CSCI 570 - Fall 2014 - HW5

Note: This homework assignment covers divide and conquer algorithms and recurrence relations. It is recommended that you read all of chapter 5 from Klienberg and Tardos, the Master Theorem from the lecture notes, and be familiar with the asymptotic notation from chapter 2 in Klienberg and Tardos.

1. The recurrence $T(n) = 7T(n/2) + n^2$ describes the running time of an algorithm ALG. A competing algorithm ALG$'$ has a running time of $T'(n) = aT'(n/4) + n^2 \log n$. What is the largest value of a such that ALG$'$ is asymptotically faster than ALG?

   Solution: We shall use Master Theorem to evaluate the running time of the algorithms.

   (a) For $T(n)$, setting $0 < \epsilon < \log_2 7 - 2 \approx 0.81$ we have $n^2 = O\left(n^{\log_2 7-\epsilon}\right)$. Hence $T(n) = \Theta\left(n^{\log_2 7}\right)$.

   (b) For $T'(n)$, if $\log_4 a > 2$ then setting $0 < \delta < \log_4 a - 2$ implies $n^2 \log n = O\left(n^{\log_4 a-\delta}\right)$ and hence $T'(n) = \Theta\left(n^{\log_4 a}\right)$.

   To have $T'(n)$ asymptotically faster than $T(n)$, we need $T'(n) = O(T(n))$, which implies $n^{\log_4 a} = O\left(n^{\log_2 7}\right)$ and therefore $\log_4 a \le \log_2 7 \implies a \le 49$. Since other expressions for run time of ALG$'$ require $\log_4 a \le 2 \implies a \le 16 < 49$, the largest possible value of a such that ALG$'$ is asymptotically faster than ALG is $a = 49$.

2. Solve the following recurrences by giving tight Θ-notation bounds in terms of n for sufficiently large n. Assume that $T(\cdot)$ represents the running time of an algorithm, *i.e.* $T(n)$ is positive and non-decreasing function of n and for small constants c independent of n, $T(c)$ is also a constant independent of n. Note that some of these recurrences might be a little challenging to think about at first.

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

   (b) $T(n) = 8T(n/6) + n \log n$

   (c) $T(n) = \sqrt{6006}T(n/2) + n^{\sqrt{6006}}$

   (d) $T(n) = 10T(n/2) + 2^n$

   (e) $T(n) = 2T(\sqrt{n}) + \log_2 n$

   (f) $T^2(n) = T(n/2)T(2n) - T(n)T(n/2)$

   (g) $T(n) = 2T(n/2) - \sqrt{n}$

Solution: In some cases, we shall need to invoke the Master Theorem with one generalization as described next. If the recurrence $T(n) = aT(n/b) + f(n)$ is satisfied with $f(n) = \Theta\Big(n^{\log_b a} \log^k n\Big)$ for some $k \geq 0$, then $T(n) = \Theta\Big(n^{\log_b a} \log^{k+1} n\Big)$.

(a) Observe that $f(n) = n^2 \log n$ and $n^{\log_b a} = n^{\log_2 4} = n^2$, so applying the generalized Master's theorem, $T(n) = \Theta\Big(n^2 \log^2 n\Big)$.

(b) Observe that $n^{\log_b a} = n^{\log_6 8}$ and $f(n) = n \log n = O\Big(n^{\log_6 8 - \epsilon}\Big)$ for any $0 < \epsilon < \log_6 8 - 1$. Thus, invoking Master's Theorem gives $T(n) = \Theta\Big(n^{\log_b a}\Big) = \Theta\Big(n^{\log_6 8}\Big)$.

(c) We have $n^{\log_b a} = n^{\log_2 \sqrt{6006}} = n^{0.5 \log_2 6006} = O\Big(n^{0.5 \log_2 8192}\Big) = O\Big(n^{13/2}\Big)$ and $f(n) = n^{\sqrt{6006}} = \Omega\Big(n^{\sqrt{4900}}\Big) = \Omega(n^{70}) = \Omega\Big(n^{13/2+\epsilon}\Big)$ for any $0 < \epsilon < 63.5$. Thus, from Master's Theorem $T(n) = \Theta(f(n)) = \Theta\Big(n^{\sqrt{6006}}\Big)$.

(d) We have $n^{\log_b a} = n^{\log_2 10}$ and $f(n) = 2^n = \Omega\Big(n^{\log_2 10 + \epsilon}\Big)$ for any $\epsilon > 0$. Therefore Master's Theorem implies $T(n) = \Theta(f(n)) = \Theta(2^n)$.

