Due: Thu Sep 15, in class.
HOMEWORK with SOLUTION, prepared by Instructor and T.A.s

INSTRUCTIONS:

• Remember that we have a “no late homework” policy. Special permission must be obtained in advance if you have a valid reason.

• Only hand in answers to the questions with positive points attached to them. Do the others at your leisure.

• All the exercise numbers below refer to Lecture I.

1. (4 Points) We interpreted the programs in Figure 1(a) and (b) as algorithms for finding the maximum of \( \{x, y, z\} \).
   But, the notion of an “algorithm” is a semantical concept, and so can be given different interpretations. Please give a different interpretation to these two programs. I.e., view them as solving a problem other than finding the maximum.

   NOTE: For this question, we regard the output at each leaf of a comparison tree as an “interpretation”. So we allow you to change the output at each leaf (but the tree program and comparisons at each internal node is unchanged).

   **SOLUTION**

   Consider the problem \( P' \) of computing any non-maxima of \( x, y, z \). We can interpret both programs as algorithms for this non-maxima problem \( P' \). We just have to change the output to return any element different than the maximum. In Figure 1(b) for the circuit model, we had tacitly assume the output element is \( x' \). Now, we can assume the output element is \( z' \) (or \( y' \)).

   Actually, there is a problem with this interpretation when not all the input elements are distinct. For instance, if \( x = y = z \), which element do we output as “non-maxima”? You need to address this issue explicitly.

   One simple solution is to declare that the elements must be all distinct. Another (better) solution is to declare that if \( x = y \) then \( y \) should be considered as “smaller” than \( x \). More generally, identify \( (x, y, z) \) with \( (x_1, x_2, x_3) \), and if \( x_i = x_j \) then we break-ties by declaring “\( x_i < x_j \)” iff \( i > j \). Now, we know how to specify the correct output regardless of the numerical values of \( x_1, x_2, x_3 \).

   For the circuit model, when two input wires to a comparator has the same numerical value, we assume the element in the top input wire is carried out to the top output wire. With this convention, we can now specify that the output wire \( z' \) is a non-maxima.

   **Comments:**

2. (4 Points) Give an upper and a lower bound for \( S(1000) \), the complexity of sorting 1000 elements. Note: we are asking for two numbers, but you must justify how you obtain these two numbers.
The lower bound is given by the ITB: $S(1000) \geq \lg(1000!)$ (must be base 2). But you need to give an explicit number, not an expression like “$\lg(1000!)$”. You can use a calculator or write a program (any language you like) to compute the numerical value of $\lg(1000!)$. It seems best to use form of Stirling’s formula found in my Lecture II: $n! = (n/e)^n \sqrt{2\pi n}e^{\alpha(n)}$ where $\frac{1}{12n+1} < \alpha(n) < \frac{1}{12n}$. Taking natural logs, I obtain

\[
\begin{align*}
\lg(n!) &= n(\lg n - \lg e) + \frac{1}{2} \lg(2\pi n) + \alpha(n), \\
\lg(1000!) &\geq 8523.0892 + 12.6172 + 0.00012 \\
&\geq 8535.7066
\end{align*}
\]

Note that since I want a lower bound, I simply truncate the values I get on my calculator (assuming that my calculator is accurate to the last digit). Thus, we conclude that $S(1000) \geq 8536$.

Next we need an upper bound on $S(1000)$. It must come from some algorithm. We know that the Mergesort algorithm makes at most the following number of comparisons, assuming $n$ is a power of 2 (so when we divide by 2, we get another power of 2, etc):

\[
T(n) = 2T(n/2) + n
\]

We remark that you can actually write the recurrence as $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n - 1$, and take advantage of this to get tighter bounds for evaluating $T(n)$ at a particular value of $n$ (e.g., $n = 1000$) as seen below. If $n = 2^k$, this has solution $T(n) = 2^k T(n/2^k) + kn$ where $k = \lg n$. Since $T(n/2^k) = T(1) = 1$, we obtain the following (crude) upper bound:

\[
S(1000) \leq S(1024) = S(2^{10}) \\
\leq 10 \cdot 2^{10} = 10240
\]

By expanding the merge sort recurrence (see below), we get the much better bound of 8594.

---

You must be careful of some pitfalls.

