Problem Set 2 Solutions

This problem set is due in lecture on Monday, October 1st, 2007.

Instructions:

- Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

- You will often be called upon to “give an algorithm” to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

  1. A description of the algorithm in English and, if helpful, pseudo-code.
  2. At least one worked example or diagram to show more precisely how your algorithm works.
  3. A proof (or indication) of the correctness of the algorithm.
  4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to the correct solution which are described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, prevent you from writing convoluted solutions, and also help you conceptualize the key idea of the problem.

- Be sure to know the section on “Guide to writing up homework” in the “Course Information” handout before writing your solutions.

Problem 2-1. Recurrences

Give asymptotic upper and lower bounds for \( T(n) \) in each of the following recurrences. Assume that \( T(n) \) is constant for \( n \leq 10 \). Make your bounds as tight as possible, and justify your answers.

(a) \( T(n) = 3 \cdot T(n/8) + \sqrt{n} \).

**Solution:** Since \( \log_8 3 > 1/2 \), we can use the first case of Master’s theorem, and conclude that \( T(n) = \Theta(n^{\log_8 3}) = O(n^{0.529}) \).
(b) \( T(n) = 125 \cdot T(n/5) + n^3 \).

**Solution:** Since \( \log_5 125 = 3 \), we can use the second case of Master’s theorem, and conclude that \( T(n) = \Theta(n^3 \log n) \).

(c) \( T(n) = T(n^{3/4}) + 1 \).

**Solution:** \( T(n) \) measures how many times do we need to raise \( n \) to power \( 3/4 \) before we get a constant – this is given by the equation \( n^{(3/4)x} < 10 \). The answer is \( x = \Theta(\log \log n) \) and thus \( T(n) = \Theta(\log \log n) \).

(d) \( T(n) = T(n - 6046) + 2\sqrt{n} \).

**Solution:** For simplicity, let \( c = 6046 \). Then,

\[
T(n) = 2\sqrt{n} + 2\sqrt{n-c} + 2\sqrt{n-2c} + \ldots + 2\sqrt{n-c[n/c]} + O(1)
\]

\[
= \sum_{i=0}^{[n/c]} 2\sqrt{n-ic}
\]

\[
\leq \int_0^{[n/c]} 2\sqrt{n-x} \, dx = \int_0^{[n/c]} 2\sqrt{x} \, dx
\]

\[
\leq \int_0^n 2\sqrt{x} \, dx
\]

By using substitution method for integration (\( u = \sqrt{x}, \, du = 1/2x^{-1/2} \, dx \)) and using integration by parts,

\[
T(n) \leq 2 \int_0^{\sqrt{n}} u^2 \, du = O(\sqrt{n^2\sqrt{n}}).
\]

The lower bound on \( T(n) \) can be shown as follows:

\[
T(n) = 2\sqrt{n} + 2\sqrt{n-c} + 2\sqrt{n-2c} + \ldots + 2\sqrt{n-c[n/c]} + O(1)
\]

\[
\geq 2\sqrt{n} + 2\sqrt{n-c} + 2\sqrt{n-2c} + \ldots + 2\sqrt{n-c[k]} \quad \text{where} \quad k < [n/c] \quad \text{is a constant}
\]

\[
\geq k2\sqrt{n-ck}.
\]

Now, if we choose \( k \) such that \( \sqrt{n-ck} \geq \sqrt{n} - 1 \), we are done. Therefore, \( k \leq (2\sqrt{n} - 1)/c < [n/c] \) for large enough \( n \). Let \( k = (2\sqrt{n} - 1)/c \). Then,

\[
T(n) \geq k2\sqrt{n-ck} \geq \frac{2\sqrt{n} - 1}{c}2\sqrt{n-1} = \Omega(\sqrt{n^2\sqrt{n}}).
\]
(e) \( T(n) = 5 \cdot T(\frac{1}{5}n) \cdot T(\frac{4}{5}n) \). It is fine to be only tight up to a constant in the exponent (i.e., give a bound of the type \( \Omega(f(n)^{c_1}) \leq T(n) \leq O(f(n)^{c_2}) \) for some constants \( c_1, c_2 > 0 \)).