(e) Use the change of variables $n = 2^m$ to get $T(2^m) = 2T\Big(2^{m/2}\Big) + m$. Next, denoting $S(m) = T(2^m)$ implies that we have the recurrence $S(m) = 2S(m/2) + m$. Note that $S(\cdot)$ is a positive function due to the monotonicity of the increasing map $x \mapsto 2^x$ and the positivity of $T(\cdot)$. All conditions for applicability of Master's Theorem are satisfied and using the generalized version gives $S(m) = \Theta(m \log m)$ on observing that $f(m) = m$ and $m^{\log_b a} = m$. We express the solution in terms of $T(n)$ by

$$T(n) = T(2^m) = S(m) = \Theta(m \log m) = \Theta(\log_2 n \log \log_2 n),$$

for large enough n so that the growth expression above is positive.

(f) Owing to the unusual nature of this recurrence, we'll try to be a little descriptive in the solution since applicability of Master's Theorem is not foreseeable. Since $T(n)$ is positive for every $n > 0$, dividing the given recurrence by $T(n)T(n/2)$ gives

$$\frac{T(n)}{T(n/2)} = \frac{T(2n)}{T(n)} - 1.$$

Setting $S(n) = T(2n)/T(n)$ for $n \geq 2$, we get the recurrence $S(n/2) = S(n) - 1$ which is equivalent to $S(n) = S(n/2) + 1$. Positivity of $T(\cdot)$ implies positivity of $S(\cdot)$ and a step-by-step solution of the recurrence gives

$$S(n) = S(n/2) + 1 = S(n/4) + 2 = \cdots = S\Big(\frac{n}{2^{k-1}}\Big) + k - 1, \tag{1}$$

where $k = \log_2 n$ and

$$S\Big(\frac{n}{2^{k-1}}\Big) = S\Big(\frac{2n}{2^{\log_2 n}}\Big) = S(2) = \frac{T(4)}{T(2)} = c_0.$$

From the recurrence relation for $T(n)$, we have $T(4) = 2T(2)+2$, so that $c_0 = T(4)/T(2) = 2 + 2/T(2) > 2$. Furthermore, (1) amounts to $S(n) = \log_2 n + c_0 - 1$. Using this expression for $S(n)$, we have

$$\begin{aligned}
T(n) &= S\Big(\frac{n}{2}\Big)T\Big(\frac{n}{2}\Big) && \text{(2a)}\\
&= \Big(\log_2\frac{n}{2}+c_0-1\Big)T\Big(\frac{n}{2}\Big) && \text{(2b)}\\
&= \Big(\log_2\frac{n}{2}+c_0-1\Big)\Big(\log_2\frac{n}{4}+c_0-1\Big)T\Big(\frac{n}{4}\Big) && \text{(2c)}\\
&= \cdots = \Big(\log_2\frac{n}{2}+c_0-1\Big)\Big(\log_2\frac{n}{4}+c_0-1\Big)\dots\Big(\log_2\frac{n}{2^{k-1}}+c_0-1\Big)T\Big(\frac{n}{2^{k-1}}\Big) && \text{(2d)}\\
&= T(2)\prod_{l=1}^{k-1}\Big(\log_2\frac{n}{2^l}+c_0-1\Big) && \text{(2e)}\\
&= T(2)\prod_{l=1}^{k-1}(\log_2 n+c_0-l-1) && \text{(2f)}\\
&= T(2)\prod_{p=0}^{\log_2 n-2}(c_0+p) && \text{(2g)}\\
&= \Theta((\log_2 n+c_0-2)!) && \text{(2h)}
\end{aligned}$$

where (2b) uses the definition of $S(n)$, (2d) is the fully expanded expression for the recurrence on $T(n)$, (2g) employed the substitution $p=\log_2 n-l-1$, and (2h) assumed that $c_0+\log_2 n$ is an integer. If you want to be more precise, you can use the Γ function to write $T(n)=\Theta(\Gamma(\log_2 n+c_0-1))$ even when $c_0+\log_2 n$ is not an integer, but this is not mandatory.
Next we shall employ Stirling's approximation to dispense with the factorial in (2h). Recall that Stirling's approximation implies

$$m! = \Theta\Big(m^{m+0.5}e^{-m}\Big).$$

Setting $c_0-2=d$, we have

$$\begin{aligned}
\Theta((\log_2 n+c_0-2)!) &= \Theta\Big((d+\log_2 n)^{d+0.5+\log_2 n}e^{-d-\log_2 n}\Big) && \text{(3a)}\\
&= \Theta\Bigg(\Big(1+\frac{d}{\log_2 n}\Big)^{d+0.5+\log_2 n}(\log_2 n)^{d+0.5+\log_2 n}e^{-\log_2 n}\Bigg) && \text{(3b)}\\
&= \Theta\Big((\log_2 n)^{c_0-1.5+\log_2 n}n^{-\log_2 e}\Big) && \text{(3c)}
\end{aligned}$$

