CS 330: Homework 3

Michael Lin

October 20, 2015

Problem 1

Using a tree structure, for UNION, keep track of the size of each tree and make the leader of the shallower tree as the leader of the deeper tree. For FIND, use path compression algorithm as shown in Erickson 17.3, where each call of FIND recursively set the parent of the parent as the leader. In essence, after enough calls (a constant number) of FIND, we will have a shallow tree with just one level below the leader.

Algorithm 1 Algorithm for Union-Find adapted from Erickson 17.3

```
1:  function UNION(x, y)
2:      if DEPTH(x) > DEPTH(y) then
3:          PARENT(y) = x
4:      else
5:          PARENT(x) = y
6:          if DEPTH(x) = DEPTH(y) then
7:              DEPTH(y) = DEPTH(y) + 1
8:  function FIND(x)
9:      if x ≠ PARENT(x) then
10:         PARENT(x) = FIND(PARENT(x))
11:     return PARENT(x)
```

Proof of correctness. UNION is correct because we simply point the leader of the shallower set (say leader A) to the leader of the deeper set (say leader B), with leader B being the new leader of the combined set. FIND is correct by Erickson 17.3. □

Proof of runtime. Each call of UNION consists of only 2 operations: determine larger set, point leader of one set to leader of the other set. So UNION has $O(1)$ amortized time.

We perform all UNION operations before FIND, which means any FIND operation will be on the same set.

To see the amortized analysis of FIND, we use charging method. Suppose we pay \$2 for each node when we first made each set. For FIND, we spend \$1 each time for compressing the path (i.e. point a node directly to the leader) and another \$1 each time to actually find the leader. When tree is deep, we will need to spend a large (but constant) amount of money to redirect all nodes to the leader, but we will have enough money since we paid for that already. After the tree becomes shallow, any subsequent calls for FIND will only require \$1 to find the leader since all nodes other than the leader point directly to the leader. In the long run, we will only need $O(1)$ time to run FIND, hence FIND also have $O(1)$ amortized time. □

Problem 2

Each POP and PUSH operation takes constant time; there are at most n buildings that can be popped onto or pushed off of the stack. Thus, the overall runtime of the algorithm is $O(n)$.

Alternatively, we can use charging method to see the amortized analysis of this algorithm. Suppose we give \$2 to each building before any operations, and we charge \$1 each time that a building is pushed onto

the stack and another \$1 each time that it's popped off the stack. Since each building is only pushed onto the stack once (after the while loop), we charge \$1 for each of the n buildings. As a result, we can pop off each building at most once, so at most we charge $\$n$ to pop off all the buildings. Together, we charge at most $\$2n$ for n buildings, thus an amortized runtime of $O(n)$.

Problem 3

We use a dynamic programming algorithm to compute the smallest vertex cover of tree. Let $C(u)$ be the size of the smallest vertex cover of a tree rooted at node u. In one of the base cases, where u is a leaf, $C(u) = 0$ since we can always choose the parent of the leaf (number of children always at least number of parent since only 1 parent). In another base case, where u is the parent of the leaf, $C(u) = 1$ since we choose this parent as the only node in the vertex set (this is because the number of leaves for a given parent is at least 1 and choosing this parent ensures the smallest vertex cover). If there are 2 nodes, choose one of the nodes as the "parent". If there is 1 node, $C(u) = 1$. For remaining cases,

$$C(u) = \min\{1 + \sum_{v \in \textsc{Ch}(u)} C(v), |\textsc{Ch}(u)| + \sum_{v \in \textsc{Gch}(u)} C(v)\}$$

where Gch finds all grandchildren of u and Ch finds all children of u. Finally, we return $C(s)$ where s is the root of the tree. To ensure fast computation time, we compute and memoize from bottom-up. First compute C for the leaves and saving those values, then compute C for the parents and saving those values, etc.

Proof of correctness. We prove by induction on the subtrees. The base case is a tree with a root v and its children as leaves, i.e. a tree with $k = 2$ levels.

1. If we choose $v \in V'$, then we add 1 to the size of our minimum vertex cover. For all $u \in \textsc{Ch}(v)$, $C(u) = 0$, so the sum is just 1.

2. If we choose $v \notin V'$, then we add number of children (at least 1 since parent must have at least 1 child) to the size of our minimum vertex cover. Since v has no grandchildren, we don't add anything more.

Option 1 would always be smaller than option 2 unless v only has one children, so the algorithm correctly returns 1.

