David Sun
CMPS 102
Homework 2

<u>Problem 1:</u> 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) $O(n + k \log k)$

Note: For all problems, an array will be used to represent the heap.
a) For part a, we initially allocate an array of size k for our heap with all the elements initialized to $-\infty$ *(according to CLRS, a heap containing all duplicate keys can still be classified as a valid min-heap).* Then, we will make a pass through our array A from $A[1 \ldots length[A]]$. For each A_i where $i \in \{1 \ldots length[A]\}$, check if $-A_i$ is greater than the minimum of the heap. If such is the case, call remove on the heap, and insert $-A_i$ into the heap. We apply this operation for $length[A]$ iterations. Once we've scanned through all the elements in A, the kth smallest element should be $-1 * smallest(heap)$.
Correctness for a: The constraint in part a is that only the smallest element can be removed. In this case, we construct a heap using the negatives of the array elements. Thus if the *negative* of an array element is *greater than* the minimum heap value, then replace the minimum heap value with the negative of the array element. Since $\forall a, b \in \mathbb{R}$, if $a > b$ then $-b > -a$, replacing the minimum value in the heap containing negatives is equivalent of replacing the maximums in a max-heap. By the end of one pass through the array, the min-heap would contain the negatives of the k-smallest values with the k-th smallest of the array as the smallest value of the heap. To obtain the minimum, multiply the smallest value of the heap by -1 and that value would be the original kth smallest element of the array.
Runtime analysis for a: One pass through the array is an O(n) operation. Each array element comparison with the smallest heap element is an $O(1)$ operation. In the worst case, we would need to replace the minimum heap at element at every compare, which would cost us $2 \log k$ for a remove and insert operation. Thus, in the worst case, the algorithm generally runs in $n \cdot 2 \log k$ time, which suggests the algorithm is $O(n \log k)$

b) Call buidheap on the array A. This ensures A is now an array that satisfies the heap property. Since we want the kth smallest element, call remove $k-1$ times. After $k-1$ removes, the kth smallest element would be the smallest element on the heap.
Correctness for b: If A is a proper min-heap containing the original array elements, then a call to remove ensures the next smallest element is at the top of the heap. For j removes on the heap, the $j + 1^{th}$ smallest element is at the top of the heap. Thus, calling remove $k-1$ times extracts the $(k-1) + 1^{th}$ smallest element, which is the kth smallest.
Runtime analysis for b: Buildheap on an array of size n is an $O(n)$ operation. After buildheap executes, calling remove on a heap of size n $k-1$ times costs $(k-1) \log n$ time. In general, the algorithm runs in $n + (k-1) \log n$ time, which means that the algorithm is $O(n + k \log n)$.

c) Call buildheap on the array, ensuring the array is now a valid min-heap, call this heap A. Allocate another array of size k for our second heap. Let H be a heap containing a key-value pair, with its keys being the elements of A and the values being the index at which the element is found. Initially, insert the smallest of A along with the index of the smallest into H. At each iteration, remove the smallest value-index pair $(A[i], i)$ from H and insert $(A[2i], 2i)$ and $(A[2i+1], 2i+1)$ into H *(Note: if $length[A] < 2i$, do not insert either pair into H. If $length[A] < 2i+1$, insert only $(A[2i], 2i)$ into H).* After $k-1$ iterations, the kth smallest element is the smallest key in H.
Correctness for c: By replacing the smallest in our new heap and inserting the children of the smallest from the original heap, all possible candidates for the next minimum are considered. Once the children are inserted, a new minimum is at the root, whether it was an element already in the heap, or a newly inserted element. Since the heap always reads the smallest in $O(1)$ time, repeating this process $k-1$ times and reading the smallest after $k-1$ iterations gives us the kth smallest value.
Runtime analysis for c: If the original array has size n, buildheap on the original array is an $O(n)$ operation. After buildheap, we execute $k-1$ operations of insert and remove on H, which has size k. In the worst case, one remove and two inserts will be called, which has a runtime of $3\log k$. In general, the algorithm takes $n + (k-1) \cdot 3\log k$ time to execute, which means the algorithm is $O(n + k\log k)$.

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.