where we have used the constant upper and lower bounds (for $\log_2 n \geq 1$)

$$\begin{aligned}
\Big(1+\frac{d}{\log_2 n}\Big)^{d+0.5+\log_2 n} &= \exp\Big[(d+0.5+\log_2 n)\ln\Big(1+\frac{d}{\log_2 n}\Big)\Big]\\
&= \exp\Big[d\Big(\frac{d+0.5}{\log_2 n}+1\Big)\frac{\ln(1+d/\log_2 n)}{d/\log_2 n}\Big]\\
&\leq \exp\Big[d\Big(\frac{d+0.5}{1}+1\Big)\times 1\Big],
\end{aligned}$$

and

$$\left(1+\frac{d}{\log_2 n}\right)^{d+0.5+\log_2 n} \geq 1^{d+0.5+\log_2 n} = 1.$$

Our final answer is essentially (3c) which gives

$$T(n) = \Theta\Big(n^{-\log_2 e}(\log_2 n)^{c_0-1.5+\log_2 n}\Big).$$

(g) For this recurrence, we cannot apply Master's Theorem since pattern matching with the template recurrence relation $T(n) = aT(n/b) + f(n)$ gives the $f(n)$ term as negative. We will solve this from first principles. We have,

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) - \sqrt{n} && \text{(4a)}\\
&= 2\left(2T\left(\frac{n}{4}\right) - \sqrt{\frac{n}{2}}\right) - \sqrt{n} = 2^2T\left(\frac{n}{2^2}\right) - \sqrt{n}\big(1+\sqrt{2}\big) && \text{(4b)}\\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) - \sqrt{\frac{n}{2^2}}\right) - \sqrt{n}\big(1+\sqrt{2}\big) = 2^3T\left(\frac{n}{2^3}\right) - \sqrt{n}\big(1+\sqrt{2}+\sqrt{2^2}\big) && \text{(4c)}\\
&= \cdots = 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) - \sqrt{n}\big(1+\sqrt{2}+\sqrt{2^2}+\cdots+\sqrt{2^{k-2}}\big) && \text{(4d)}\\
&= 2^{k-1}T\left(\frac{n}{2^{k-1}}\right) - \sqrt{n}\left(\frac{\sqrt{2^{k-1}}-1}{\sqrt{2}-1}\right) && \text{(4e)}
\end{aligned}$$

where $k = \log_2 n$, (4d) represents the fully expanded form of the recurrence for $T(n)$ and (4e) was obtained using the formula for sum of a geometric progression. Further simplification of (4e) gives

$$\begin{aligned}
T(n) &= \frac{2^{\log_2 n}}{2}T\left(\frac{2n}{2^{\log_2 n}}\right) - \sqrt{n}\left(\frac{\sqrt{2^{\log_2 n}}-\sqrt{2}}{2-\sqrt{2}}\right) && \text{(5a)}\\
&= \frac{n}{2}T(2) - \sqrt{n}\left(\frac{\sqrt{n}-\sqrt{2}}{2-\sqrt{2}}\right) && \text{(5b)}\\
&= n\left(\frac{T(2)}{2} - \frac{1}{2-\sqrt{2}}\right) + \frac{\sqrt{2n}}{2-\sqrt{2}}, && \text{(5c)}
\end{aligned}$$

resulting in the following two cases.

Case 1: If $T(2) > \sqrt{2}/\big(\sqrt{2}-1\big)$, then the coefficient of n in (5c) is positive and therefore $T(n) = \Theta(n)$.

Case 2: If $T(2) = \sqrt{2}/\big(\sqrt{2}-1\big)$, then the coefficient of n in (5c) is zero and thus, $T(n) = \Theta(\sqrt{n})$.

Note that $T(2) < \sqrt{2}/\big(\sqrt{2}-1\big)$ is not possible, since that would imply that $T(n)$ is asymptotically negative which contradicts the premise of the question as $T(n)$ denotes running time of an algorithm.

3. Consider the following algorithm `StrangeSort` which sorts n distinct items in a list A.

(a) If $n \leq 1$, return A unchanged.

(b) For each item $x \in A$, scan A and count how many other items in A are less than x.

(c) Put the items with count less than $n/2$ in a list B.

(d) Put the other items in a list C.

(e) Recursively sort lists B and C using `StrangeSort`.

(f) Append the sorted list C to the sorted list B and return the result.

Formulate a recurrence relation for the running time $T(n)$ of `StrangeSort` on a input list of size n. Solve this recurrence to get the best possible $O(\cdot)$ bound on $T(n)$.