Solution: The product can be easily reduced to a sum by taking logarithm for both sides. In particular, let \( L(n) = \log T(n) \). Then, we have the following recurrence:

\[
L(n) = L(n/5) + L(4n/5) + O(1).
\]

Drawing the recursion tree, it’s easy to note that the leaves are dominating. Thus \( L(n) = \Theta(n) \), and \( T(n) = 2^{L(n)} = 2^{\Theta(n)} \).

(f) \( T(n) = \min_{2 \leq b \leq n} 2 \cdot T(n/b) + b \). In other words, our recurrence depends on a parameter \( b \) that we may tweak in order to minimize \( T(n) = 2 \cdot T(n/b) + b \) (note that \( b \) may be a function of \( n \)). It is fine to be only tight up to a constant in the exponent (i.e., give a bound of the type \( \Omega(f(n)^{c_1}) \leq T(n) \leq O(f(n)^{c_2}) \) for some constants \( c_1, c_2 > 0 \)).

Solution: The best \( b \) is of the order \( b = 2^{\sqrt{\log n}} \). Later we show it yields \( T(n) = 2^{\Theta(\sqrt{\log n})} \). But for now, let us give some indication of how to actually come up with such a guess.

Let’s draw the recursion tree for a fixed value of \( n \). Let’s make a simplifying assumption (restriction): suppose that at each level we choose the same value of \( b \) (which, nonetheless, may be a function of the fixed value \( n \)). There are \( \log_{b} n = \frac{\log n}{\log b} \) level of recursion. The first level contributes \( b \), the second \( 2b \), ..., the \( i^{th} \) level contributes \( 2^i \cdot b \). Thus, total contribution is

\[
\sum_{i=1}^{\log_{b} n} 2^i b = \Theta(2^{\log_{b} n} \cdot b) = \exp \left[ \Theta(\frac{\log n}{\log b} + \log b) \right],
\]

where \( \exp \) is the exponential function \( \exp[x] = e^x \).

The sum inside the exponent is minimized when \( \frac{\log n}{\log b} = \log b \), meaning that \( \log b = \sqrt{\log n} \) and thus \( b = 2^{\sqrt{\log n}} \) (ignoring the non-important constant factors).

This analysis/intuition immediately gives the upper bound on \( T(n) = O(\exp \left[ \Theta(\frac{\log n}{\log b} + \log b) \right]) = 2^{O(\sqrt{\log n})} \).

Of course, the question that remains is: can we somehow win from varying the value of \( b \) over different level of the tree? The answer is no (up to variation of the constant in the exponent), and to prove this, one can employ usual induction for our guesstimate of \( T(n) = 2^{\Theta(\sqrt{\log n})} \).

In particular, let’s prove that \( T(n) \geq 2^{c \sqrt{\log n}} \) for some constant \( c \) (to be determined later). Base case works since we can just choose \( c \) sufficiently small (note that this imposes a condition on \( c \) later). Now, by inductive hypothesis, assume \( T(m) \geq 2^{c \sqrt{\log m}} \),
for \( m < n \). Then,

\[
T(n) \geq \min_b 2T(n/b) + b.
\]

If the optimal \( b \) is \( b \geq 2^c \sqrt[2c]{n} \) then we are done \( (T(n) \geq b \geq 2^c \sqrt[2c]{n}) \). Otherwise, we have that for any \( b < 2^c \sqrt[2c]{n} \), by inductive hypothesis,

\[
2T(n/b) \geq 2^c \sqrt[2c]{n/b} \geq 2^{1+c} \sqrt[2c]{n-\sqrt[2c]{n}} \geq 2^{1+c}(\sqrt[2c]{n}-c) \geq 2^c \sqrt[2c]{n} \cdot 2^{1-c^2}.
\]

For the above inequalities to hold true, we need to impose couple of constraints on \( c \).

