minimum number of leaves we need.

Therefore,

$$2^h > total\_leaves > n$$

Taking $\log_2$ on each side,

$$h.log_2 2 > log_2 n$$

$$h > log_2 n$$

Looking at above relationship we can say that height of the tree which is also the total work done T(n) is lower bounded by $\log_2$n.

Therefore, for selection in a sorted array T(n) is Ω(log n).

■

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. document