Solution: Suppose that A has n elements. Then steps (c), (d) and (f) can take up to $O(n)$ time, step (a) takes constant time and step (e) takes $2T(n/2)$ time. Step (b), when implemented in the way described above, takes $\Theta(n^2)$ time since each scan of A takes $\Theta(n)$ time and the scan is repeated for each of the n elements of A. Counting times taken for all the steps, we have

$$T(n) = 2T(n/2) + O(n) + \Theta\Big(n^2\Big) + O(1) = 2T(n/2) + \Theta\Big(n^2\Big)$$

Comparing the above recurrence with $T(n) = aT(n/b) + f(n)$ we have $f(n) = \Theta(n^2)$ and $n^{\log_b a} = n$ so that Master's Theorem gives $T(n) = \Theta(n^2)$.

Note: Although steps (b) and (c) together can be implemented more efficiently by sorting A (or finding the median of A) first, the question specifically asks for the running time of the given implementation without any further optimizations.

4. Solve Kleinberg and Tardos, Chapter 5, Exercise 1.

Solution: *Ideas and Notation:* Let us denote the combined set of elements as $A \bigcup B$. The core idea behind the solution to this problem is to recursively discard from consideration, an *equal* number of elements from the left and right of the median of $A \bigcup B$ (so that the median of the reduced set is the same as the median of the original set), until we are left with only two elements for which the median can be explicitly computed. Let A^s and B^s respectively represent the arrays A and B sorted in ascending order. For any $1 \leq i \leq n$, the design of the databases implies that $A^s[i]$ and $B^s[i]$ can be obtained by one query to their respective databases. Furthermore, we will work with the definition that the median for a set of m distinct elements is the $\lceil m/2 \rceil^{\text{th}}$ element when all the elements are sorted in ascending order.

Algorithm: We recursively define the procedure MEDIAN(l, a, b) below such that it returns the median of the set $A^s[a+1 : a+l] \bigcup B^s[b+1 : b+l]$. Then the answer we seek will be the return value of the function call MEDIAN$(n, 0, 0)$.

```
1: function MEDIAN($l$, $a$, $b$)
2:     if $l = 1$ then
3:         return $\min\{A^s[a+1], B^s[b+1]\}$
4:     end if
5:     $k \leftarrow \lceil l/2 \rceil$
6:     if $A^s[a+k] < B^s[b+k]$ then
7:         return MEDIAN($k, a + \lfloor l/2 \rfloor, b$)
8:     else
```

```
 9:         return MEDIAN($k, a, b + \lfloor l/2 \rfloor$)
10:     end if
11: end function
```

Query Complexity: Assuming that A and B had n elements each, let $T(n)$ denote the number of database queries made by the function call MEDIAN$(n, 0, 0)$. For $l = 1$, MEDIAN(l, a, b) executes line 3 using 2 database queries, regardless of values of a and b, implying $T(1) = 2$. For $l > 1$, line 6 requires 2 explicit database queries and exactly one of lines 7 or 9 is executed, amounting to $T(\lceil n/2 \rceil)$ implicit queries. Therefore, $T(n) = T(\lceil n/2 \rceil) + 2$. One can either assume n to be a power of 2 to get the simplified recurrence relation $T(n) = T(n/2) + 2$ and invoke Master's Theorem to arrive at $T(n) = \Theta(\log n)$, or solve the original recurrence from first principles (as below) to arrive at $T(n) \leq 2\lceil \log_2 n \rceil + 2$.

$$
\begin{aligned}
T(n) &= T\left(2^{\log_2 n}\right) \\
&\leq T\left(2^{\lceil \log_2 n \rceil}\right) \\
&= T\left(2^{\lceil \log_2 n \rceil - 1}\right) + 2 \\
&= T\left(2^{\lceil \log_2 n \rceil - 2}\right) + 4 \\
&= \cdots = T\left(2^{0}\right) + 2\lceil \log_2 n \rceil \\
&= 2\lceil \log_2 n \rceil + 2.
\end{aligned}
\tag{6}
$$

Proof of Correctness: We need to show that for any $l \in \{1, 2, \ldots, n\}$ and any valid indices $a, b \in \{0, 1, \ldots, n-1\}$, MEDIAN$(l, a, b)$ correctly finds the median of $A^s[a+1 : a+l] \bigcup B^s[b+1 : b+l]$. We switch to relative indexing for convenience, *i.e.* let $A^s_a[1 : l] = A^s[a+1 : a+l]$ and $B^s_b[1 : l] = B^s[b+1 : b+l]$. Also, let $(A^s_a \bigcup B^s_b)^s$ denote the array $A^s_a \bigcup B^s_b$ sorted in ascending order. For $l = 1$, line 3 correctly returns the median of 2 elements as per our working definition of the median. For $l > 1$, we have a total of $2l$ elements and utilizing the fact that all elements across A and B are distinct, we have two cases.