First, the fourth inequality \( 2^{1+c}(\sqrt[2c]{n}-c) \) requires that \( \sqrt[2c]{n} - c \sqrt[2c]{n} \geq n/c \), which requires that \( c \leq \sqrt{n} - 2 \sqrt[2c]{n} \). This is not hard to satisfy once \( n \) is large enough. The second constraint comes from \( 2^c \sqrt[2c]{n} \cdot 2^{1-c^2} \geq 2^c \sqrt[2c]{n} \). As long as \( c \) is smaller than 1, we are done: \( T(n) \geq 2T(n/b) \geq 2^c \sqrt[2c]{n} \). Note that this does not contradict the earlier condition (from the base case) that \( c \) has to be “small”.

### Problem 2-2. Fast Discrete Fourier Transforms

Let \( n = 2^k \) for some positive integer \( k \). The Discrete Fourier Transform of a sequence of \( n \) integers \( \vec{x} = (x_0, x_1, \ldots, x_{n-1}) \) is defined as follows:

- For an integer \( i \in \{0, 1, \ldots, n-1\} \), let \( i_{k-1}, \ldots, i_0 \) denote the binary expansion of \( i \). (So \( i_\ell \in \{0, 1\} \) and \( i = i_0 + 2i_1 + 4i_2 + \cdots + 2^{k-1}i_{k-1} \).)

- For integers \( i, j \in \{0, 1, \ldots, n-1\} \) let the “inner product” of \( i \) and \( j \), denoted \( \langle i, j \rangle \), be the quantity \( \sum_{\ell=0}^{k-1} i_\ell j_\ell \).

- Let \( H_n \) be the \( n \times n \) matrix given by \( H_n(i, j) = (-1)^{\langle i, j \rangle} \), for \( i, j \in \{0, 1, \ldots, n-1\} \). (Note that the rows and columns of \( H_n \) are numbered from 0 to \( n-1 \) rather than the usual 1 to \( n \).)

- The Discrete Fourier Transform of a sequence \( \vec{x} \) is simply the vector \( \vec{y} = H_n \cdot \vec{x} \). (Why is this the Discrete Fourier Transform? We won’t get into the full details, but just to see the similarity with the standard Fourier Transform, note that if we had defined a Fourier matrix \( F_n \) by the formula \( F_n(i, j) = \omega^{ij} \) where \( \omega \) is a complex \( n \)th root of unity, then the standard Fourier Transform of \( \vec{x} \) would have simply been \( F_n \cdot \vec{x} \).)

In this problem your goal is to give an efficient algorithm (based on divide and conquer) that takes as input a sequence \( \vec{x} \) and outputs its Discrete Fourier Transform.

**(a)** Write down the matrices \( H_2, H_4, \) and \( H_8 \).

**Solution:**

\[
H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.
\]
(b) Give a recursive formula describing $H_n$ in terms of $H_{n/2}$.

Solution: The general formula for $H_n$ is

$$H_n = \begin{bmatrix} H_{n/2} & H_{n/2} \\ H_{n/2} & -H_{n/2} \end{bmatrix}.$$  

(c) Use the recursive formulation to describe an (efficient) algorithm to compute the Discrete Fourier Transform $H_n \cdot \vec{x}$. (Hint: Your algorithm should reduce this computation to computing $H_{n/2} \cdot \vec{u}, H_{n/2} \cdot \vec{v}, \ldots$, for a small number of sequences $\vec{u}, \vec{v}$.)

Solution: Let $\vec{x} = \vec{y} \circ \vec{z}$ where $\vec{y} = \langle x_0, \ldots, x_{n/2-1} \rangle$, $\vec{z} = \langle x_{n/2}, \ldots, x_{n-1} \rangle$ and the symbol “$\circ$” simply denotes the concatenation of sequences.

Then, we note that $H_n \cdot \vec{x}$ is simply $(H_{n/2} \cdot \vec{y} + H_{n/2} \cdot \vec{z}) \circ (H_{n/2} \cdot \vec{y} - H_{n/2} \cdot \vec{z}) = H_{n/2} \cdot (\vec{y} + \vec{z}) \circ H_{n/2} \cdot (\vec{y} - \vec{z})$.

