CS 600 Advanced Algorithms

Homework 4 Solutions

1 C-7.4 Converting a collection of objects to a set

In order to convert a collection of objects, A into a set, we first sort A and scan through the sorted sequence and remove any duplicates we find. Using a sufficiently efficient sorting algorithm, we can achieve this in $O(n \log n)$ time – it takes $O(n \log n)$ time to sort A, and $O(n)$ time to scan and remove the duplicates. Finally, we use $\mathsf{makeSet}(e)$ for every $e \in A$, creating a singleton set containing e, and use the union operation to combine these sets. Creating the singleton sets takes $O(n)$ time. According to *Theorem 7.2*, if a list-based implementation is used, the amortized time of each union operation is $O(\log n)$. Overall, we have a runtime complexity of $O(n \log n)$.

2 A-7.2 The Offline-Min problem

We are given a sequence σ of $O(n)$ priority queue operations where each operation is either an $\mathsf{insert}(i)$ or a $\mathsf{removeMin}()$ for a distinct integer i in the range from 1 to n. To solve the sequence σ is to list out the sequence of keys returned by each $\mathsf{removeMin}()$ in σ. As an example consider,

$$\sigma = \{\underline{\mathsf{insert}(15), \mathsf{insert}(23)}, \mathsf{removeMin}(), \underline{\mathsf{insert}(7)}, \mathsf{removeMin}() \, \underline{\mathsf{insert}(42)}, \mathsf{removeMin}()\}$$

The solution to this sequence of operations is the sequence $\{15, 7, 23\}$.

We can solve this problem using a union-find data structure. We start by grouping the $\mathsf{insert}(i)$ operations together demarcating them by the $\mathsf{removeMin}()$ operations (shown in the example above). We create a set S_k for each group, I_k, of insertions using the $\mathsf{makeSet}$ operation. We then loop over each i in the range $[1, n]$ and find the set S_j such that $i \in S_j$ using the find operation. This i now is the j^{th} value returned when solving the sequence σ. We then call union on the sets S_j and S_{j+1} and keep continuing the process using we output results of all $\mathsf{removeMin}$ operations.

Overaaall, we perform a total of n $\mathsf{makeSet}$, find, and union operations. From *Theorem 7.6*, we know that this takes $O(n\alpha(n))$ time.

3 C-8.3 Linear time algorithm for computing the union of two sequences

We are given two n element sorted sequences A and B that may contain duplicates. The goal is to compute $A \cup B$ in $O(n)$ time. To achieve this, we first call $\mathsf{merge}(A, B, C)$, merging A and B to form the sequence C. We then traverse through C and remove all duplicates we encounter. From *Theorem 8.1*, we know that merging A and B takes $O(n)$ time. Further, scanning C and removing duplicates takes another $O(n)$ time. Our final asymptotic time complexity is $O(n)$.

4 C-8.7 Determine whether there are two equal elements in a sequence

We know that a total order relation is defined on S. To determine whether there are two equal elements in S, we first sort S, which takes $O(n \log n)$ time (using an algorithm like mergesort or quicksort depending on the size of n). We then scan the sequence looking for two consecutive elements that are equal, taking an addition $O(n)$ time. Overall, we achieve a time complexity of $O(n \log n)$

5 A-8.4 Matching up Nuts and Bolts

We have a set A of n nuts and a set B of n bolts. Each nut has a unique matching bolt (However they all look the same). The only comparison we can make is to take a nut-bolt pair (a, b) and test if the threads of a match with the threads of b.

This problem can be solved by using a *divide-and-conquer* approach. First we select a random bolt and partition the remaining nuts around it. That is, we select a bolt, $b_r \in B$, find a matching nut a_r and create two sets, $L_a^r = \{a_i \mid a_i \in A, a_i < a_r\}$ and $H_a^r = \{a_i \mid a_i \in A, a_i > a_r\}$. We then partition the remaining bolts around the nut we chose. That is, we create two sets, $L_b^r = \{b_i \mid b_i \in B, b_i < b_r\}$ and $H_b^r = \{b_i \mid b_i \in B, b_i > b_r\}$. At this point, we have matched the pair a_r and b_r, making a total of $n+(n-1)$ comparisons (n from first matching the bolts with the nuts, and $n-1$ from matching the nuts with the bolts). We can keep doing this procedure for the remaining nuts and bolts until they are all matched up.

Note that this is similar to the randomized quicksort algorithm. At each stage, we take $O(n)$ time. We have an expected $\lceil \log n \rceil$ levels in the recursion tree, therefore, our total runtime complexity is $O(n \log n)$.

6 C-9.5 Find equal numbers in a sequence of n integers in range $[1, n^3]$

We know that each integer in S lies in the range $[1, n^3]$. We first sort the sequence S using radix-sort, viewing the elements as triples of numbers in the range [1, n]. This takes $O(n)$ time. We then scan through the sequence and look for two consecutive elements that are equal. This takes an addition $O(n)$ time. Overall, we have a $O(n)$ time algorithm.

6.1 Viewing elements as triples of numbers in the range [1, n]

Note that an integer k is represented in base b using $\lceil \log_b k \rceil$ digits. So, given an integer k in S in the range $[1, n^3]$, we can represent k using $\lceil \log_b n^3 \rceil = 3 \log n$ digits. This means we can view an integer k in the range $[1, n^3]$ as a tuple (k_1, k_2, k_3) where k_1, k_2, and k_3 are integers with at most $\log n$ digits. Here is an example conversion,

$$S = \{2, 18, 27\} \rightarrow \{(0, 0, 2), (0, 1, 8), (0, 2, 7)\}$$

At this point, using a radix-sort, we can sort S in $O(3n) = O(n)$ time.

6.2 An alternate approach

An alternative for representing a number in S as a 3-tuple is to use the *Quotient Remainder Theorem* – given two positive integers k and n, there exists unique integers q and r such that $k = q \times n + r$ where $0 \leq r < n$.

So given k in the range $[1, n^3]$, there are two unique integers a and b such that $k = a \times n^2 + b$ where $0 \leq b < n^2$. We can apply the theorem again to b and n and get $b = c \times n + s$. Substituting b in the previous

result, we can conclude that there are three unique integers x, y, and z such that $k = x \times (n^2) + y \times n + z$ where $0 \leq x, y, z < n$. Hence, we can view k as a unique triple (x, y, z) and use radix-sort to obtain a linear time algorithm to solve this problem.

7 A-9.5 University High School Election Problem

We are given an array A containing n votes listed in no particular order. The goal is to find student numbers of every candidate that received more than $n/3$ votes in $O(n)$ time.

We can use a linear-time selection algorithm to search for items, x, and y, of rank $n/3$ and rank $2n/3$ respectively. Note that if a candidate has recieved over $n/3$ votes then his or her student number must be included in at least one of the items at rank $n/3$ and $2n/3$. So given x and y, we scan A once more and count how many times x and y appear in A.