HW2 - S16

Chris Troiano - ctroiano 4-12

1. Design 3 algorithms based on binary min-heaps (and/or max-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.

   Solution:

a)
```
negate every element in array A[]
buildheap (k)
x = k+1
while x <= n
if (smallest () < A[x])
delete (smallest ())
        insert (A[x])
    x++
negate every element in array A[] to get back to positive values
return smallest ()
```

We build the heap in $O(k)$ time and then iterate through the array $(n - (k + 1))$ times to compare the heap and $(n - (k + 1))$ elements of the array. During comparing we use $2 \log k$ functions, giving us in total an $O(n \log k)$.

b)
```
buildheap (n)
x = 0
while x <= k-1
delete (smallest ())
    x++
return smallest ()
```

We build the heap in $O(n)$ time and then iterate through the array k times to find the kth smallest element. In the loop, delete is called for a time of $O(\log n)$. This gives us an algorithm with $O(n + k \log n)$.

c)
```
buildheap (n)
insert (smallest ()) into a new heap
```

```
for every element <= kth element
if element == k
     return element
    else insert the children of the extracted element into new array
```

We build the heap in $O(n)$ time. We then insert the smallest element into a new array which only takes $O(1)$. We then iterate through the original heap k times, if the k element isn't found, we insert the children of inspected element into the new array, which takes $O(k \log k)$

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?

   Solution: Induction proof based on size of array l
   base: when $l \leq 3$, the algorithm trivially sorts the array
   Inductive Hypotesis: let l >3 and assume 2/3 sort, sorts all arrays of size <l. The algorithm makes 3 recursive calls.

   1. sort inital 2/3
   2. sort last 2/3
   3. sort inital 2/3

   For convenience, call the 1st, 2nd, and 3rd parts of the array [A, B, C]

   1. After 1st recursive call, A & B are sorted; B's elements are greater than or equal to A's, by inductive hypothesis.
   2. After 2nd recursive call, B & C are sorted; C's elements are the largest in the array, and we are done sorting C
   3. The last pass guarantees A & B's elements are sorted. Therefore, the array is sorted in increasing order.

   By Master Theorem:
   $T(n) = 3T(\frac{n}{2/3}) + O(1)$
   $n^{\log_{\frac{3}{2}} 3} \approx n^{2.7} > 1 \Rightarrow \Theta(n^{\log_{\frac{3}{2}} 3})$

3. KT, problem 1, p 246.

   Solution: The median can be solved recursively with databases A & B.
   First find median of both A & B.
   $A^* = \frac{n}{2} smallest$ $\qquad$ $B^* = \frac{n}{2} smallest$

   - $A^* > B^*$, the elements in $A[\frac{n}{2}...n] > B^*$, so we can throw them away. The median cannot lie in $B[1...\frac{n}{2}]$ either, so we can throw that away too. We can now recursively solve a subproblem with $A[1...\frac{n}{2}] \& B[\frac{n}{2}...n]$.

- $A^* < B^*$, we can throw away $B[\frac{n}{2}...n]\&A[1...\frac{n}{2}]$. We can now recursively solve a subproblem with $A[\frac{n}{2}...n]\&B[1...\frac{n}{2}$.

In both cases, the subproblem reduces by a factor of $\frac{1}{2}$ and we spend constant time comparing the two. This gives us the recurrence relation $T(n) = T(\frac{n}{2}) + O(1)$.
By Master Theorem:
$n^{log_2 1} = n^0 = O(1)$
Therefore $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$

4. Suppose you are choosing between the following 3 algorithms:

   (a) 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.

   (b) Algorithm B solves problems of size n by recursively solving 2 subproblems of size $n-1$ and the combining the solutions in constant time.

   (c) 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?

   Solution:

   (a) $T(n) = 5T(\frac{n}{2}) + O(1)$
   By Master Theorem:
   $n^{\log_2 5} > n$, so $T(n) = \Theta(n^{\log_2 5})$

   (b) $T(n) = 2T(n-1) + O(1)$
   By Substitution:
   $n = 1 : T(1) = 1$
   $n = 2 : T(2) = 1 + (2+1) = 4$
   $n = 3 : T(3) = 1 + 3 + (4+1) = 9$
   $n = 4 : T(4) = 1 + 3 + 5 + (6+1) = 16$
   We can tell that this runs in $O(2^n)$.

   (c) $T(n) = 9T(\frac{n}{3}) + O(n^2)$
   By Master Theorem:
   $n^{\log_3 9} = n^2$, so $T(n) = \Theta(n^{\log_3 9} logn) = \Theta(n^2 logn)$

5. (a) Compute the FFT of the polynomial $1 + 2x - x^3$ by computing the 4 dimensional FFT matrix and multiplying it with the coefficient vector $[1\ 2\ 0\ \ -1]^\top$.

   The FFT matrix uses powers of a root of unity. First determine the appropriate root of unity.

   (b) Now compute the inverse FFT of the vector $[1\ 2\ 0\ \ -1]^\top$. Again find the appropriate matrix and multiply this matrix by the vector.

   (c) Check that the two matrices used above are inverses of each other.

   Solution:

(a) n^{th} root $= \omega = e^{\frac{2\pi i}{4}} = e^{\frac{\pi i}{2}} = \cos(\frac{\pi i}{2}) + i\sin(\frac{\pi i}{2}) = i$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1+3i \\ 0 \\ -2i \end{bmatrix}$$

(b)

$$\frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}^{-1} \Rightarrow \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4}i & -\frac{1}{4} & \frac{1}{4}i \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4}i & -\frac{1}{4} & -\frac{1}{4}i \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{4} - \frac{3}{4}i \\ 0 \\ \frac{1}{4} + \frac{3}{4}i \end{bmatrix}$$

(c)

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \times \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4}i & -\frac{1}{4} & \frac{1}{4}i \\ \frac{1}{4} & -\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4}i & -\frac{1}{4} & -\frac{1}{4}i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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

(b) 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 $O(n^{\log_2 5})$."

(c) 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 $O(n^c)$.

Solution:

(a)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a^2 + bc & ab + bd \\ ca + cd & cb + d^2 \end{bmatrix}$$

1. a^2
2. d^2
3. $b(a+d)$
4. $c(a+d)$
5. cb

(b) We cannot use the solution for a). The reason that we were able to use 5 multiplications was becuase we were squaring two matrices. This will not work for Strassen's algorithm because we are not guaranteed that the subproblem will multiply 2 identical matrices.

(c) When multiplying two $n \times n$ matricies, the resulting output contains n^2 elements. When evalutaing the n^2 elements, we will need n operations. This results in time complexity $n * n^2 = n^3 \Rightarrow O(n^c)$