This leads to the algorithm

\begin{align*}
\text{FDFT}(n, \vec{x} = \langle x_0, \ldots, x_{n-1} \rangle) \\
\text{If } n = 1 \text{ Return } \vec{x}; \\
\quad \vec{y} \leftarrow \langle x_0, \ldots, x_{n/2-1} \rangle \\
\quad \vec{z} \leftarrow \langle x_{n/2}, \ldots, x_{n-1} \rangle \\
\quad \vec{u} \leftarrow \vec{y} + \vec{z} \\
\quad \vec{v} \leftarrow \vec{y} - \vec{z} \\
\quad \vec{s} \leftarrow \text{FDFT}(n/2, \vec{u}) \\
\quad \vec{t} \leftarrow \text{FDFT}(n/2, \vec{v}) \\
\text{Return } \vec{s} \circ \vec{t}.
\end{align*}
(d) Analyze the running time of your algorithm

Solution:
Let \( T(n) \) denote the running time on inputs of length \( n \). The preprocessing before the
recursive calls takes \( O(n) \) time, and there are two recursive calls of length \( n/2 \). So
we get the familiar recurrence
\[
T(n) = 2T(n/2) + \Theta(n)
\]
with \( T(1) = \Theta(1) \). As in Mergesort, this yields a running time bound of
\( T(n) = \Theta(n \log n) \).

Problem 2-3. Order statistics in several sorted arrays

Suppose you are given a positive integer \( i \) and \( k \) arrays, each of \( n \) elements, where each array is
sorted. The goal of this problem is to obtain some fast algorithms of finding the \( i \)th element in the
union of the \( k \) arrays.

In what follows, assume \( k \) is a constant, while \( n \) is growing. Assume the input is given in the form
of \( k \) arrays \( A_1, \ldots, A_k \). To simplify things, you can also assume that the elements are not repeated,
i.e., each array has distinct elements and every element appears in at most one array.

Note that when \( k = 1 \), this problem can be solved in 1 step.

(a) Give an \( O(\log n) \) time algorithm when \( k = 2 \). Specifically, give an algorithm 2-
Select \( (A_1, A_2, i) \) that finds the \( i \)th ranked element in the union of two sorted arrays
\( A_1 \) and \( A_2 \). Hint: If you know that the \( i \)th element is in the range \( A_1[L..R] \). What
can you say about its location in \( A_2 \)?

Solution: If the \( i \)th ranked element is between \( A_1[L..R] \) then must also be between
\( A_2[i - R..i - L] \). For ease of notation, below we consider that if the index \( j < 1 \),
then \( A_1[j] = -\infty \), and if \( j > n \), then \( A_1[j] = +\infty \) (and same for \( A_2 \)).

We can search for the \( i \)th ranked element by a binary search on \( A_1 \). Suppose we know
the \( i \)th ranked element is between \( A_1[L] \) and \( A_1[R] \).

- Let \( M = [(L + R)/2] \).
- If \( A_2[i - M] < A_1[M] < A_2[i - M + 1] \) then \( A_1[M] \) is the \( i \)th ranked element.
- If \( A_1[M] < A_2[i - M] < A_1[M + 1] \) then \( A_2[i - M] \) is the \( i \)th ranked element.
- If \( A_1[M] > A_2[i - M + 1] \) then the \( i \)th ranked element is in the range \( A_1[L..M] \).
  Set \( R \leftarrow M \) and repeat.
- If \( A_2[i - M] > A_1[M + 1] \) then the \( i \)th ranked element \( i \)th ranked element is in the
  range \( A_1[M + 1..R] \). Set \( L \leftarrow M + 1 \) and repeat.

If we start with \( L \leftarrow 0 \) and \( R \leftarrow n \), then each iteration takes \( \Theta(1) \) time and reduces
\( R - L \) by a factor of 2. Thus the algorithm terminates in \( O(\log n) \) time.
Now we start working on general $k$. To start off, design a helper function, $\text{RANK}(A_1, \ldots, A_k, x)$ which returns the rank of the element $x$ in the union of the $k$ arrays. (I.e., it returns the number of elements smaller than $x$ in the union of the $k$ arrays). Your function should run in $O(k \log n)$ time.