(a) Let $A^s_a[k] < B^s_b[k]$ so that line 7 is executed. Then the k elements in $A^s_a[1 : k]$ are all less than $B^s_b[k]$. Furthermore, the $k-1$ elements in $B^s_b[1 : k-1]$ are also less than $B^s_b[k]$ so that there are at least $2k = 2\lceil l/2 \rceil \geq l$ elements less than or equal to $B^s_b[k]$ in $A^s_a \bigcup B^s_b$. Since the median of $A^s_a \bigcup B^s_b$ is the l^{th} element when sorted in ascending order, it must be less than or equal to $B^s_b[k]$ implying that the elements in $B^s_b[k+1 : l]$ are strictly to the right of the median in $(A^s_a \bigcup B^s_b)^s$. Similarly, we note that the $l - k + 1$ elements in $B^s_b[k : l]$ are all greater than $A^s_a[k]$. Moreover, the k elements in $A^s_a[l-k+1 : l]$ are greater than or equal to $A^s_a[k]$ (since $l - k + 1 = l - \lceil l/2 \rceil + 1 = \lfloor l/2 \rfloor + 1 \geq \lceil l/2 \rceil = k$), so that there are at least $(l - k + 1) + k = l + 1$ elements greater than or equal to $A^s_a[k]$ in $A^s_a \bigcup B^s_b$. Since there are exactly $l + 1$ elements greater than or equal to the median in $A^s_a \bigcup B^s_b$, the elements in $A^s_a[1 : l-k]$ are strictly to the left of the median in $(A^s_a \bigcup B^s_b)^s$. Line 7 finds the median of $A^s_a[\lfloor l/2 \rfloor + 1 : \lfloor l/2 \rfloor + k] \bigcup B^s_b[1 : k]$. Observing that $\lfloor l/2 \rfloor = l - \lceil l/2 \rceil = l - k$, line 7 is finding the median of $A^s_a[l-k+1 : l] \bigcup B^s_b[1 : k]$ and thus is discarding exactly $l - k$ elements to the left as well as to the right of the original median MEDIAN(l, a, b) in the new function call MEDIAN$(k, a + \lfloor l/2 \rfloor, b)$. This is correct since discarding an equal number of elements from either side of the median does not change the median element.

(b) If $A_a^s[k] > B_b^s[k]$, then line 9 is executed. To prove correctness, the preceding argument can be repeated with the roles of A and B reversed. So we shall have the elements $A_a^s[k+1:l]$ strictly to the right of the median and the elements $B_b^s[1:l-k]$ strictly to the left of the median, and all of these elements are discarded in the line 9 function call $\text{MEDIAN}(k, a, b + \lfloor l/2 \rfloor)$ which finds the median of $A_a^s[1:k] \bigcup B_b^s[l-k+1:l]$.

5. Solve Kleinberg and Tardos, Chapter 5, Exercise 3.

Solution: Let us call a card to be a *majority card* if it belongs to a set of equivalent cards encompassing at least half of the total number of cards to be tested, *i.e.* for a total of n cards, if there is a set of more than $n/2$ cards that are all equivalent to each other, then any one of these equivalent cards constitutes a majority card. To keep the conquer step as simple as possible, we shall require the algorithm to return a majority card if one exists and return nothing if no majority card exists (this can be implemented by indexing the cards from 1 to n and a return value of 0 could correspond to returning nothing). Then the algorithm consists of the following steps, with S denoting the set of cards being tested by the algorithm and $|S|$ denoting its cardinality.

(a) If $|S| = 1$ then return the card.

(b) If $|S| = 2$ then test whether the cards are equivalent. If they are equivalent then return either card, else return nothing.

(c) If $|S| > 2$ then arbitrarily divide the set S into two disjoint subsets S_1 and S_2 such that $|S_1| = \lfloor |S|/2 \rfloor$ and $|S_2| = \lceil |S|/2 \rceil$.

(d) Call the algorithm recursively on the set of cards S_1.

(e) If a majority card is returned (call it card m_1), then test it for equivalence against all cards in $S \setminus \{m_1\}$. If m_1 is equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, then return card m_1.

(f) If m_1 is not equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, OR if no majority card was returned by the recursive call on S_1, then call the algorithm recursively on the set of cards S_2.

(g) If a majority card is returned (call it card m_2), then test it for equivalence against all cards in $S \setminus \{m_2\}$. If m_2 is equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, then return card m_2.

(h) If m_2 is not equivalent to at least $\lfloor |S|/2 \rfloor$ other cards in S, OR if no majority card was returned by the recursive call on S_2, then return nothing.