1. First of all, you must not confuse natural logs with logs base 10 or base 2.
2. If you tried to compute the value $1000!$ before taking the natural log, your value will overflow because $1000!$ needs over 8000 bits (and most calculators or computer programs uses 64 bits or at most 128 bits in their calculation).
3. To get the upper bound, you can use any algorithm, but hopefully one that is $O(n \log n)$. But it is not so simple – there is a hidden constant in the big-Oh notation. So you need to think through how to get the upper bound. Here is one clever way by someone in our class:

The student computes an upper bound on $S(1000)$ by expanding the “exact” Mergesort recurrence $T(n) = 2T(\lceil n/2 \rceil) + n - 1$, as follows (I paraphrase him a bit):

\[
S(1000) \leq 2S(500) + 999 \\
\leq 4S(250) + 999 + 2(499) \\
\leq 8S(125) + 999 + 2(499) + 4(249) \\
\leq 16S(63) + 999 + 2(499) + 4(249) + 8(124) \\
\leq 32S(32) + 999 + 2(499) + 4(249) + 8(124) + 16(62) \\
\leq 64S(16) + 999 + 2(499) + 4(249) + 8(124) + 16(62) + 32(31) \\
\leq 128S(8) + 999 + 2(499) + 4(249) + 8(124) + 16(62) + 32(31) + 64(15) \\
\leq 256S(4) + 999 + 2(499) + 4(249) + 8(124) + 16(62) + 32(31) + 64(15) + 128(7) \\
\leq 512S(2) + 999 + 2(499) + 4(249) + 8(124) + 16(62) + 32(31) + 64(15) + 128(7) + 256(3) \\
= 1 + 999 + 2(499) + 4(254) + 8(124) + 16(62) + 32(31) + 64(15) + 128(7) + 256(3) \\
= 9000 - 406 = 8594.
\]

Note that we are using $S(125) \leq 2S(63) + 124$ in the last line since 125 is odd. And so on. This is a much better upper bound than previous one.
3. (12 Points) In the recitation, we described an algorithm for merging two sorted lists in our Tape Model. Recall that the tape primitives are \( \text{READ}(T, x) \), \( \text{WRITE}(T, x) \) and \( \text{RESET}(T) \) where \( T \) is a tape, and \( x \) a variable to store an item. We also have a predicate \( \text{EOT}(T) \) that is true if the head of \( T \) is at the end-of-tape. In this exercise, we ask you to design a Tape Algorithm to sort. Assume that the input tape is \( T_0 \) containing a sequence of \( n \) items. Finally, you must output the sorted items in tape \( T_0 \). Besides \( T_0 \), it is sufficient to have two other tapes, but any number of intermediate tapes you like.

HINT: use some form of Merge Sort. One important concept is the notion of a run which is any longest contiguous sequence of items in a tape that is non-decreasing. E.g., \((1, 8, 2, 5, 9, 4)\) has three runs: \((1, 8), (2, 5, 9), (4)\). You want to merge the runs. But first you need to “distribute” the runs in the input tape into two other tapes, and then merge them.

**SOLUTION** The input tape \( T_0 \) originally has \( m \) runs, where \( m \leq n \) and there are \( n \) items. Our goal in this algorithm is to minimize the number of \( \text{RESET} \)’s. In fact, we need at most \( \lg m \) \( \text{RESET} \)’s. There are two basic steps: Distribute and Merge. You alternately run the Distribute and Merge Steps until only one run is left.

(1) **In the Distribute Step**, we start with \( m \geq 1 \) runs in tape \( T_0 \), we want to put \( \lceil m/2 \rceil \) runs into \( T_1 \) and \( \lfloor m/2 \rfloor \) runs into \( T_2 \). Here is a simple solution:

```
DISTRIBUTE ALGORITHM:
> Initialization: set-up \( x_1, b_1 \) (\( i = 1, 2 \))
    \text{RESET}(T_0), \ \text{RESET}(T_1), \ \text{RESET}(T_2).
    b_0 \leftarrow \text{true}. \ \langle \text{If } b_0 \text{ is true, } x_0 \text{ has an item to be output.} \rangle
    \text{If } \text{EOT}(T_0) \text{ then } b_0 \leftarrow \text{false}
    \text{else } \text{READ}(T_0, x_0).
    \text{runFlag} \leftarrow \text{true}. \ \langle \text{If runFlag is true, output to } T_1, \text{ else to } T_2 \rangle
    \text{lastItem} \leftarrow -\infty. \ \langle \text{If } x_0 \text{ is less than lastItem then start a new run.} \rangle

> Main Loop: \( b_0 \) is true
    \text{while } (b_0)
        \text{If } (x_0 < \text{lastItem}) \text{ then } \text{runFlag} \leftarrow \neg \text{runFlag} \ \langle \text{Flip runFlag value} \rangle
        \text{If } (\text{runFlag}) \text{ then } \text{WRITE}(T_1, x_0).
        \text{else } \text{WRITE}(T_2, x_0).
        \text{If } \text{EOT}(T_0) \text{ then } b_0 \leftarrow \text{false}
        \text{else } \text{READ}(T_0, x_0).
        \text{lastItem} \leftarrow x_0. \ \langle \text{Update lastItem} \rangle
```

