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
1.   findp(A)
2.        BuildHeap(A) // O(n)
3.        for i in [1..k-1] // O(k)
4.             Delete(A) // O(log(n))
5.        return Smallest(A) // O(1)
6.        // 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 of length n on elements of arbitrary order will be corectly sorted by the 2/3rds sorting algorithm.'

Base Case

$P(n = 1)$ - If $n = 1$, there is only one element to be sorted so we just return the array. Confirmed.
$P(n = 2)$ - If $n = 2$. no recursion is needed, we simply perform a single comparison and swap the items into place, then return the array. Confirmed.

Inductive Step

For $n \geq 3$ where n is a multiple of 3, assume that $P(k)$ is true for $3 \leq k \leq n$, i.e. that an array of length k can be sorted correctly by the given sorting algorithm.
Start with an array of size $n + 3$ (the next multiple of 3). Let A_1, A_2, and A_3 represent the 1st, 2nd, and final thirds of the array indices. On the first call we attempt to recurse on $A_1 \cup A_2$. Because this sub-array is of size $\frac{2}{3}(n + 3)$, which is less than or equal to n for $n \geq 3$, we can apply the inductive hypothesis on this subarray.
After applying the inductive hypothesis, $A_1 \cup A_2$ will be sorted properly, and thus all of the elements in A_2 will be at least as great as the elements in A_1.

$$\forall x \in A_1, y \in A_2, x \leq y \qquad (1)$$

We then recurse on $A_2 \cup A_3$, once again the inductive hypothesis applies by the same argument given above, which means $A_2 \cup A_3$ is sorted. This implies that all elements in A_3 are at least as great as those in A_2.

$$\forall x \in A_2, y \in A_3, x \leq y \qquad (2)$$

If you combine this fact with the result of the last recursive call (1), we deduce the following :

$$\forall x \in A_1 \cup A_2, y \in A_3, x \leq y \qquad (3)$$

This means that all elements in A_3 are in the right place. The final 3rd of the array being in the right place implies that all of the elements in the indices $A_1 \cup A_2$ belong in that first 2/3rds of the array, but they are possbily scattered and out of orderby our last sorting call.
With one final recursive call on $A_1 \cup A_2$, our inductive hypothesis once again applies and (1) is restored. (3) is still valid because the last sort only rearranged items in A_1 and A_2, which were all less than or equal to items in A_3 to begin with. Combining (1) and (3) :

$$\forall x \in A_1, y \in A_2, z \in A_3, x \leq y \leq z \qquad (4)$$

Add to this the fact that the individual thirds are correctly sorted internally (by our ind. hyp.), and we have shown that the entire array of $n + 3$ elements has been properly sorted. ///

Runtime

This algorithm is slightly odd be it does almost no work while dividing nor when recombining. When we get down to a length less than 3, we simply compare pairs of elements are swap them if necessary. Only at the bottom level do we perform any comparisons, on every other level we simply recurse 3 times on an input of

size 2/3rds the current sub-array size. Thus the recurrence is :

$$T(n) \left\{ \begin{array}{cc} = O(1) & \text{if } 1 \leq n \leq 2 \\ \leq 3\text{T}(2\text{n}/3) + \text{O}(1) & \text{if } n \geq 3 \end{array} \right\}$$

We can apply the master theorum with $a = 3$, $b = 3/2$, and $f(n) = O(1)$. $\log_b(a) = log_{\frac{3}{2}}(3) = 2.7095$. The exponent on f is 0, so $f = O(n^{\log_b(a)-\epsilon})$, thus we are in case A of the master theorem. This means that :

$$T(n) = \Theta(n^{\log_{1.5}(3)} = n^{2.7095})$$

///

Problem 3

KT, problem 1, p 246. : You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values, so there are 2n values total and you may assume that no two values are the same. Youd like to determine the median of this set of 2n values, which we will define here to be the nth smallest value.
However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most O(log n) queries
Since I am required to write an algorithm that runs in $O(\log n)$ quries, I know that I have to divide my search space by a constant multiple every constant number of queries, like how bianry search narrows the search space by a factor of a half every iteration. Like binary search, I'll start by looking in the middle of the data sets, which is the $\frac{n}{2}$ smallest number in each database of size n. The key idea that I'll use is that the median of the data set has to have a value that is between the medians of the individual data-sets. This is because if get the $\frac{n}{2}$ smallest number from both data-sets, we know there are at minimum going to be n-numbers smaller than the greater of the two medians. I'll go into this in more detail in the proof of correctness.

```
// db_n = data base #n
// db_query(k, db_n) returns the kth smallest item in db_n
1.   FindMedian(db_1, db_2, n)
2.        ind_1 = n/2; // get the median value #1
2.        ind_2 = n/2; // get the median value #2
3.        for k in [1..log(n)] // log(n) iterations to find median
4.
5.        return Smallest(A) // O(1)
6.        // O(n) + O(k)*O(logn) + O(1)
```

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.

Problem 4 continued on next page...

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 $O(n^c)$.