Correctness: Use induction on n, the size of the array, which we'll call A.
Base Case: $n = 3$. Sorting the initial 2/3 ensures $A[1] \leq A[2]$. Sorting the final 2/3 ensures $A[2] \leq A[3]$, thus ensuring that $A[1] \leq A[3]$ and $A[2] \leq A[3]$. Sorting the initial 2/3 ensures again $A[1] \leq A[2]$, ensuring $A[1] \leq A[2] \leq A[3]$.
Inductive Step: Assume for all k, $3 \leq k < n$ that the algorithm properly sorts an array of size k. Then, for an array A of size n, we can partition up A into three segments B, C, and D, with B containing $A[1 \ldots \frac{n}{3}]$, C containing $A[\frac{n}{3}+1 \ldots \frac{2n}{3}]$, and D containing $A[\frac{2n}{3}+1 \ldots n]$.
Sorting the initial 2/3 of the array sorts B and C. By the inductive hypothesis, B and C are properly sorted and BC is a properly sorted segment with every element in B less than or equal to every element in C.
Sorting the final 2/3 of the array sorts C and D. By the inductive hypothesis, C and D are properly sorted and CD is a properly sorted segment with every element in C less than or equal to every element in D. This ensures that every element in D is greater than or equal to the elements in both B and C.
Sorting the initial 2/3 of the array sorts B and C again. In the process of sorting CD, C contained the minimal values of D, which could be less than the elements found in B. Thus sorting the the initial 2/3 again ensures every element in B is less than or equal every element in C, by the inductive hypothesis.
Since B, C, D are properly sorted segments and $\forall b \in B$, $\forall c \in C$, $\forall d \in D$, $b \leq c \leq d$, the algorithm properly sorts the array.

Runtime analysis: The algorithm divides the problem into three sub-problems each of size $\frac{2n}{3}$. At each level, there is no other work done so we'll add an $O(1)$ overhead. The recurrence can be written as $T(n) = 3T(\frac{2n}{3}) + O(1)$ or $T(n) = 3T(\frac{n}{1.5}) + O(1)$. We can now apply the master theorem. $a = 3$, $b = 1.5$, $f(n) = n^0$, $k = \log_{1.5} 3$. $f(n) = O(n^{k-\epsilon})$ so $T(n) = \Theta(n^{log_{1.5}3})$.

Problem 3: KT, problem 1, p 246.
Algorithm: Call our two databases D1 and D2. Our divide and conquer algorithm will be called **findMedian**(D1, D2, lo1, hi1, lo2, hi2), where $lo1$ and $hi1$ are lower and upper bounds for D1 and $lo2$ and $hi2$ are lower and upper bounds for D2. The algorithm will initially be called in the manner **findMedian**(D1, D2, 1, N, 1, N), if N is the number of elements within each of the databases. The base case of the algorithm will checks if $lo1 == hi1$ and if $lo2 == hi2$. This means we have two sub-lists containing 1 element each, so return $min(D1[lo1], D2[lo2])$ to find the median.
If the above comparisons do not hold, let $mid1 = \lfloor \frac{lo1+hi1}{2} \rfloor$ and let $mid2 = \lfloor \frac{lo2+hi2}{2} \rfloor$. Compare $D1[mid1]$ and $D2[mid2]$. If $D1[mid1] > D2[mid2]$, the median of the two lists lies within the lower half of D1 and in the upper half of D2, so call **findMedian**(D1, D2, lo1, mid1, mid2, hi2). If instead $D1[mid1] < D2[mid2]$ then the median lies of the two lists lies within the upper half of D1, and in the lower half of D2, then call **findMedian**(D1, D2, mid1, hi1, lo1, mid2).

Proof of correctness: If we have two lists containing one element each, then the median would lie at the 1st position, which would be the minimum of the two lists. Otherwise, for databases each with more than 1 element, taking the median of two databases means that there are at least half the elements less than the higher median and half the elements greater than the lower median, suggesting that the median must lie between the lower and higher median. Therefore, it is necessary to discard half of each database until we reach one element for both databases.

Runtime analysis: At each level, half the input size n is discarded. If each database is n/2 in size, then $((n/2)/2) + ((n/2)/2) = n/2$. At each level, there are also number comparisons, done in $O(1)$ time. Thus, the recurrence can be written as $T(n) = T(n/2) + O(1)$. Using the master theorem, we can show that $T(n) = O(\log n)$.

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?

Runtime analysis:

• The recurrence for (A) can be written as $T_a(n) = 5 \cdot T_a(n/2) + O(n)$
$k = \log_2 5$, $f(n) = n^1$, so $f(n) = O(n^{\log_2 5-\epsilon})$ so $T_a(n) = \Theta(n^{\log_2 5})$, thus $T_a(n) = O(n^{\log_2 5})$

• The recurrence for (B) can be written as $T_b(n) = 2 \cdot T_b(n-1) + O(1)$
In this case, the master theorem does not apply here. However, analyzing this recurrence, we can see that $T_b(n) = 2T_b(n-1) + k = 2 \cdot (2 \cdot T_b(n-2) + k) + k = 2 \cdot (2 \cdot (2 \cdot T_b(n-3) + k) + k) + k = 2 \cdot (2 \ldots (T_b(n-n) + k) \cdots + k) + k$. Notice that by expanding the recurrence, there are $n-1$ products of $2s$. Since the k values are constants, 2^{n-1} is the dominating term. Thus, we can conclude that the algorithm runs in $O(2^n)$ time.

