Advanced Algorithms, Homework 2

Nathan Stouffer Kevin Browder

due: 3 September 2020

CSCI 432 Problem 2-1

Collaborators: *n/a*

In this class, we will assign groups for the group project. Now is your chance to weigh in on how we choose them!

1. Describe the problem of choosing the groups formally, including describing what the input and output is. Be sure to explain any properties of the output that are important (e.g., the groups are all of size three and everyone has the same number of characters in their first name).
2. Describe, in paragraph form, the algorithm you propose.
3. Provide this algorithm in the algorithm environment.
4. Prove that your algorithm terminates.

Answer

1. The input is an unsorted list of students, the output is a two dimensional list of students with the first dimension being the number of groups $\lceil n/3 \rceil$ and the second dimension being the number of students in each group (3). We then need to assign each student to a group. Not all groups may be of size 3, some may be smaller if $n \neq 0 \mod 3$
2. We propose a fairly simple algorithm. First we create a 2 dimensional list with length $\lceil n/3 \rceil$. If this number is not an integer we round up and will make a group of less than three. Next we will randomly scramble the input array of students. We will then iterate through the input array. The first student will be added to the first index in the 2 dimensional list. We will then increment to the second index and so on until we reach the end. We will then reset and begin at the beginning of the list again. The results of a class with 8 students would look like $\{\{***\},\{***\},\{**\}\}$ and everything will be choosen randomly.
3. Here is the algorithm in psuedocode.

```
Assign(A[1..n])
    A = RandomShuffle(A)
    int groups = ⌈n/3⌉
    groups = list[][]
    location = 0
    for i in A.length
        list[location].add(i)
      location++
   if (location == groups)
         location = 0
```

4. This algorithm at its most basic level only iterates through the array of students. An array cannot be infinite because each index in an array is assigned to a memory location. There is no such thing as infinite memory so therefore the length of an array is finite. Since we are only iterating over a finite array once. The algorithm must terminate.

CSCI 432 Problem 2-2

Collaborators: *Nathan Stouffer and Kevin Browder*

Chapter 1, Problem 37 (Largest Complete Subtree).

For this problem, a subtree of a binary tree means any connected subgraph. A binary tree is complete if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the largest complete subtree of a given binary tree. Your algorithm should return both the root and the depth of this subtree.

1. Describe the problem in your own words, including describing what the input and output is..
2. Describe, in paragraph form, the algorithm you propose.
3. Provide this algorithm in the algorithm environment.
4. What is the runtime of your algorithm?
5. Prove that the algorithm is correct.

Answer

1. The problem is to find the largest subtree of a binary graph recursively. The input of the algorithm is a binary tree and the output should be a subtree which is in the format of the root and the depth of the subtree.
2. The algorithm I propose is to do a post order traversal over very node n in the binary tree. At each node then recursively compute the depth of the sub tree on both the left and right side by counting the number of nodes down that have two children. Then take the smaller returned depth and add 1 (for the root node n). This is the depth of subtree. Keep track of the depths for each node as you do the post order traversal. Once you are finished simply choose the largest depth and return the depth and node.
3. We now give the algorithm in psuedocode.

```
Depth(n):
    if (right(n) == null or left(n) == null)
        return -1
    else
        RD = Depth(right(n)) // right child
        LD = Depth(left(n)) // left child
        return min{LD, RD} + 1

LST(T):
    maxRoot = 0
    maxDepth = 0
    post order traverse of $n$'s in T:
        if right(n) == null or left(n) == null
            depth(n) = 0
        else
            depth = min{CD(right(T), CD(left(T)))} + 1
            if depth > maxDepth
                maxDepth = depth
                maxRoot = n
```

4. For time complexity, the post order traversal will take O(n) because it has to traverse over every node. The depth function will never have to traverse the full tree so the time complexity for the algorithm is O(n).

5. First we set up the recursive invariant: The initialization step is checking if the parent node has children. The maintenance step is checking if both children have two children. This means that every time we make the recursive call the subtree is broken into two smaller subtrees. Using induction, we have the base case of either child is null. Our inductive hypothesis: We assume that the recursive invariant holds true for all previous recursive steps. As our inductive step, since we assume the inductive hypothesis, we need only check if both children have children. So, by induction, we have proven that the algorithm is correct.

CSCI 432 Problem 2-3

Collaborators: *Nathan Stouffer and Kevin Browder*

Chapter 1, Problem 9 (Pancakes). When describing your algorithm, please give a prose explanation (in paragraph form) as well as in the algorithm environment. To "Pass", we expect and answer to (a) and (b). To earn a "high pass" on this question, you must answer (c) as well.

Suppose you are given a stack of n pancakes of dierent sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a flip—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over

a. Describe an algorithm to sort an arbitrary stack of n pancakes using $O(n)$ flips. Exactly how many flips does your algorithm perform in the worst case?

b. For every positive integer n, describe a stack of n pancakes that requires $\Omega(n)$ flips to sort.

c. Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using $O(n)$ flips. Exactly how many flips does your algorithm perform in the worst case?

Answer

a. We begin with a prose explanation of the algorithm. The input to the algorithm is a permutation of $1, 2, 3, ..., n$ where k references the k^{th} largest pancake. The specific permutation is determined by the stack of pancakes (we order the numbers left to right as the pancakes are ordered top to bottom). Our goal is to produce $1, 2, 3, ..., n$ as output using only the flip operation $O(n)$ times.

Our algorithm first identifies the largest number k that is out of place, say k is i pancakes from the top. Then we flip the top i pancakes so that k is on top. At this point, we flip the top k pancakes to put k in the correct spot in the stack. We then repeat this process for every k.

We now give the pseudocode for this algorithm. Note that for $a \in \mathbb{N}$, IndexOf(a) returns the index referencing the value a and Flip(a) inserts the spatula at the a^{th} pancake from the top and flips the stack.

```
SortPancakes(order[1..$n$])
1. for $k \leftarrow n$ to 2
2.     $i \leftarrow$ IndexOf($k$)
3.     if $i < k$
4.         Flip($i$)
5.         Flip($k$)
```

In the very worst case, we must execute the flip operation twice every time the for loop runs: we must flip $2 * (n - 1)$ times.

b. To make sure that the algorithm runs in $\Omega(n)$ flips, we construct a starting state so that every pair of consecutive pancakes should not end up as consecutive pancakes in the final state.

c. We now devise an algorithm to also ensure that the burned side of a pancake is face down. We first identify the largest number k that is out of place and record its index i. Then we flip the top i pancakes. If the burned side is not up, we flip the top pancake. After that, flip the top k pancakes. We then repeat this process for every k.

We introduce the function BurnedUp(a) for $a \in \mathbb{N}$ which returns $True$ if the a^{th} pancake's burned side is facing up and $False$ if the a^{th} pancake's burned side is facing down.

```
SortBurnedPancakes(order[1..$n$])
1. for $k \leftarrow n$ to 1
2.     $i \leftarrow$ IndexOf($k$)
3.     Flip($i$)
4.     if not $BurnedUp$(1)
5.         Flip(1)
6.     Flip($k$)
```

In the worst case, we must execute the flip operation three times every time the loop runs: $3 * n$.