Midterm Exam
(solutions)

1. Amortized Complexity of Union – Find

(i) The arbitrary Union strategy makes it possible to build any tree, for instance, the binomial tree \( B_k \), where \( k = \log(n/2) \), with the first \( n/2 \) Union operations. This will take \( c \cdot n/2 \) time. Call this \( B_k \) the big tree. The next \( n \) operations will come in pairs:

Union the big tree with a singleton set, arbitrarily linking the root of the big tree to the root of the trivial (singleton) tree as its parent (that’s what adversaries do.) Then,

Find the deepest node of the big tree (the only node at depth \( k + 1 \)), compressing the find-path to the root.

Since each node on the find-path has a binomial tree \( B_i \) as a principal subtree (by definition of binomial tree), linking them to the root during path compression creates again a binomial tree \( B_k \) (plus an extra node). Each of this pair of operations takes \( c \cdot (1 + k) \) time. Thus, the sequence of these \( 3n/2 \) operations takes \( c \cdot (n/2 + nk/2) \) time, or \( \Theta(\log n) \) amortized per operation.

(ii) Each non-trivial parent pointer is created by a Union operation. Each pointer traversal during a Find operation can be apportioned to (a) the Union operation that created it, or (b) the Find operation itself (if it points to the root). In terms of a credit invariant for the Unions phase: “each pointer carries a credit”. It can be maintained by paying out two credits per operation: a Union creates a pointer and leaves an extra credit. For the Finds phase, the invariant becomes “each pointer not to the root carries a credit”: pointer credits pay for their traversal and linking to the root during path compression and Find pays for the last pointer traversal.

2. Greedy Loop Invariant

The greedy algorithm combines two smallest frequency (pseudo)messages into one, with the frequency equal to their sum. Since those messages are deleted, the remaining (pseudo) messages have frequencies that are not smaller. This is also an invariant “R: remaining frequencies are not smaller than those deleted” of the execution. Denoting by \( f_1, f_2 \) the two smallest frequencies and by \( f'_1, f'_2 \) the two smallest after the deletion, we have an invariant: \{R and \( f_1 + f_2 = c \}\} combine, delete, and insert \{R and \( f'_1 + f'_2 \leq c \}\}.
A list of (pseudo)messages linked in non-decreasing order of frequencies allows for constant time access to and deletion of the two smallest frequencies as well as insertion in-order of the new frequency. The latter is possible by keeping the pointer to the last inserted item (initially at the beginning of the list); this pointer will only have to advance with the non-increasing order, never to back up. The time complexity is the constant work per each two consecutive list items, plus the total number of pointers to follow, both equal to the original \( n \) plus the \( n - 1 \) newly inserted.

3. Divide and Conquer

Correctness follows by induction (on the number of recursive calls of order statistics procedure) from the partitioning of the original elements about the median \( M \): the \( k \)th smallest of the original collection is the \( k \)th smallest of the \( m \) smallest element if \( k \leq m \) and the \( k - m \)th smallest among the rest, otherwise. Since \( M \) is not smaller than \( n/6 \) middle elements \( m_i \), each not smaller than the smallest element in its triple, \( M \) is not smaller than (and not larger than) \( 4n/6 \) of the original elements.

Complexity \( T(n) \) of the original problem follows from adding the time of the first recursive call, \( T(n/3) \), to find \( M \), to the time of the second recursive call, \( T(4n/6) \), and the time to sort the \( n/3 \) triples and to partition the \( n \) elements (both linear tasks.) Each node of the tree of recursive calls represents the time proportional to the size of the problem: \( n \) for the root, \( n/3 + 4n/6 = n \) for its children, \( n/3(1/3 + 4/6) + 4n/6(1/3 + 4/6) = n \) for their children, and so on. There being \( \log n \) levels in the tree, the total is \( T(n) = \Theta(n \log n) \).

4. Fancy Fourier

Not really FFT but something closely related.

(i) Let us define an auxiliary polynomial \( B'(x) = \sum_{j=1}^{n} j^{-2}x^j \), so that \( B(x) = B'(x) - B'(x^{-1}) \). The product of \( A(x) \) and \( B(x) \) is then equal to

\[
C(x) = A(x)B'(x) - A(x)B'(x^{-1}) = \sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} a_i b_j x^{i+j} - \sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} a_i b_j x^{i-j},
\]

where \( a_i = q_i \) and \( b_i = i^{-2} \) are the coefficients of \( x^i \) in the polynomials \( A(x) \) and \( B'(x) \), respectively.

Let us change the order of summation in the first sum by substituting \( k = i + j \). Considering that indices \( i \) and \( j \) obey the conditions \( 1 \leq i \leq n \) and \( 1 \leq j = k - i \leq n \), we introduce new indices \( k \) and \( i \) with range \( 2 \leq k \leq 2n \) and \( \max\{1, k-n\} \leq i \leq \min\{k-1, n\} \), respectively. Similarly, we perform the substitution \( k = i - j \) in the second sum, and switch from indices \( i \) and \( j \) to indices \( k \) and \( i \) such that \(-n + 1 \leq k \leq n - 1 \), and
\[
\max\{1, k+1\} \leq i \leq \min\{k+n, n\}. \text{ In such a manner, we obtain}
\]
\[
C(x) = \sum_{2 \leq k \leq 2n} a_i b_{k-i} x^k - \sum_{-n+1 \leq k \leq n-1} a_i b_{-i-k} x^k. 
\]

The polynomial \( C(x) \) has nonzero coefficients \( c_k \) for all powers \( x^k \) \((-n+1 \leq k \leq 2n)\). If we consider \( c_k \) for values of \( k \) limited to the range from 2 to \( n+1 \), the above formula can be rewritten as:

\[
c_k = \sum_{1 \leq i \leq k-1} a_i b_{k-i} - \sum_{k+1 \leq i \leq n} a_i b_{i-k} = \sum_{1 \leq i < k} \frac{q_i}{(k-i)^2} - \sum_{k < i \leq n} \frac{q_i}{(k-i)^2} 
\]

Clearly, these coefficients correspond linearly to the forces \( F_j \) defined in Exercise 4 of Chapter 5: \( F_j = Cq_j c_{j+1}, 1 \leq j \leq n \).

(ii) Since all \( F_j \) can be found through the coefficients \( c_2, c_3, \ldots, c_{n+1} \) of the convolution \( C(x) = A(x)B(x) \), we can use the Fast Fourier Transform divide-and-conquer algorithm to find them (along with all other coefficients \( c_k \)) in \( \Theta(n \log n) \) time.