CS2800: Design and Analysis of Algorithms

Mini Quiz #3

April 21, 2023

Max Marks: 10

Duration: 1 hour

Instructions

- Write your name and roll number clearly in the answer sheet before you start.
- This quiz contains 3 questions totalling 10 marks. All questions are compulsory.
- Please be as precise as possible in your answers. Write only what is necessary to substantiate your answer.
- Anything used in class can be used as is. If you are using a result that was given in the problem set directly, you can mention it and use it. Anything else must be fully justified.
- Write only one answer for a question. If you make multiple attempts, please strike off the attempts that you think are incorrect.
- Whenever you describe an algorithm, you must include a proof of its correctness and an analysis of its running time. **Algorithms/pseudocode without any explanation will receive partial/zero marks.**

---

1. **(3 marks)** Consider the following sequence of operations that are performed on a disjoint-set data structure that uses union-by-rank with path compression. Draw the trees after each operation and write the down the ranks of each of the nodes after the final operation.

   MakeSet(1), MakeSet(2), MakeSet(3), MakeSet(4), MakeSet(5), MakeSet(6), MakeSet(7), MakeSet(8), MakeSet(9), MakeSet(10), Union(1, 2), Union(2, 3), Union(3, 10) Union(4, 5), Union(6, 7), Union(5, 7), Union(8, 9), Union(7, 9), Union(3, 9), Find(10)

   Solution: Multiple answers possible. Point to note is that the union operation has a find operation within it, and this find also performs path compression.

2. **(4 marks)** A *quack* is a data structure that allows pushes, pops and dequeues in the following way: You can think of a quack as a list written from left to right such that the following operations are permitted:

   - qPush(x): add item x to the left end of the quack.
   - qPop: remove and return the item from the left end of the quack.
   - qDequeue: remove and return the item from the right end of the quack.

   Describe how to implement a quack using three stacks so that the amortized cost of all the operations is $O(1)$.

   Each element of the quack must be stored in at most one of the three stacks at any point of time, and you can access the stacks only through the standard Push and Pop operations.

   Hint: Use two stacks to split the list amongst themselves. Use the third stack as an auxilliary stack to perform the earlier operation. Do the accounting accordingly.

Solution: Let S_1, S_2, and S_3 be the three stacks, where we maintain S_3 as an auxilliary stack for moving elements between S_1 and S_2. The left end of the quack is the top of S_1 and the right end of the quack is the top of S_2. The operations are performed as follows:

- qPush(x): Push x on to S_1
- qPop: If S_1 is non-empty, pop from S_1. Otherwise, reorder the stacks to divide the elements equally into S_1 and S_2 using S_3 so that the left end is on top of S_1 and the right end is on top of S_2. You can do this by pushing half the elements of S_2 on to S_3, pushing the other half on to S_1, and then pushing all the elements from S_3 back on to S_2. Now pop from S_1.
- qDequeue: If S_2 is non-empty, pop from S_2. Otherwise, reorder the stacks to divide the equally into S_1 and S_2 using S_3 so that the left end is on top of S_1 and the right end is on top of S_2. You can do this by pushing half the elements of S_1 on to S_3, pushing the other half on to S_2, and then pushing all the elements from S_3 back on to S_1. Now pop from S_2.

To prove that the amortized cost of these operations is $O(1)$, consider the following accounting strategy: each of the operations pay 4 units.

The only costly operation is the movement of half the elements of one stack into another stack via S_3. We want to show that it has already been paid for through the qPush/qPop/qDequeue operations done earlier.

- qPush: This has worst-case $O(1)$-time.
- qPop: The only costly operation is when S_1 is empty, and this could have only happened because all the elements of S_1 where popped. The number of these operations would have been at least as large as the number of elements in S_2 at that instant, and hence the cost of copying these elements would all have been paid for already.
- qDequeue: The only costly operation is when S_2 is empty, and the reason can be one of two.
  1. This was the first qDequeue operation, in which case the cost of moving elements from S_1 to S_2 has already been paid for.
  2. Otherwise, all the elements in S_2 were popped. But, then the number of elements in S_1 would be the number of elements in S_2 that were popped, together with some new elements that were pushed. Since, we are paying 4 units for all these operations, the cost of moving half the elements to S_2 has already been paid for.

3. **(3 marks)** Alan has just graduated with a degree in Computer Science and Engineering, and being unable to find any other job has joined Willy Wonka's chocolate factory for delivering chocolates from his factory in Plymouth to Aberdeen. Alan gets to drive the battery-powered Chocolate Delivery Vehicle along Willy's Chocolate Delivery Network. The two cities are connected by a single line of the Chocolate Delivery Network.

   The Chocolate Delivery Vehicle runs on single-use batteries that must be replaced after at most 100 kilometers. There are specific Battery Replacement Stations within the direct line from Plymouth where the batteries can be replaced. Alan uses Google Maps to find the distances of the Battery Replacement Stations from Plymouth.

   As Alan is about to set out, he realizes that each Battery Replacement Station charges a different amount for replacing the battery. Furthermore, even if you have some battery left, you still have to pay the entire amount for replacing it with a new battery. Alan has the array $D[1, 2, \ldots, n]$ that gives

the distance of the i^{th} battery replacement station from Plymouth, and another array $C[1, 2, \ldots, n]$ where $C[i]$ is the cost of replacing the battery at the i^{th} station. You can assume that $D[0] = 0$ is the location of Plymouth, and $D[n + 1]$ is the distance to Aberdeen. Describe an efficient algorithm to decide the stations where Alan should purchase batteries so that the total cost of replacing batteries is minimized. You can assume that for every Battery Replacement Station, there is another one within 100 kilometers of it. You can also assume that Alan starts with a full battery at Plymouth which he doesn't have to pay for.

If you are writing a dynamic programming-based algorithm, then you must give the recurrence before writing the pseudocode, including the correct base case.

Solution: Let $\text{cost}(i)$ be the minimum cost of reaching location i from 0 with battery replacements. We are interested in calculation $\text{cost}(n + 1)$. As a base case, we know that $\text{cost}(0) = \text{cost}(1) = 0$.

Let L_i denote the set of battery replacement stations before i that are at a distance of at most 100 kilometers from i. Formally, $L_i = \{j | j < i, D[i] - D[j] \leq 100\}$.

Thus, we can write the following recurrence:

$$\text{cost}(i) = \min_{j \in L_i} \{\text{cost}(j) + C[j]\}.$$

Converting this recurrence gives an $O(n^2)$-time algorithm.

In case, you have assumed that the distances are integers, the size of the set L_i is at most 100 and this gives an $O(n)$-time algorithm. Both solutions get full credit.