CpSc 8400: Design and Analysis of Algorithms

Instructor: Dr. Brian Dean
Webpage: `http://www.cs.clemson.edu/~bcdean/`
Handout 2: Homework #1, Due Thursday 1/21/16

Spring 2016
TTh 12:30-1:45
McAdams 119

Some of the questions in this assignment ask you to "describe" an algorithm. You should generally do this in a clear, concise fashion in English, although you can also use pseudocode *if this adds clarity to your presentation*. Diagrams are also recommended if they help to clarify the operation of your algorithm. Any time you describe an algorithm, you should also say a few words about why it is correct (especially if this involves subtle or non-obvious observations) and also analyze its running time. Your write-ups generally do not need to be too lengthy as long as all the important details are present. Don't forget that typesetting is required, and do not forget to list your collaborators.

1-1. Virtual Initialization. One problem with arrays is that they must typically be initialized prior to use. On most computing environments, when we allocate an array of $O(n)$ words of memory they start out filled with "garbage" values (whatever data last occupied that block of memory), and we must spend $O(n)$ time setting the words in the block to some initial value. In this problem, we wish to design a data structure that behaves like an array (i.e., allowing us to retrieve the ith value and modify the ith value both in $O(1)$ time), but which allows for initialization to a specified value v in $O(1)$ time as well. That is, if we ask for the value of an element we have not modified since the last initialization, the result should be v. The data structure should occupy $O(n)$ space in memory (note that this could be twice or three times as large as the actual space we need to store the elements of the array), and the data structure should function properly regardless of whatever garbage is initially present in this memory. As a hint, try to combine the best features of an array and a linked list.

1-2. Enumerating Subsets by Incrementing a Binary Counter. This is a somewhat classical amortized analysis problem. Suppose we store the digits of an n-bit binary number B in an array A of length n. We wish to increment B from 0 to $2^n - 1$ while continually modifying the contents of A to reflect the digits in B. For example, if $n = 3$, then we will start with $A = (0, 0, 0)$, then we toggle the last entry to obtain $A = (0, 0, 1)$. The next increment operation changes A to $A = (0, 1, 0)$, and so on, until we finally reach $A = (1, 1, 1)$. Please describe how to implement the operation *increment* that takes an array A and modifies it by incrementing its associated binary number. Although your operation will require $\Theta(n)$ time in the worst case, please show that its amortized running time is only $O(1)$ (assuming the entries in A all start out at zero). Please show how to use the accounting method as well as a potential function to perform this amortized analysis.

1-3. Enumerating Permutations. Consider a length-n array $A = (1, 2, 3, \ldots, n)$. We would like to step through all $n!$ permutations of A, updating the array as we go to represent each subsequent

permutation. Permutations should be generated in lexicographic order; for example, with $n = 3$ we start with $A = (1, 2, 3)$, then move to $A = (1, 3, 2)$, $A = (2, 1, 3)$, $A = (2, 3, 1)$, $A = (3, 1, 2)$, and finally $A = (3, 2, 1)$. Please describe how to implement an operation *next-perm* with $O(1)$ amortized running time that modifies A to produce the next permutation in lexicographic order. Use a potential function in your analysis.

1-4. In-Place Matrix Transposition. Suppose we store an $m \times n$ matrix in *row-major* form — as an array of length $N = mn$ in which we list the elements of the matrix one row at a time from left to right. By taking the *transpose* of such a matrix, we convert it into *column-major* form — an array of length N listing the columns of the matrix one at a time, each one from top to bottom. Suppose the matrix is sufficiently large that we would like to permute its memory representation from row-major to column-major order *in place* (using only $O(1)$ extra storage beyond the memory that holds the matrix). Please describe and analyze an $O(N \log N)$ algorithm for solving this problem. As a hint, try to design an approach modeled on the "domination radius" algorithm from lecture, noting that for each element in the matrix, you can easily determine the location to which it needs to move, as well as the location of the element that will replace it.