Complexity: Let $T(n)$ be the number of equivalence tests performed by the algorithm on a set of n cards. We have $T(2) = 1$ from step (b). Steps (d) and (f) can incur a total of $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$ equivalence tests whereas steps (e) and (g) could require up to $2(n-1)$ equivalence tests. Therefore, we have the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2n - 2$, which can be solved from first principles like in (6). Alternatively, the recurrence simplifies to $T(n) \leq 2T(n/2) + 2n - 2$ on assuming n to be a power of 2. The simplified recurrence can be solved using Master's Theorem to yield $T(n) = O(n \log n)$.

Proof of Correctness: We need to show that the algorithm returns the correct answer in both cases, *viz.* majority card present and majority card absent. If $|S| = 1$ then this single card forms a majority by our definition and is correctly returned by step (a). If $|S| = 2$ then either card is a majority card if both cards are equivalent and neither card is a majority card if they are not equivalent. Clearly, step (b) implements exactly this criterion. For steps (d)-(h) we consider what happens when a majority card is present or absent.

(a) *Majority card present:* If a majority card is present (say m), then at least one of the subsets S_1 or S_2 must have m as a majority card. It is simple to see this by contradiction since if neither S_1 nor S_2 have a majority card (or if m is not a mojority card in either subset) then necessarily the number of equivalent cards (or cards equivalent to m, including m) is upper bounded by $0.5\lfloor |S|/2 \rfloor + 0.5\lceil |S|/2 \rceil = 0.5|S|$ which immediately implies the absence of any majority card in S. Thus, at least one of m_1 (in step (e)) or m_2 (in step (g)) will be returned and at least one of these will be equivalent to m. Since steps (e) and (g) respectively test m_1 and m_2 against all other cards, the algorithm will necessarily find one of them to be equivalent to m and hence also as a majority card (since the set of majority cards is necessarily unique) and would return the same.

(b) *Majority card absent:* There are multiple possibilities in this scenario. If neither S_1 nor S_2 returns a majority card then clearly there is no majority card, and the algorithm terminates at step (h) returning nothing. If m_1 or m_2 or both are returned as valid majority cards then they would be respectively discarded as candidates in steps (e) and (g) when checked against all other cards in S as they are not truly in majority on the set S. Thus the algorithm correctly terminates in step (h) returning nothing.

Note: There are a lot of repeated equivalence tests in the algorithm described above, primarily because we are solving the problem in a top-down manner. Can you think of a bottom-up approach to solving this problem with potentially fewer equivalence tests? Hint: It is possible to achieve $O(n)$ complexity in terms of the number of equivalence tests required.

6. Solve Kleinberg and Tardos, Chapter 5, Exercise 6.

Solution: For simplicity, we shall identify a binary tree by its root node. Let us denote the values at the root node (level 1), the left child node (level 2) and the right child node (level 2) respectively by R, l and r. Consider the following algorithm with a complete binary tree as the input.

(a) If both left and right subtrees are empty, then return root node.

(b) If $R < l$ and $R < r$ then return root node.

(c) If $l < R < r$ then call the algorithm recursively on the left subtree rooted at the left child node.

(d) If $r < R < l$ then call the algorithm recursively on the right subtree rooted at the right child node.

Complexity: A complete binary tree with n nodes has a depth d such that $n = 2^d - 1$. In each function call, exactly one of the steps (a)-(d) will be executed. Moreover, step (a) or (b) is only executed in the last function call whereas every other function call executes either step (c) or step (d). It is clear that execution of either step (c) or step (d) has the effect of descending one level on the binary tree (either the left or the right subtree is the next function call argument), so that the algorithm has to terminate after at most $d = \log_2(n+1)$ recursive calls. Each function call involves at most 3 probes, *viz.* values at the root node, the left child node and the right child node, and therefore a total of at most $3\log_2(n+1) = O(\log n)$ probes are executed.

Proof of Correctness: It is clear that the algorithm terminates in either step (a) or step (b).

(i) If the algorithm terminates in step (b) then a node x with value R is returned such that $R < l$ and $R < r$ are satisfied, *i.e.* the returned node has smaller value than both of its children. Since the tree rooted at x was passed as input to this final function call, it had to have originated in either step (c) or step (d) of the penultimate recursive function call. If the origin was step (c) of the penultimate call then node x was the left child node with value less than its parent node. If the origin was step (d) of the penultimate call then node x was the right child node with value less than its parent node. In either case, node x has value less than its parent node. Since node x has value less than its parent node as well as its children nodes, it is a local minimum and hence a correct solution.

(ii) If the algorithm terminates in step (a) then a leaf node x is returned. We can repeat the foregoing argument to establish that node x has a value less than its parent node. Since node x has no children, it therefore is a local minimum and hence a correct solution.