**Solution:** For each of the $k$ arrays, employ the binary search to find the number of elements smaller than $x$. Then the total rank of $x$ is the sum of these numbers.

(c) Now give an $O(k^2(\log n)^2)$-time algorithm $k$-$\text{SELECT}(A_1, \ldots, A_k; i)$ that returns the $i$th ranked element in the union of the $k$ arrays. (Hint: First give an $O(k(\log n)^2)$-time algorithm that, given an array index $j$, returns the location of the $i$th ranked element in $A_j$. I.e., it returns an index $\ell$ such that the $i$th ranked element $x$ satisfies $A_j[\ell] \leq x \leq A_j[\ell + 1]$.)

**Solution:** We first design $k$-$\text{SELECTHELPER}(A_1, \ldots, A_k; i, j)$ that returns $\ell$, the location of the $i$th ranked element in $A_j$. $k$-$\text{SELECTHELPER}(A_1, \ldots, A_k; i, j)$ simply does a binary search for the $i$th ranked element in $A_j$ using $\text{RANK}$ as a subroutine. Specifically it maintains an interval $L \cdots R$ that contains $\ell$. It then checks the $\text{RANK}$ of $A_j[(L + R)/2]$ and if this is less than $i$ then it sets $R$ to $(L + R)/2$ and otherwise it sets $L$ to $(L + R)/2$ and recurses. It stops when $R = L + 1$. It is clear that $k$-$\text{SELECTHELPER}$ makes $\log n$ calls to $\text{RANK}$ and so runs in $O(k(\log n)^2)$ time.

Once we have $k$-$\text{SELECTHELPER}$ it is easy to find the $i$th element. Simply run $k$-$\text{SELECTHELPER}$ for every $j \in [1 \ldots k]$ and let $\ell_1, \ldots, \ell_k$ be the answers returned. Now check if $\text{RANK}$ of $A_j[\ell_j]$ or $A_j[\ell_j + 1]$ is $i$ for every $j \in [1 \ldots k]$. Return the element for which this is true.

The first step involves $k$ calls to $k$-$\text{SELECTHELPER}$ and thus takes $O(k^2(\log n)^2)$-time. The second set involves $2k$ calls to $\text{RANK}$ and so terminates takes $O(k^2 \log n)$-time, leading to an overall running time bound of $O(k^2(\log n)^2)$.

**Alternative solution.** We also give an outline of an alternative solution that gives $O(k^2 \log n)$ time, and, if using heaps, even $O(k \log k \cdot \log n)$. Suppose we look for the $i$th ranked element in $k$ arrays of lengths $a_1, a_2, \ldots a_k$. We will perform a number of iterations; in each iteration we will half one of the array, zooming in on the interval where the $i$th ranked element might be. Initially, all arrays are of length $n$.

In one iteration, we consider the median of each of the $k$ arrays. Let these medians be $m_1, \ldots m_k$. Sort the medians, and, wlog, suppose $m_1 < m_2 < \ldots m_k$. Then, if $i \geq \frac{1}{2}(a_1 + a_2 + \ldots a_k)$, we can half the first array (containing $m_1$), and, specifically, disregard the first half of the array. It is relatively simple to prove that the first half of the first array cannot contain the $i$th ranked element. Note that we also need to update $i$ for the next iteration (specifically, decrease $i$ by $a_1/2$). Otherwise, if $i < \frac{1}{2}(a_1 + a_2 + \ldots a_k)$, we can half the last array (containing $m_k$), and, specifically, disregard the second half of the array. If the halved array contains only one element, we “delete” the array altogether, ignoring it for the rest of the algorithm.
After $O(k \log n)$ of such iterations, all arrays will have a constant number of elements, when we can find the $i^{th}$ ranked element in $O(k)$ time.