Now suppose the algorithm is true for all $k < K$, i.e. $C(u)$ correctly computes the size of the minimum vertex cover for all trees with less than K levels rooted at node u.

Let V' be the minimum vertex cover set. Vertex v, which is the parent of all the u's and is the root of a K-level tree, is either in V' or it is not in V'.

1. If $v \in V'$, then size of V' increases by one. Then, we look at whether each child of v is in V'. Hence, the size of the minimum vertex cover of the subtree rooted at each child.

2. If $v \notin V'$, then all children of v must be in V', so we add the number of children to the size of V'. Then, we look at whether each grandchild of v is in V'. Hence, we also add the size of the minimum vertex cover of the subtree rooted at each grandchild.

Since each previous C value was correctly computed by induction hypothesis, taking the minimum of the above two options correctly returns the size of the smallest vertex cover of the tree rooted at v, i.e. the entire set. □

Proof of runtime. We can compute and memoize from bottom-up by starting our computations at the leaves and moving up toward the root. The C values for each node can be stored in a HashMap, and the C value for the parent of each node can be computed by retrieving the C value of all the children and grandchildren. For all the nodes, we sum the C values for both the children ($|E|$ calculations) and the grandchildren ($|E|$ calculations), so a total of $2|E|$ calculations. This means the algorithm takes $O(|E|)$ time, which for a tree is the same as $O(|V|)$ (since $|E| = |V| - 1$). Thus, the algorithm runs in linear time. □

Problem 4

(a) Algorithm for finding the largest sum of elements in a contiguous subarray of the array $A[1 \dots n]$. Using dynamic programming, we can find a recursive formula to compute the largest sum. Let $S(i)$ be the sum of a contiguous subarray that begins somewhere between $A[1]$ and $A[i]$, and ends on element $A[i]$. This sum is essentially the maximum contiguous sum that ends on $A[i]$. For the base case, $S(1) = A[1]$. Otherwise,

$$S(k) = \max\{S(k-1) + A[k], A[k]\}$$

To find our answer, simply return

$$\max_{1 \le k \le n} S(k)$$

Proof of correctness. The algorithm first finds the maximum contiguous sum that ends at $A[i]$ for all possible i. Certainly, this gives all contiguous sums that might be candidates for the maximum contiguous sum for the entire array. We can see this by induction on $S(k)$. By way that S is defined, $S(1) = A[1]$, which is indeed the largest contiguous sum that ends on $A[1]$. Suppose the recursive formula holds for all cases less than or equal to k. Now, to show that the formula holds for $k+1$, we first observe that $S(k+1)$ is either the largest contiguous sum that ends on $A[k]$ plus $A[k+1]$, or $A[k+1]$ itself. But the largest contiguous sum that ends on $A[k]$ is just $S(k)$, which means $S(k+1)$ is either $S(k) + A[k+1]$ or $A[k+1]$. To find out which one it is, we want the larger of the two because $S(k)$ is defined as the largest contiguous sum that ends at $A[k]$. Therefore, in mathematical notation, we have again

$$S(k+1) = \max\{S(k) + A[k+1], A[k+1]\}$$

For our answer, since we don't care where in the array our largest contiguous sum ends, we simply take the largest of all values of S by iterating through the array S. □

Proof of runtime. We have $O(n)$ different values of S to compute, and each computation only takes $O(1)$ time if we memoize all previous values of S (this can be done by computing $S(1)$, then $S(2)$, then $S(3) \dots$). Thus, to compute all values of S takes $O(n \times 1) = O(n)$ time. To find the maximum over all values of S, we need only iterate through all n values once by updating current maximum value. This part thus takes $O(n)$ time. Together, we have $O(n+n) = O(n)$. Therefore, the algorithm runs in linear time. □

(b) Algorithm for finding the largest product of elements in a contiguous subarray of the array $A[1 \dots n]$. Using dynamic programming, we can find a recursive formula to compute the largest product. Let $P(i)$ be the product of a contiguous subarray that begins at some element $A[j]$ between $A[1]$ and $A[i]$, and ends on element $A[i]$. Let $Q(i)$ be the product of a contiguous subarray that begins at some element $A[j']$ between $A[1]$ and $A[i]$, and ends on element $A[i]$. Define $P(i)$ as the maximum contiguous product that ends on $A[i]$, and define $Q(i)$ as the minimum contiguous product that ends on $A[i]$. For the base case, $P(1) = Q(1) = A[1]$. Otherwise,

