CMPS 102: Homework #2

Due on Tuesday, April 14, 2015

John Allard 1437547

Problem 1

Design 3 algorithms based on binary min heaps that find the kth smallest # out of a set of n #'s in time:

a) $O(n \log k)$

b) $O(n + k \log n)$

c) $O(n + k \log k)$

Use the heap operations (here s is the size):

- Insert, delete: $O(\log s)$
- Buildheap: $O(s)$
- Smallest: $O(1)$

Give high level descriptions of the 3 algorithms and briefly reason correctness and running time. Part c) is the most challenging.

1. **Part A** - I found this to be a very wierd problem, any attempts to get the required run-time had me going out of my way to find a slower than optimal algorithm. I know from looking at the runtime that we need to perform n total insertion or deletion operations on a heap of size k, which would give runtime $O(n \log(k))$, but there didn't seem to be a natural way of doing so.

2. **Part B** - Designing an algorithm to run in $O(n + k \log(n))$ seemed the most inutive to me. My algorithm is like heapsort, except it stops after k iterations, which reduces its run time from $O(n \log(n))$ to $O(n + k \log(n))$.

```
// A = array of n elements of arbitray order
  findp(A)
       BuildHeap(A) // O(n)
       for i in [1..k-1] // O(k)
            Delete(A) // O(log(n))
       return Smallest(A) // O(1)
       // O(n) + O(k)*O(logn) + O(1)
```

   The algorithm starts by constructing a heap over all n elements, which is where the $O(n)$ term comes from in the runtime. Now, we simply perform $k-1$ deletion operations, which each take $O(\log(n))$ time to complete. After these operations, the kth smallest element will be at the top of the heap, so we can perform a simple Smallest retrieval operation, which will give us the kth smallest element. Runtime - $O(n)$ for Buildheap, $O(k)$ iterations of delete-min at a cost of $O(\log(n))$ gives a total run-time of $O(n + k \log(n))$.

3. **Part C** -

Problem 2

Consider the following sorting algorithm for an array of numbers (Assume the size n of the array is divisible by 3):

Problem 2 continued on next page...

- Sort the initial 2/3 of the array.
- Sort the final 2/3 and then again the initial 2/3.

Reason that this algorithm properly sorts the array. What is its running time?

Proof of Correctness

I am assuming that this is a recursive definition, and that we recurse until we reach two elements at which point we just swap them into place with a single operation. Let $P(n)$ be the statement 'for $n \geq 1$, an array A on $3n$ elements of arbitrary order will be corectly sorted by the 2/3rds sorting algorithm. '

Base Case

$P(n = 1)$ -

Problem 3

KT, problem 1, p 246.

Problem 4

Suppose you are choosing between the following 3 algorithms:

1. Algorithm A solves problems by dividing them into 5 subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
2. Algorithm B solves problems of size n by recursively solving 2 subproblems of size $n - 1$ and the combining the solutions in constant time.
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and the combining the solution in $O(n^2)$ time.

What are the running times of each of these algs. (in big-O notation), and which would you choose?

Problem 5

The *Hadamard matrices* $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0, H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Problem 6

(Extra Credit) The square of a matrix A is its product with itself, AA.

1. Show that 5 multiplications are sufficient to compute the square of a 2×2 matrix.

2. What is wrong with the following algorithm for computing the square of an $n \times n$ matrix.
   "Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part a). Using the same analysis as in Strassen's algorithm we can conclude that the algorithm runs in time $O(n^{\log_2 5})$."

3. In fact, squaring matrices is no easier that matrix multiplication. Show that if $n \times n$ matrices can be squared in time $O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.