ArsDigita University
Month 5: Algorithms - Professor Shai Simonson

Problem Set 1 – Sorting and Searching

When you are asked to *write, design or describe* an algorithm, you may use a computer language or pseudo-code, or a careful English description. When you are asked to write a program or code, you must actually write the program using whatever tool you like.

1. Practice with Sorting Algorithms

Use the following array (12, 10, 3, 37, 57, 2, 23, 9) for the practice problems below and show the values of the array after the indicated operations:

a. Three iterations of the outer loop in Bubble Sort.
b. Four iterations of the outer loop in Insertion Sort.
c. The first call to Partition in Quicksort.
d. One iteration of the outer loop in Radix sort.
e. All the recursive calls of Mergesort.

2. Another Slow Sort

Write a program **MAXSORT** to sort an array **A** of **size** strings. **MAXSORT** works by finding the index of the maximum element in the array from A[0] through A[size-1] and swapping it with the current last location. The current last location starts at **size – 1,** and **size** is decremented in each iteration until it equals 1.

a. Your program should read in the array of strings, sort them using **MAXSORT**, and print out the results.
b. Analyze the complexity of your algorithm by either writing a recurrence equation for the worst case time complexity of your algorithm and solving it, or by generating an appropriate sum.

3. Binary Search

a. Binary search takes $\theta(\log n)$ time, where n is the size of the array. Write an algorithm that takes $\theta(\log n)$ time to search for a value x in a sorted array of n positive integers, where you do not know the value of n in advance. You may assume that the array has 0's stored after the last positive number. Prove that your algorithm has the correct time complexity. (For you Java people, forget about the assumption that zeros follow the last legitimate array value, and assume that array.length is not available to you, but you can use catch and try to check if an index is out of bounds).
b. In insertion sort, we scan back through the sorted portion of the list to determine where the new value should be inserted. We shift all the scanned values downward, and insert the new value in the open location. Explain how to use binary search to find the appropriate insertion spot

rather than a linear scan. Explain why this does NOT help the overall time complexity of the algorithm.

4. A Better Quicksort?

Quicksort has worst-case time complexity $\theta(n^2)$ and average-case $\theta(nlogn)$. Explain how to redesign the algorithm to guarantee a worst case $\theta(nlogn)$.

5. When Constant Factors Matter

Do problem 1-2 on page 17 in your text.

6. Building Heaps

There are two ways to iteratively build a heap. We discussed these in class with an example. One way is to build a heap is to start at the end of the array (the leaves) and *push* each new value downward if necessary. Another way is to start at the root and *push* each value upward if necessary.

There are two recursive ways to build a heap. One recursively builds a heap out of the first *n-1* elements and then pushes the *n*th element upwards. The other recursively builds two heaps for each of the subtrees of the root, and then pushes the root downwards.

a. Which one of the recursive methods corresponds to which iterative method and why?
b. Construct and solve two recurrence equations for each of the recursive methods and get the Big-θ time complexity for each?
c. Write code for each of the iterative methods, and test which one is actually faster.
d. For each of the methods show what heap is built out of an array with values 1-15 in ascending order.

7. Using Heaps to Find the *K*th Largest

It is straightforward to use a heap in order to find the *k*th largest value by just deleting the deleting the top of the heap *k* times.

a. What is the time complexity of this method? Explain.

b. Challenging. Write an algorithm to do the same thing in $\theta(n + k \log k)$. Hint: Try to avoid rebuilding the original heap of size *n* each time. Instead use an extra heap that will have at most *k* elements in it, and rebuild that.

c. Write code for the two methods and test them on arrays with various values of *n* and *k*. How do your experiments compare with the theoretical results when $k = n/2$? when $k = lg\ n$?

8. An Application of Radix Sort

Given two integer arrays *A* and *B*, each of length *n*, whose values are all between 1 and *n* inclusive. Write an algorithm that sorts $C(i)=A(i)\times B(i)$ in $\theta(n)$ time. Note that straight counting sort will run here in $O(n + n^2)$. (Hint: See related problem 9.3-4 on page 180 of your text).

9. Optional: Algorithm Design Challenge

Given three arrays of *n* real numbers (the numbers can be positive or negative), write an algorithm to determine if there are three numbers, one from each array whose sum is 0. Design the most efficient algorithm you can think of to solve this problem. It can be done in $\theta(n^2)$ time. (Hint: See related problem 1.3-7 on page 16 of your text).

10. Optional: Check Your Sources

Analyze our guest speaker's theory about Heapsort and *d-ary* heaps by doing problem 7-2 on page 152 of your text.