Problem Set 1 Solutions

This problem set is due at 11:59pm on Wednesday, February 20, 2013.

Problem 1-1. Stitch and Wave Sorts

Unlucky Louie is trying to come up with a new divide-and-conquer sorting algorithm. His ultimate goal is to make it as efficient as possible.

(a) His first attempt, which he calls “stitch sort,” sorts a list $L$ of length $n$ using the following recursive algorithm (for which only the recursive case is shown):

Stitch-sort the first $n/2$ elements of $L$, then stitch-sort the last $n/2$ elements, and then stitch-sort the middle $n/2$ elements. Then, stitch-sort the first $n/2$ elements again, then the last $n/2$ elements again, and finally the middle $n/2$ elements again.

Find the running time of this algorithm.

**Solution:** The running time $T(n)$ is given by $T(n) = 6T(n/2) + \Theta(1)$. Using the Master Theorem, we get $T(n) = \Theta(n^\log_6^6)$, which is approximately $\Theta(n^{2.585})$.

(b) Prove by induction that stitch sort is a correct sorting algorithm. You may assume that $n$ is a power of 2.

**Solution:** We prove this for $n = 2^k$ by induction on $k$.

**Base cases:** $k = 0$ means $n = 1$, which is trivially true. $k = 1$ means $n = 2$, which we will treat as a base case in the algorithm: if a list containing two elements is not sorted, swap the two elements.

**Inductive step:** Assume by induction that stitch sort works on all $2^{k-1}$-element lists. We wish to show that the algorithm works on an arbitrary list $L$ of $n = 2^k$ elements. We identify elements by their ranks, i.e., their positions in the intended sorted list.

We first show that all ranks between 0 and $n/4 - 1$ are eventually put in the correct place. We will call such ranks low ranks. We start by showing that they will all end up in the first $n/4$ slots in $L$. Whenever we do a recursive stitch sort on an $n/2$-element section $L'$ of $L$, any low rank, $m$, must end up in the lower half of $L'$; otherwise it would mean that there are at least $n/4$ elements less than $m$, which is clearly false.

After the second subsort, any low ranks in the uppermost $n/4$ slots in $L$ will get pushed into the second-highest $n/4$ slots in $L$. After the third subsort, all low ranks in the second-highest $n/4$ slots will get pushed down into the second-lowest $n/4$ slots, putting all low ranks into the lower half. After the fourth subsort, all $n/4$ low ranks...
Problem 1 Solutions

will get pushed into the lowest \( n/4 \) slots in \( L \), and they will be sorted. Therefore all low ranks will eventually get put in the correct place.

Similarly, one may show that the first, third, and fifth subsorts will eventually push all ranks between \( 3n/4 \) and \( n - 1 \) inclusive into the highest \( n/4 \) slots in \( L \) and that they will also be in their correct positions.

This leaves the ranks between \( n/4 \) and \( 3n/4 - 1 \) inclusive. Since the other ranks are already in the correct places, these remaining ranks must occupy the middle \( n/4 \) slots in \( L \). The sixth subsort will sort them, resulting in a fully sorted list. Therefore stitch sort works on lists of length \( n \), and the induction is complete.

(c) Disappointed with the poor performance of stitch sort, Louie tries to come up with another algorithm. He believes that the problem comes from the fact that there are too many subproblems in his recursion. He therefore comes up with a new algorithm, which he calls “wave sort,” which again sorts a list \( L \) of length \( n \):

Wave-sort the first \( 3n/5 \) elements of \( L \), then the middle \( 3n/5 \) elements, then the last \( 3n/5 \) elements, then the middle \( 3n/5 \) elements again, and finally the first \( 3n/5 \) elements again.

Find the running time of this algorithm.

**Solution:** The running time \( T(n) \) is given by \( T(n) = 5T(3n/5) + \Theta(1) \). Using the Master Theorem, we get \( T(n) = \Theta(n^{\log_{3}5}) \), which is approximately \( \Theta(n^{3.151}) \).

(d) Which algorithm performs better, stitch sort or wave sort? Explain why Louie’s strategy for improving asymptotic performance in part (c) did or did not work.

**Solution:** Stitch sort performs better. With each level of recursion, we increase the total size of the subproblems by a factor of 3. However, the recursion tree for wave sort requires more levels of recursion to get down to base cases, so it will take longer to cover all of them. This is why Louie’s strategy didn’t work.

(Of course, we use the term “better” rather loosely here, especially since other sorting algorithms like mergesort are much better!)

Problem 1-2. Master Theorem

In this problem you will practice solving more recurrences, and you will explore some of the limitations of the Master Theorem.

For the following recurrences, use the Master Theorem to come up with a tight asymptotic running time for \( T(n) \). Make sure to cite which case of the Master Theorem you are using.