7. Consider an array A of n numbers with the assurance that $n > 2$, $A[1] \geq A[2]$ and $A[n] \geq A[n-1]$. An index i is said to be a local minimum of the array A if it satisfies $1 < i < n$, $A[i-1] \geq A[i]$ and $A[i+1] \geq A[i]$.

   (a) Prove that there always exists a local minimum for A.

   (b) Design an algorithm to compute a local minimum of A. Your algorithm is allowed to make at most $O(\log n)$ pairwise comparisons between elements of A.

   Solution: We prove the existence of a local minimum by induction. For $n = 3$, we have $A[1] \geq A[2]$ and $A[3] \geq A[2]$ by the premise of the question and therefore $A[2]$ is a local minimum. Let $A[1:n]$ admit a local minimum for $n = k$. We shall show that $A[1:k+1]$ also admits a local minimum. If we assume that $A[2] \leq A[3]$ then $A[2]$ is a local minimum for $A[1:k+1]$ since $A[1] \geq A[2]$ by premise of the question. So let us assume $A[2] > A[3]$. In this case, the k length array $A[2:k+1] = A'[1:k]$ satisfies the induction hypothesis ($A'[1] = A[2] \geq A[3] = A'[2]$ and $A'[k-1] = A[k] \geq A[k+1] = A'[k]$) and hence admits a local minimum. This is also a local minimum for $A[1:k+1]$. Hence, proof by induction is complete.

   Consider the following algorithm with array A of length n as the input, and return value as a local minimum element.

   (a) If $n = 3$, return $A[2]$.

   (b) If $n > 3$,

      i. $k \leftarrow \lfloor n/2 \rfloor$.

      ii. If $A[k] \leq A[k-1]$ and $A[k] \leq A[k+1]$ then return $A[k]$.

      iii. If $A[k] > A[k-1]$ then call the algorithm recursively on $A[1:k]$, else call the algorithm recursively on $A[k:n]$.

Complexity: If the running time of the algorithm on an input of size n is $T(n)$, then it involves a constant number of comparisons and assignments and a recursive function call on either $A[1 : \lfloor n/2 \rfloor]$ (size $= \lfloor n/2 \rfloor$) or $A[\lfloor n/2 \rfloor : n]$ (size $= n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$). Therefore, $T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$. Assuming n to be a power of 2, this recurrence simplifies to $T(n) \leq T(n/2) + \Theta(1)$ and invoking Master's Theorem gives $T(n) = O(\log n)$.

Proof of Correctness: We employ induction. For $n = 3$ it is clear that step (a) returns a local minimum using the premise of the question that $A[1] \geq A[2]$ and $A[3] \geq A[2]$. Let us assume that the algorithm correctly finds a local minimum for all $n \leq m$ and consider an input of size $m + 1$. Then $k = \lfloor (m + 1)/2 \rfloor$. If step (b)(ii) returns, then a local minimum is found by definition and the algorithm gives the correct output. Otherwise, step (b)(iii) is executed since one of $A[k] > A[k - 1]$ or $A[k] > A[k + 1]$ must be true for step (b)(ii) to not return. If $A[k] > A[k - 1]$, then $A[1 : k]$ must admit a local minimum by the first part of the question and the given algorithm can find it correctly if $k \leq m$, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if $\lfloor (m + 1)/2 \rfloor \leq m$ which is true for all $m \geq 1$. Similarly, if $A[k] > A[k + 1]$, then $A[k : m + 1]$ must admit a local minimum by the first part of the question and the given algorithm can find it correctly if $m - k + 2 \leq m$, using the induction hypothesis on the correctness of the algorithm for inputs of size up to m. This holds if $k \geq 2$ or equivalently $\lfloor (m + 1)/2 \rfloor \geq 2$ which holds for all $m \geq 3$. Therefore, the algorithm gives the correct output for inputs of size $m + 1$ and the proof is complete by induction.

8. A polygon is called convex if all of its internal angles are less than 180°and none of the edges cross each other. We represent a convex polygon as an array V with n elements, where each element represents a vertex of the polygon in the form of a coordinate pair (x, y). We are told that $V[1]$ is the vertex with the least x coordinate and that the vertices $V[1], V[2], \ldots, V[n]$ are ordered counter-clockwise. Assuming that the x coordinates (and the y coordinates) of the vertices are all distinct, do the following.

   (a) Give a divide and conquer algorithm to find the vertex with the largest x coordinate in $O(\log n)$ time.

   (b) Give a divide and conquer algorithm to find the vertex with the largest y coordinate in $O(\log n)$ time.