(2) **The Merge Step** is a simple generalization of our Tape Merge Algorithm in the Lecture Notes. We assume that \( T_1 \) and \( T_2 \) has an “almost” equal number of runs (either they are equal or \( T_1 \) has one more run than \( T_2 \)). For each \( k = 1, 2, 3, \ldots \), you merge the \( k \)th run of \( T_1 \) with the \( k \)th run of \( T_2 \) and store the result into \( T_0 \). You just need to detect when a run in either tape has ended, and modify the original algorithm accordingly. The upshot of this Merge Step? If there is a total of \( m \) runs in \( T_1 \) and \( T_2 \), you now have \( \leq \lceil m/2 \rceil \) runs in \( T_0 \).

We remark that there is no explicit marker in the tapes to tell you when one run ends and the next run begins. But it is easy to track this information. The Distribute algorithm below shows how this is done.
(3) **Putting it together.** Thus we do Distribute, Merge, Distribute, Merge, etc. How do we know when to stop? A simple way is to keep track of the number of runs during the Distribute Step (it is trivial to modify the above algorithm to do this). At the end of the Distribute Algorithm, if you found that there was only one run, you stop! The answer is already found in $T_0$.

Why is it $O(\log m)$ RESET’s? Intuitively, it is clear that you roughly half the number of runs in each Distribute-Merge Step, so at most $\log m$ RESET’s are needed. Each Distribute-Merge Step needs 6 RESET’s.

If you have another extra tape $T_3$, then as you merge from $T_3$ and $T_2$, you can put the merged runs into $T_0$ and $T_3$ alternately. This saves half of the number of RESET’s. Detecting stopping is only slightly more complicated.

But with 4 tapes, you can do a different trick to reduce the number of RESET’s, by doing a 3-way merge. I.e., given that the tapes $T_1, T_2, T_3$ have “almost” equal number of runs each, you can merge their contents into $T_0$ with only 4 resets. The number of Merge Steps is now $\log_3 n$.

4. (2 Points) Our asymptotic notations falls under two groups: $O, \Omega, \Theta$ and $o, \omega$. In the first group, we have $\Theta(f) = O(f) \cap \Omega(f)$. This suggests the “small-theta” analogue for the second group, “$\theta(f) = o(f) \cap \omega(f)$”. Why was this not done?

**SOLUTION** Recall that $o(f)$ comprises those $g$ such that for all $C > 0, Cf \geq g \geq 0$ (ev.). Likewise $\omega(f)$ comprise those $g$ such that for all $Cg \geq f \geq 0$ (ev.). This means that if $g \in o(f) \cap \omega(f)$, then

$$\forall C > 0)[Cf \geq g \geq f/C](ev.).$$

Now, if $f$ is the 0 constant, then this statement implies $g = 0$ (ev.). But such $g$’s are essentially equal to the 0 constant, and not very interesting. If $f$ is non-zero infinitely often, then this statement can never be satisfied by any $g$. THE ONLY EXCEPTION is when $g \uparrow$ (ev.) (eventually undefined). But this is NOT a uninteresting class of functions. Thus $o(f) \cap \omega(f)$ is not an interesting concept.

**Comments:**

5. (4 Points) Let $P : D \rightarrow \{0, 1\}$ be a partial predicate over some domain $D$. When we do quantification, $\forall x$ and $\exists x$ it is assumed that $x$ range over $D$. Show that the following equivalences (called “de Morgan’s laws for quantifiers”) hold:

(a) $\neg(\forall x)P(x)$ is equivalent to $(\exists x)\neg P(x)$
(b) $\neg(\exists x)P(x)$ is equivalent to $(\forall x)\neg P(x)$

**SOLUTION** Note that (a) and (b) is well-known when $P$ is a total predicate and is easily seen to be true. So this exercise aims at making you think about partial predicates and their meaning when we quantify over them. Let us see why (a) is true:

The statement $(\forall x)P(x)$ means that for any $x \in D$, either $P(x) = 1$ or $P(x) = 0$. So the negation says that there must exist some $x \in D$, where both $P(x) = 1$ and $P(x) = 0$. In other words, there exist $x \in D$ such that $\neg P(x) \downarrow$ and $\neg P(x)$ holds. But this last statement is precisely the meaning of $(\exists x)\neg P(x)$.

The same kind of reasoning applies to part (b).

**Comments:**

6. (0 Points) Do Exercise 7.6 and 7.7 in Lecture I.