Homework 5 sample solution

Due 10/10/2024

October 18, 2024

1. Give pseudocode for an algorithm to compute the number of *quadratic nonresidues* modulo n, where a quadratic nonresidue mod n is an integer r in the range of 0 to $n-1$ such that there is no integer x in this range where $x^2 \bmod n = r$.

   For example, the quadratic nonresidues of $n = 6$ are 2 and 5, as shown by the table below:

x	$x^2 \bmod 6$
0	0
1	1
2	4
3	3
4	4
5	1

   Thus, the solution for $n = 6$ is 2.

   Your algorithm should be as efficient as possible. Be sure to describe which data structure(s) you're using and why.

   There are many different possible solutions to this problem, though the most efficient ones are linear time.

   Using an array-based map:

```
Input: $n$: positive integer
Output: number of quadratic nonresidues mod $n$
Algorithm: QNR_map
$residue$ = Array($n$)
Initialize $residue[0..n-1]$ to false
for $i = 0$ to $n - 1$ do
    $residue[i^2 \bmod n]$ = true
end
$count = 0$
for $i = 0$ to $n - 1$ do
    if $\neg residue[i]$ then
        $count = count + 1$
    end
end
return $count$
```

Using a hash table:

```
Input: $n$: positive integer
Algorithm: QNR_hash
$hash$ = HashTable($n$)
$count = n$
for $i = 0$ to $\lfloor n/2 \rfloor$ do
    $res = i^2 \bmod n$
    if $\neg hash$.Search($res$) then
        $count = count - 1$
        $hash$.Insert($res$)
    end
end
return $count$
```

2. Dr. Snevets is working on an algorithm that uses a Union-Find to maintain a set partition, but her algorithm also needs to efficiently determine the smallest element in each partition. Answer the following about developing an improved Union-Find that efficiently supports a FindMin(x) operation that can return the minimum element in the same partition as x.

   (a) One way to implement this improved data structure is to store a third array min such that $min[r]$ is the minimum element in the tree rooted at r, for every root element r. (min can equal any value at indexes that are not roots.) Describe how to modify the pseudocode for the Union-Find initialize operation (below) to also initialize min appropriately.

```
    Input: $n$: size of the Union-Find to initialize
    Output: a Union-Find of size $n$ where every element
            points to itself
1 Algorithm: Union-Find.Initialize
2 $uf$ = Array($n$)
3 Initialize $uf$ to $1..n$
4 $size$ = Array($n$)
5 Initialize $size$ to 1
6 return $(uf, size)$
```

Initialize *min* to be an array containing the numbers 1 to n (same as uf) and return all 3 (uf, *size*, and *min*).

(b) What is the worst-case time complexity for your modified Initialize algorithm? Justify your answer.

It takes $\Theta(n)$ time: the original Initialize takes $\Theta(n)$, and initializing *min* also takes $\Theta(n)$, for a total of $\Theta(n)$.

(c) Describe how to modify the Union operation (below) so that *min* always stores the minimum element of the tree at every root (*min* can store anything at an index that is not a root of the Union-Find.)

```
    Input: $(uf, size)$: the Union-Find to modify
    Input: $a$: index of one element to union
    Input: $b$: index of another element to union
    Output: modified $uf$ that efficiently merges the trees
            containing $a$ and $b$
1 Algorithm: $uf$.Union
2 $ra = uf$.Find($a$)
3 $rb = uf$.Find($b$)
4 if $size[ra] > size[rb]$ then
5   | Swap $ra$ and $rb$
6 end
7 $uf[ra] = rb$
8 $size[rb] = size[ra] + size[rb]$
```

At the end (or at least after line 7), set $min[rb] = \min(min[ra], min[rb])$ (or set $min[rb] = min[ra]$ if $min[ra] < min[rb]$).

(d) What is the amortized time complexity of your modified Union algorithm? Show your work. Note that the Find operation does not need to be changed from the optimized Union-Find presented in class.

The existing optimized union operation takes $\Theta(\alpha(n))$ amortized time, and our modification only takes $\Theta(1)$, so the modified algorithm will take $\Theta(\alpha(n))$ amortized time.

3. Find the worst-case complexity of the hierarchical clustering algorithm below. You may assume that the distance function takes $\Theta(1)$ time to compute.

```
   Input: $data$: set of data points
   Input: $n$: size of $data$
   Input: $distance$: distance function that takes two data points and
          returns a nonnegative real number
   Input: $c$: desired number of clusters; must be an integer between 1 and
          $n$
   Output: single-linkage hierarchical clusters for $data$
1  Algorithm: SingleHClust
2  $heap$ = MinHeap()
3  for $i = 1$ to $n - 1$ do
4      for $j = i + 1$ to $n$ do
5          Insert $distance(data[i], data[j])$ into $heap$, along with the
            corresponding indexes $i$ and $j$
6      end
7  end
8  $uf$ = UnionFind($n$)
9  $count = n$
10 while $count > c$ do
11     $(i, j, dist) = heap$.DeleteMin()
12     if $uf$.Find($i$) $\neq$ $uf$.Find($j$) then
13         $uf$.Union($i$, $j$)
14         $count = count - 1$
15     end
16 end
17 return $uf$
```

Line 2 takes $\Theta(1)$ time. Line 5 takes $O(\lg n)$ time, as there will be at most $\binom{n}{2} = O(n^2)$ elements in the heap and $\lg(n^2) = O(\lg n)$. The inner for loop iterates $n - i$ times, for a total time of $O((n - i)\lg n)$. Since this complexity depends on i, the outer loop takes a total time of

$$\begin{aligned}
\sum_{i=1}^{n-1} O((n-i)\lg n) &= O\left(\lg n \sum_{i=1}^{n-1} n - i\right) \\
&= O\left(\lg n \sum_{i=1}^{n-1} i\right) \\
&= O(\lg n \Theta((n-1)^2)) \\
&= O(n^2 \lg n)
\end{aligned}$$

Line 8 takes $\Theta(n)$ and line 9 takes $\Theta(1)$. Line 11 takes $O(\lg n)$ time (for the same reason as line 5). Lines 12 and 13 take $\Theta(\alpha(n))$ time, while line 14 takes $\Theta(1)$, so each iteration of the while loop takes $O(\lg n) + \Theta(\alpha(n)) + \Theta(1) = O(\lg n)$ time. In the worst case, *count* is rarely decremented, causing us to iterate through most of the $\binom{n}{2}$ elements of *heap*, for $O(n^2)$ iterations. Thus, the while loop would take $O(n^2 \lg n)$ time in worst case. Line 17 takes $\Theta(1)$, so the total time for the algorithm is $\Theta(1)+O(n^2 \lg n)+\Theta(n) + \Theta(1) + O(n^2 \lg n) + \Theta(1) = O(n^2 \lg n)$.

Technically, the size of the heap changes as we add and delete elements, so we should compute the duration of the nested for loops in lines 2–6 as $\sum_{i=1}^{\binom{n}{2}} O(\lg(i))$. This sum, though, has $\binom{n}{2}/2 = \Theta(n^2)$ elements that are at least $O(\lg(n/2)) = O(\lg n)$, so it adds up to $O(n^2 \lg n)$ either way. Similarly, the cost for deleting an element is $O(\lg(n/2))$ "on average," which is logarithmic, so the total cost for the while loop will also be $O(n^2 \lg n)$.