(a) \( T(n) = 14T(n/3) + n^3 \)

**Solution:** \( \Theta(n^3) \), by Case 3 of the Master Theorem.
(b) \( T(n) = 8T(n/3) + n^3 \)

**Solution:** \( \Theta(n^3) \), by Case 3 of the Master Theorem.

Even when the Master Theorem is not directly applicable, it may come still come in handy when proving certain bounds. Use the Master Theorem to come up with an \( O(\cdot) \) bound on \( T(n) \).

(c) \( T(n) = 4T(n/2) + (n/ \log \log n)^2 \)

**Solution:** Observe that the recurrence \( T'(n) = 4T'(n/2) + n^2 \) does more work at each level than \( T(n) \) and they have the same number of levels. This implies that \( T(n) = O(T'(n)) \). By Case 2 of the Master Theorem, we get that \( T'(n) = n^2 \log n \), which in turn means that \( T(n) = O(n^2 \log n) \).

For the following recurrences, come up with the best \( O(\cdot) \) bound you can. Use any means taught in class or Chapter 4 of CLRS.

(d) \( T(n) = 2T(\sqrt{n}) + \log n \)

**Solution:** \( O(\log n \log \log n) \). Substitute \( k = \log n \), and then use \( U(k) = T(2^k) \). This leads to \( U(k) = 2U(k/2) + k \), which by the Master Theorem gives us \( U(k) = \Theta(k \log k) \). Translating back, we get \( T(n) = U(k) = \Theta(\log n \log \log n) \).

(e) \( T(n) = 2T(n/10) + (n^2)! \)

**Solution:** \( T(n) = \Theta((n^2)!) \) by Case 3 of the Master Theorem.

(f) \( T(n) = 2T(n-1) + 3T(n-2) + 1 \)

**Solution:** The solutions to the associated quadratic equation are \(-1 \) and \( 3 \), leading to a general solution of the form \( O(3^n + (-1)^n) = O(3^n) \).

(g) \( T(n) = 3T(n/2) + n + \log n \)

**Solution:** \( T(n) = n^{\log_2 3} \), by Case 1 of the Master Theorem.

(h) \( T(n) = 2T(n/4) + T(n/2) + n \)

**Solution:** By the recursion tree method, each level does \( n \) work, and we have \( \Theta(\log n) \) levels, resulting in \( \Theta(n \log n) \) total work.
Problem 1-3. Algorithmic Physics

Fergilab needs your help! Their theoretical physicists have developed a brand-new analytic formula for the interactions among a large number of charged Fergi particles, and their experimental physicists wish to verify their prediction. Specifically, their experimental setup consists of $n + 1$ Fergi particles (labeled 0 through $n$) equally spaced along a straight line. Particle $j$ has (possibly negative) charge $q_j$ and sits at location $j$ (so the distance between location $i$ and location $j$ is exactly $i - j$ units). The experiment then computes the force $F_j$ exerted on particle $j$, while the theory predicts that

$$F_j = \sum_{i<j}^{} q_i q_j (i - j)^2 - \sum_{i>j}^{} q_i q_j (i - j)^2.$$ 

Unfortunately, while the experiment runs quickly, the physicists at Fergilab compute the predicted $F_j$ using a rather slow $\Theta(n^2)$-time algorithm that computes each of the $n$ $F_j$s independently in $\Theta(n)$ time each. Help them out by providing an algorithm that computes all predicted $F_j$ in total time $O(n \log n)$.

Solution: We will solve the problem by making use of the FFT. Consider the following two vectors:

$$\vec{a} = (q_0, \ldots, q_n)$$
$$\vec{b} = (-1/n^2, -1/(n - 1)^2, \ldots, -1, 0, 1, \ldots, 1/n^2)$$

In other words, $a_i = q_i$ for all $i \in \{0, 1, \ldots, n\}$, $b_i = -1/(n - i)^2$ for $i \in \{0, 1, \ldots, n - 1\}$, $b_n = 0$. And $b_i = 1/(i - n)^2$ for $i \in \{n + 1, n + 2, \ldots, 2n\}$. Using the FFT, we can take compute the convolution of $\vec{a}$ and $\vec{b}$ in time $O(n \log n)$ to obtain $\vec{c}$, where

$$c_k = \sum_{i, j | i+j=k} a_i b_j.$$ 

Now let’s see what information we can obtain from $\vec{c}$. Fix some $k \in \{n, n + 1, \ldots, 2n\}$ and consider $c_k$. Using the definition of $\vec{a}$ and $\vec{b}$ and the formula for $c_k$ above, we see that:

$$c_k = a_0 b_k + \cdots + a_n b_{k-n}$$
$$= q_0/(k - n)^2 + \cdots + q_{k-n-1} + 0 - q_{k-n+1} - \cdots - q_n/(2n - k)^2$$

