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. Edit - so after seeing the hint by the TA on piazza, I think I know how to build the algorithm. We start by building a max-heap of the first k elements in the array. We then scan through the array, and everytime we find a smaller element we delete the max element from the heap and insert the new smaller element. Since we will continually add the smallest elements to a max-heap of constant size k, at the end the kth smallest (the largest of the k smallest elements) will be at the top of the heap and we can simply grab that value with a single function call.

```
// A = array of n elements of arbitray order
1.   findp(A)
2.        H = BuildMaxHeap(A[1..k]) // build max-heap of first k elements
3.        for i in [k..n] // O(n)
4.             if A[i] < Largest(H) // if new val belongs in heap
5.                  Delete(H) // remove top value
5.                  Insert(H, A[i]) // push new val into heap
6.        return Largest(H)
```

The algorithm starts by constructing a heap on k elements. This is key, because this lets us achieve the log(k) insetion and deletion time that is needed. We then iterate over the rest of the elements (on the order N, could be much less if k is close to n), performing a single deletion and insertion operation if we find an element that is smaller than the max item in the max heap. Doing this ensures that the smallest values are inserted into the heap, and since we start with k values and only perform single deletions and insertions together, the size of the heap will always remain constant at k. Thus at the end, we will have a max-heap containing the k smallest elements in A. The max of these elements will be at the top of the heap, and it will be the kth smallest element of A (the largest of the k smallest elements), so we return it. This algorithm runs for O(n) iterations, performing an insert and deletion operation each iteration for the worse case. Thus the run-time is :

$$(n-k) * 2 * \lg(k) = 2 * n * \log(k) - 2 * k = O(n \log(k))$$

Problem 1 continued on next page...

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.        H = BuildHeap(A) // O(n)
3.        for i in [1..k-1] // O(k)
4.             Delete(H) // O(log(n))
5.        return Smallest(H) // 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):

- 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.

Problem 2 continued on next page...

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

Problem 3 continued on next page...

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 binary 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
// pow(x, n) raises x to the nth power
1.   FindMedian(db_1, db_2, n)
2.        ind_1 = n/2; // get the median value #1
3.        ind_2 = n/2; // get the median value #2
4.        for k in [2..log(n)] :  // log(n) iterations to find median
4.             med_1 = db_query(ind_1, db_1) // get ind.  median value
4.             med_2 = db_query(ind_2, db_2) // get ind.  median value
5.             if med_2 < med_1 :
6.                  ind_1 = ind_1 - n/pow(2,k) // recurse in the lower half
7.                  ind_2 = ind_2 + n/pow(2,k) // recurse in the upper half
8.             else :  // med_1 > med_2 (no dups)
9.                  ind_1 = ind_1 + n/pow(2,k)
10.                  ind_2 = ind_2 - n/pow(2,k)
11.        // at this point the median is the smaller of the two ind_n vals
12.        if med_1 < med_2 :
13.             return med_1
14.        else :
15.             return med_2
16.        // 2+ [lg(n)-1] = lg(n)+1 = O(log(n)) db queries
```

Continuing from above, I start by getting the two medians of the respective data-bases, `ind_1` and `ind_2`. Because these are the n/2 smallest number in their respective sets, there are at least n numbers smaller than the max of the two medians. We also know that the median has to be greater than or equal to the minimum of the two individual medians, otherwise it would no longer be the nth smallest number in the union of the two data-bases.

Since we can pin the median to being between the two individual medians, we can then iterate through half the search space by looking only in the halves of the databases that could possibly contain the global median. If `ind_1` $<$ `ind_2`, then we know that the median must either be in the lower half of db_1 or in the upper half of db_2, so we narrow the search space to those halves by adjusting the index. If `ind_1` $>$ `ind_2`, we do the opposite and narrow the search space to the upper half of db_1 and the lower half of db_2. Since we are narrowing the search space of each data-base of length n by half each time, we only need to perform lg(n) iterations to reduce the halves to single elements. Once we have done this, `ind_1` and `ind_2` will the nth and n+1st smallest items (in some order) in the total data set. This means we can simply return the smaller of the two values, which will be nth smallest value in the combined data-set, which is exactly what the algiorithm is supposed to do.

Runtime Analysis - Given two datasets of unique elements in which we can only query individual sets for the kth smallest element, the algorithm given above will run in time O(log(n)). To start, notice that we always iterate lg(n)-1 times, this is because we are cutting the space in half every time and we start the

for-loop already at the middle of the search spaces. Each iteration of the loop we perform 2 query operations, so f(n) = 2. Thus the recurrance for this algorithm is :

$$T(n) = T(n/2) + 2$$

Which, can be solved using the master theorum :

$$\log_b(a) = \log_2(1) = 0, f(n) = 2 * n^0. \text{ Case \#2}$$

$$T(n) = O(log(n))$$

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?

1. The recurrance (extracted from the desceription) is as follows :

   $$T(n) = 5T(n/2) + O(n)$$

   Since $\log_2(5) > 1$, then $f(n) = O(n^{\log_2(5)-\epsilon}$, we are in case one of the master theorum. Thus :

   $$T(n) = O(n^{log_2(5))=O(n^{2.322}})$$

2. The master theorem doesn't directly apply to this one, so I used iteration and substitution like is used in the derivation of the master theorem. The recurrence is :

   $$T(n) = 2T(n-1) + c$$

   Substituting $T(n-1)$ and multiplying out:

   $$T(n) = 4T(n-2) + 3c$$

   Substituting in $T(n-2)$ and multiplying :

   $$T(n) = 8T(n-3) + 7c$$

   Generalizing from this pattern :

   $$T(n) = 2^k T(n-k) + (k-1)c$$

Problem 4 continued on next page...

If we let k go to n, which bottoms out the recurrance, we get :

$$T(n) = 2^n T(0) + (n-1)c$$

$$T(n) = 2^n + O(n)$$

$$T(n) = O(2^n)$$

Thus the run-time is of exponential order for this algortihm. This makes sense to me, as we double the amount of problems we have to solve for every call while only reducing the amount of work each subsequent call has to by 1.

3. The recurrance (extracted from the desceription) is as follows :

$$T(n) = 9T(n/3) + O(n^2)$$

Since $\log_3(9) = 2$, then $f(n) = \Theta(n^{log_3(9))} = n^2$, we are in case two of the master theorum. Thus :

$$T(n) = O(n^2 \log(n))$$

Between the first and the third options, the third option just barely wins out. This is because $n^x log(n) = O(n^{x+\epsilon})$ for all $x \geq 1$. Letting $x = 2, \epsilon = \log_2(5) - 2$ we can see that the third option is Big-O of the first, which makes the third option the better choice. Option #2 is obviously the worse choice with super-polynomial run-time, so both option #3 and option #1 are better choices than it. This leaves option #3 as the best choice for fastest algorithm.

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 $O(n^c)$.