How much time is needed per iteration? At the beginning, we need $O(k \log k)$ time to sort the list of medians $m_1, m_2, \ldots, m_k$. After that, at each iteration we need to update the list – for this we delete either the minimum or the maximum element of the list (depending on whether we half the first or the last array), and insert some other element (the new median of the halved array). This update can be done in $O(k)$ time; then the total time is $O(k^2 \log n)$. It is also possible to keep the list as a heap, when the update will take only $O(\log k)$ time; the total time then becomes $O(k \log k \cdot \log n)$.

(bonus Question.) $O(k^2 (\log n)^2)$ is not the best running time bound one can get for this problem. It is possible to design an $O(k \log n)$-expected time randomized algorithm also. For bonus credit you may try to design such an algorithm. (Warning: The algorithm may be quite complex to describe. If you choose to attempt this problem, you must not only get the algorithm right, but also give a clear writeup.)

Solution: First we show an approach that gives $O(k \log^2 n)$. For each array we keep an “active interval”, an interval that sandwiches the median. Formally, we say an interval that starts at $e_1$ and ends at $e_2$ sandwiches the median if $\text{Rank}(e_1) \leq N/2$ and $\text{Rank}(e_2) > N/2$. Let $\text{activeInterval}_i$ be the length of the active interval of the array $i$ (including the endpoints).

We will do $O(\log n)$ steps of contracting (cutting off) these intervals until we have identified the median (when all active intervals become of length 2, one of the endpoints have to be the median).

At each step pick a random element $r$ from the union of the active intervals (using the primitive from the previous part). Compute the rank of $r$; we are done if $\text{Rank}(r) = N/2$.

Otherwise, we want to restrict the active intervals to be left or right of $r$, depending on whether $\text{Rank}(r) > N/2$ or $\text{Rank}(r) < N/2$. If $\text{Rank}(r) > N/2$, we need to update the right-end points of the active intervals so that all elements of the active intervals are smaller than $r$. If $\text{Rank}(r) < N/2$, we need to update the left-end points of the active intervals. Each updating is done via simple binary search.

Note that each step takes $O(k \log n)$ time. Also, as in the quicksort, we expect to have $O(\log n)$ steps in total.

Thus, the total time is $O(\log n \cdot k \log n) = O(k \log^2 n)$.

To improve the running time, we observe is that we do not need to compute $\text{Rank}(r)$ exactly, but to some approximation only. Even more importantly, it is fine to cut down the active intervals only approximately – as long as we cut a constant fraction of the total length of the active intervals, we make some progress.

Suppose for the moment that we choose some $r$ and we know that $\text{Rank}(r) > N/2$. 

Let’s define the quantity \( cut_i \), which, intuitively, will express the amount that we approximately “cut” from the active interval by \( r \). Specifically, consider some array with an active interval \( e_1, \ldots, e_2 \) of length \( activeInterval_i \), and the random element \( r \). Perform only the first 10 steps of the binary search of the “update” process (in the previous part). Let \( cut_i \) be the number of elements that are already certain to not appear in the updated active interval (i.e., the elements to the right of the current upper bound on the position of \( r \) inside the active interval). Expected value of \( cut_i \) is at least \( 1/5 \cdot activeInterval_i \).

Thus, expected value of \( \sum_i cut_i = \sum_i activeInterval_i \cdot 1/5 \). Then, with probability at least \( 1/5 \), \( \sum_i cut_i \geq \sum_i activeInterval_i \cdot 1/5 \), i.e., we “cut off” a constant fraction of elements in the active intervals.

Now, to guarantee this event, we can repeat choosing \( r \) until the event actually happens (instead of going with the initial choice of \( r \)).

We also need to ensure that \( r \) is such that \( \text{Rank}(r) > N/2 \). We can do so by the same procedure as above. Specifically, we modify the \( \text{Rank} \) procedure to perform a constant number of binary search steps in each of the array. With the same analysis as above, we can ensure that the rank of \( r \) is at least \( N \cdot (1/2 + 1/5) \), which is enough for the above purposes.

The total running time is \( O(k \log n) \): there are \( O(\log n) \) steps (since in each step we guarantee to cut the total length of active intervals by a constant fraction), and each step takes \( O(k) \) in expectation (in expectation because we may choose bad \( r \), in which case we repeat).