• The recurrence for (C) can be written as $T_c(n) = 9T(n/3) + n^2$. In this case $k = 2$, $p = 0$ so $f(n) = \Theta(n^2)$, so $T_c(n) = \Theta(n^2 \log^{p+1} n)$, thus $T_c(n) = \Theta(n^2 \log n)$ so $T_c(n) = O(n^2 \log n)$.

Since algorithm B is an exponential, it grows faster than A and C, and we can conclude it is not the most optimal algorithm. We compare $O(n^{\log_2 5})$ with $O(n^2 \log n)$. Note that $\log_2 5 \approx 2.31$. So, $O(n^{\log_2 5}) = O(n^2 n^{0.31})$. We compare $n^{0.31}$ to $\log n$.

$$\lim_{n\to\infty} \frac{\log n}{n^{0.31}} = 0$$

Therefore, algorithm C is the most optimal algorithm.

Problem 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.

The polynomial has four roots of unity, which are $e^{\frac{0\cdot 2\pi i}{4}}$, $e^{\frac{1\cdot 2\pi i}{4}}$, $e^{\frac{2\cdot 2\pi i}{4}}$, and $e^{\frac{3\cdot 2\pi i}{4}}$. Let $\omega = e^{\frac{1\cdot 2\pi i}{4}} = e^{\frac{\pi i}{2}}$.
The FFT matrix can be expressed as

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

(a) We multiply this matrix by the coefficient vector by the coefficient vector to get the FFT.

$$\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & -1 & -i\\ 1 & -1 & 1 & -1\\ 1 & -i & -1 & i\end{bmatrix}\begin{bmatrix}1\\ 2\\ 0\\ -1\end{bmatrix} = \begin{bmatrix}2\\ 1+3i\\ 0\\ 1-3i\end{bmatrix}$$

(b) The inverse Fourier matrix can be expressed as

$$f_4^{-1} = \frac{1}{4}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3}\\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6}\\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9}\end{bmatrix} = \frac{1}{4}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & i\\ 1 & -1 & 1 & -1\\ 1 & i & -1 & -i\end{bmatrix}$$

We multiply this matrix with the coefficient vector.

$$\frac{1}{4}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -i & -1 & i\\ 1 & -1 & 1 & -1\\ 1 & i & -1 & -i\end{bmatrix}\begin{bmatrix}1\\ 2\\ 0\\ -1\end{bmatrix}=\begin{bmatrix}\frac{1}{2}\\ \\ \frac{1-3i}{4}\\ \\ 0\\ \\ \frac{1+3i}{4}\end{bmatrix}$$

Thus obtaining the inverse FFT of the vector.

(c) To check if the two 4x4 matrices are inverses of each other, we multiply them together and check if the product is the identity matrix. In this case

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

Thus, we can conclude that the matrices are indeed inverses.

Problem 6: Extra Credit

(a) Let A = $\begin{bmatrix}a & b\\ c & d\end{bmatrix}$ therefore AA = $\begin{bmatrix}a^2+bc & ab+bd\\ ac+cd & cb+d^2\end{bmatrix}$

Thus the five Multiplications can be expressed as

$M_1 = a^2$

$M_2 = bc$

$M_3 = a(b+d)$

$M_4 = c(a+d)$

$M_5 = d^2$

So AA can be expressed as $\begin{bmatrix}M_1+M_2 & M_3\\ M_4 & M_2+M_5\end{bmatrix}$

(b) **Proof by counter-example:** Let A be a $4x4$ matrix containing four $2x2$ sub-matrices. Thus A can be expressed as

A = $\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}$, so $A^2 = \begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix}\begin{bmatrix}A_{11} & A_{12}\\ A_{21} & A_{22}\end{bmatrix} = \begin{bmatrix}A_{11}^2+A_{12}A_{21} & A_{11}A_{12}+A_{12}A_{22}\\ A_{21}A_{11}+A_{22}A_{21} & A_{21}A_{12}+A_{22}^2\end{bmatrix}$

Note that only squares of matrices can be calculated using 5 multiplications. It takes seven multiplications to calculate the product of two different matrices. So if any of the sub-matrices are not equal (e.g. $A_{12} \neq A_{21}$), then it would take more than 5 multiplications to calculate the product, thus the proposition does not hold.