$$P(k) = \max\{P(k-1)A[k], Q(k-1)A[k], A[k]\}$$

$$Q(k) = \min\{P(k-1)A[k], Q(k-1)A[k], A[k]\}$$

To find our answer, simply return

$$\max_{1 \le k \le n} P(k)$$

Proof of correctness. The algorithm first finds the maximum contiguous product, along with the minimum contiguous product, that ends at $A[i]$ for all possible i. Certainly, this gives all contiguous products that might be candidates for the maximum contiguous product for the entire array. We can see this by induction on $P(k), Q(k)$. By way that P, Q are defined, $P(1) = Q(1) = A[1]$, which is indeed, respectively, the largest and smallest contiguous product that ends on $A[1]$. Suppose the recursive formulas hold for all cases less than or equal to k. Now, to show that the formulas hold for $k+1$, we first observe

that $P(k+1)$ can be the largest contiguous product that ends on $A[k]$ times $A[k+1]$ (if both factors are positive), the smallest contiguous product that ends on $A[k]$ times $A[k+1]$ (if both factors are negative, since two negatives make a positive), or $A[k+1]$ itself (if both the maximum and minimum contiguous products are negative and $A[k+1]$ is positive). But the largest contiguous product that ends on $A[k]$ is just $P(k)$, and the smallest contiguous product that ends on $A[k]$ is just $Q(k)$, which means $P(k+1)$ can be $P(k)A[k+1]$, $Q(k)A[k+1]$, or $A[k+1]$. To find out which one it is, we want the largest of the three because $P(k)$ is defined as the largest contiguous product that ends at $A[k]$. Therefore, in mathematical notation, we have again

$$P(k+1) = \max\{P(k)A[k+1], Q(k)A[k+1], A[k+1]\}$$

The argument is similar for $Q(k)$, which is, again, the minimum contiguous product. For our answer, since we don't care where in the array our largest contiguous product ends, we simply take the largest of all values of P by iterating through the array P. □

Proof of runtime. We have $O(n)$ different values of P, Q to compute, and each computation only takes $O(1)$ time if we memoize all previous values of P, Q (this can be done by computing values of P, Q sequentially from 1). Thus, to compute all values of P, Q takes $O(2 \times n \times 1) = O(n)$ time. To find the maximum over all values of P, we need only iterate through all n values once by updating current maximum value. This part thus takes $O(n)$ time. Together, we have $O(n+n) = O(n)$. therefore, the algorithm runs in linear time. □

Problem 5

Using dynamic programming, we can find a recursive formula to determine whether it is possible to make change for v. Define the following function:

$$G(i) = \begin{cases} 1 & \text{if cannot make change for } i \\ 0 & \text{if can make change for } i \end{cases}$$

Note, this means $G(i) = 1$ for all $i < 0$ because negative currency values are not defined and thus cannot make change. For base cases, let $G(x_k) = 0$ for all $k \in [1, n]$ since any value with the same value as a coin can make change with that coin. For other cases,

$$G(j) = \prod_{k=1}^{n} G(j - x_k)$$

Each value of G should be calculated from bottom up, i.e. calculate G for small input values first. Any values of G already computed should be stored for access in subsequent calculations. Return $G(v)$ for solution.

Proof of correctness. Fact 1: Since we have an unlimited supply of coins, it is clear that change can be made for a value $v \geq 0$ if and only if change can also be made for a value $v - x_k \geq 0$ for some coin k. Fact 2: Certainly, if the currency value in question has the same value as a coin, then that value can make change by simply using that coin. The algorithm uses these facts to check if it is possible to make change for a currency value v. The base case in the algorithm is Fact 2, which sets $G(x_k) = 0$ for all $k \in [1, n]$, i.e. sets all coin values as possible to make change. All other cases are checked with Fact 1, which examines whether, by first returning one of any of the coins, any value would still be able to make change. Since we defined $G(i) = 0$ if we can make change for i, then the product of G's ensures $G(j)$ can be made into change if any of $G(j - x_k)$ can as well (multiplying any value by 0 gives 0). Thus, the algorithm is correct. □

Proof of runtime. If we use memoization for each value of G, it is clear that there would be v different values of G. This gives $O(v)$ values of G that need to be computed. For each value of G, we need to subtract all n coin denominations from the total value, which gives $O(n)$ operations. Multiplying the two parts gives a total runtime of $O(nv)$. □