Now observe that we can rewrite:
\[ F_j = \sum_{i<j} \frac{q_i q_j}{(i-j)^2} - \sum_{i>j} \frac{q_i q_j}{(i-j)^2} \]

\[ = q_j \left( \sum_{i<j} \frac{q_i}{(i-j)^2} - \sum_{i>j} \frac{q_i}{(i-j)^2} \right) \]

\[ = q_j \left( q_0/j^2 + \cdots + q_{j-1} + 0 - q_{j+1} - \cdots - q_n/(n-j)^2 \right) \]

\[ = q_j \cdot c_{n+j} \]

So we have now observed that \( F_j = q_j c_{n+j} \). We computed all \( c_{n+j} \) in time \( O(n \log n) \) using the FFT. Because each \( q_j \) is given, we can now do an extra \( n \) multiplications to obtain each \( F_j \).

**Problem 1-4. Problem Set Scheduling**

Your friend is taking an experimental new algorithms class, 6.0046, in which problem sets can be posted on any day of the week and can be due on any day of the week. The professors are not allowed to assign overlapping problem sets; the due date for one problem set must be before (or on the same day as) the posting date of the next problem set. Unfortunately, given the experimental nature of the class, the professors frequently change their minds by adding a new problem set, splitting an existing problem set into two smaller ones, or merging two small problem sets into a massive one.

Your goal is to develop a data structure maintaining the current state of the problem sets. The query you need to support is, given a date, to efficiently determine whether your friend should be working on a problem set on that date (meaning that it’s been posted but isn’t yet due), and if so, to figure out which problem set. Instead of identifying problem sets by number, you can just identify them by an ordered pair of the posting date and the due date. (If the date has both a problem set due and another one assigned, you should identify both problem sets.)

In other words, you need to support the following operations:

- **ADD-PROBLEM-SET\((a, b)\)** creates a new problem set posted on date \( a \) and due on date \( b \).
- **SPLIT-PROBLEM-SET\((a, b, k)\)** means that the problem set originally posted on date \( a \) and due on date \( b \) is now split into two problem sets: one posted on \( a \) and due on \( k \), plus one posted on \( k \) and due on \( b \).
- **MERGE-PROBLEM-SETS\((a, b, c)\)** merges the problem set posted on \( a \) and due on \( b \) with the problem set posted on \( b \) and due on \( c \); the new massive problem set is posted on \( a \) and due on \( c \).
- **QUERY-DATE\((k)\)** asks you to return the set of all problem sets \((a, b)\) posted on \( a \) and due on \( b \) that satisfy \( a \leq k \leq b \).
Problem Set 1 Solutions

Assume that dates have been numbered 0, 1, . . . , u − 1. For your convenience, you can assume that all operations are called with valid arguments. (For instance, ADD-PROBLEM-SET(a, b) will never be called with a date range that overlaps with an existing problem set.)

Your goal is to implement all operations above in $O(\lg \lg u)$ time per operation.

Solution: One way is to create two van Emde Boas trees, $A$ and $B$. $A$ will store posted dates, and $B$ will store due dates.

Because problem sets cannot overlap, you can uniquely correspond posted dates from $A$ with due dates from $B$; the posted date $a$ corresponds to the smallest entry in $B$ that’s greater than $a$.

- ADD-PROBLEM-SET($a$, $b$): Insert $a$ into $A$ and $b$ into $B$.
- SPLIT-PROBLEM-SET($a$, $b$, $k$): Insert $k$ into $A$ and $B$.
- MERGE-PROBLEM-SETS($a$, $b$, $c$): Delete $b$ from $A$ and $B$.
- QUERY-DATE($k$): If $k$ is in $A$, then a problem set was posted on this date; find the successor of $k$ in $B$ for the corresponding due date. If $k$ is in $B$, then a problem set is due on this date; find the predecessor of $k$ in $A$ for the corresponding posted date. (The first two cases are not necessarily mutually exclusive.) If neither of the first two cases occurred, then we still need to check for problem sets extending before/after this date. Let $a$ be the predecessor of $k$ in $A$, and let $b$ be the successor of $a$ in $B$. If $k$ is in the range ($a$, $b$), then $a$ is the posted date and $b$ is the due date. Return all problem sets found by this process, or the empty set otherwise.

Another solution is to use just a single van Emde Boas tree, and to augment each entry with two bits: one indicating whether a problem set was posted on that date, and another indicating whether a problem set was due on that date. For every date in the tree, at least one (and possibly both) bit must be set. The operations are very similar. However, you do need to prove (or at least sketch) why augmenting with these two extra bits does not change the asymptotic running time.