   Solution: Since $V[1]$ is known to be the vertex with the minimum x-coordinate (leftmost point), moving counter-clockwise to complete a cycle must first increase the x-coordinates and then after reaching a maximum (rightmost point), should decrease the x-coordinate back to that of $V[1]$. To see this more formally, we claim that there cannot be two distinct local maxima in the x-axis projection of the counter-clockwise tour. For the sake of contradiction, assume otherwise. Then there exists a vertical line that would intersect the boundary of the polygon at more than two points. This is impossible by convexity of the polygon since the line segments between the intersection points must lie completely in the interior of the polygon, thus contradicting our assumption. Thus, $V_x[1 : n]$ is a unimodal array, and the first part of this question is synonymous with detecting the location of the maximum element in this array.

   Consider the following algorithm:

   (a) If $n = 1$, return $V_x[1]$.

(b) If $n = 2$, return $\max\{V_x[1], V_x[2]\}$.

(c) $k \leftarrow \lceil n/2 \rceil$.

(d) If $V_x[k] > V_x[k-1]$ and $V_x[k] > V_x[k+1]$, then return $V_x[k]$.

(e) If $V_x[k] < V_x[k-1]$ then call the algorithm recursively on $V_x[1 : k-1]$, else call the algorithm recursively on $V_x[k+1 : n]$.

Complexity: If $T(n)$ is the running time on an input of size n, then beside a constant number of comparisons, the algorithm is called recursively on at most one of $V_x[1 : k-1]$ (size $= k - 1 = \lceil n/2 \rceil - 1 \leq \lfloor n/2 \rfloor$) or $V_x[k+1 : n]$ (size $= n - k = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$). Therefore, $T(n) \leq T(\lfloor n/2 \rfloor) + \Theta(1)$. Assuming n to be a power of 2, this recurrence simplifies to $T(n) \leq T(n/2) + \Theta(1)$ and invoking Master's Theorem gives $T(n) = O(\log n)$.

Proof of Correctness: Proof is very similar to the last question and hence is omitted.

For the second part of this question, let p denote the index at which the x-coordinate was maximized. Observe that joining $V[1]$ and $V[p]$ by a straight line divides the polygon into an upper half and a lower half and the the vertex achieving the maximum y-coordinate must lie above this line. Once again invoking convexity of the polygon, it can be shown in a manner analogous to the first part of the question, that the counter-clockwise traversal $V_y[p] \rightarrow V_y[p+1] \rightarrow \cdots \rightarrow V_y[n] \rightarrow V_y[1]$ is unimodal and we need to detect the location of the maximum element of this array. So, considering the same algorithm as above with this new array $[V_y[p], V_y[p+1], \ldots, V_y[n], V_y[1]]$ will return the vertex with the maximum y-coordinate. The running time is still $O(\log n)$ as the algorithm is unchanged and the problem size is no bigger.

9. Given a sorted array of n integers that has been rotated an unknown number of times, give an $O(\log n)$ algorithm that finds an element in the array. An example of array rotation is as follows: original sorted array $A = [1, 3, 5, 7, 11]$, after first rotation $A' = [3, 5, 7, 11, 1]$, after second rotation $A'' = [5, 7, 11, 1, 3]$.

Solution: There are many different ways to solve this problem. Here, we shall consider reusing well known algorithms and algorithms developed in other questions on this homework as building blocks. Consider the following approach.

(a) Compute the rotation index p, *i.e.* find the minimum p such that $A[1 : p]$ and $A[p+1 : n]$ are both sorted arrays in ascending order (if there is no rotation, then return $p = 0$).

(b) If $p \neq 0$, run binary search on $A[1 : p]$ to find the element. Return *success* if found.

(c) If not found, then run binary search on $A[p+1 : n]$ to find the element. Return *success* if found and *failure* if not found.

Correctness of the algorithm is obvious. The two binary searches can be accomplished in $O(\log n)$ time each. Hence, if we can accomplish step (a) in $O(\log n)$ time then the problem is solved.

If there is no rotation, it can be confirmed by testing whether $A[1] < A[n]$ is true in constant time. In the presence of rotations it is necessarily true that $A[1] > A[n]$. If rotations are present then to find the rotation index p, we recall the algorithm in question 7, to find the local minimum in an array, and assume that all elements in A are distinct. Using the definitions in question 7, it is straightforward to see that $A[p+1]$ is the only local minimum if there exists a non-zero rotation p and thus finding the local minimum index is equivalent to finding the rotation index. In particular, we have $A[1] < A[2] < \cdots < A[p]$, $A[p{+}1] < A[p{+}2] < \cdots < A[n]$ and $A[p] > A[p+1]$. Thus, the rotation index p can be found in $O(\log n)$ time from the complexity analysis in question 7, and this completes our solution.