CS 660, HOMEWORK 1 (DUE: SUNDAY SEPTEMBER 15, 11:59 PM)

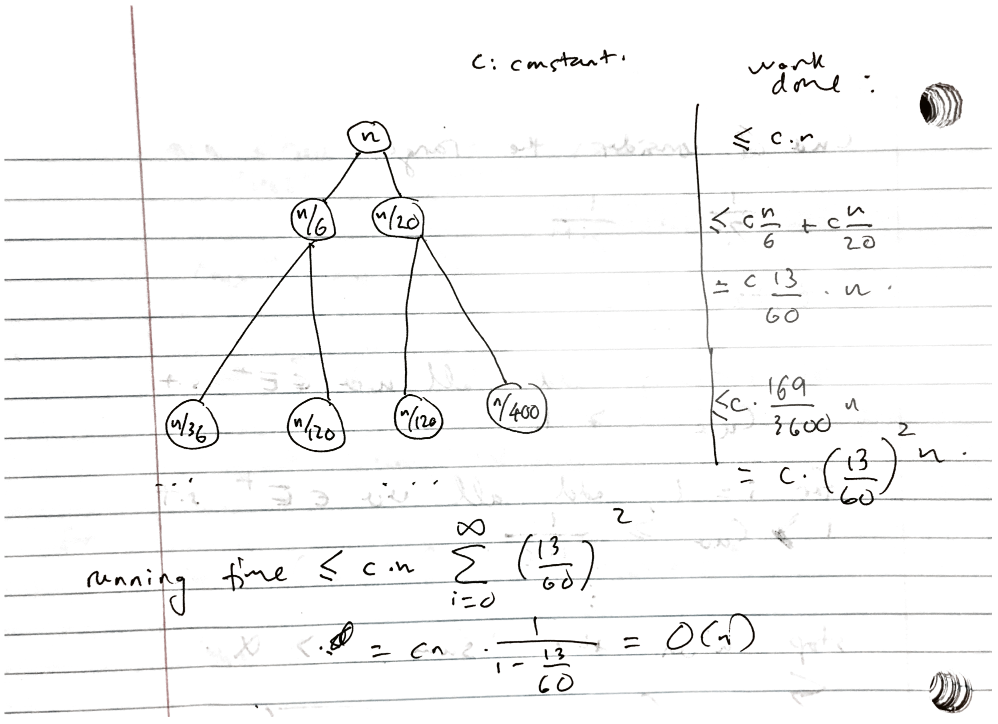
INSTRUCTOR: HOA VU

Each question is worth 20 points. The extra credit question is worth 10 points.

When you are asked to design an algorithm, do the following: a) describe the algorithm, b) explain (or more rigorously prove) why it is correct, and c) provide the running time.

Unless specified otherwise, the problems are from the textbook by Jeff Erickson that we use.

(1) **Question 1:**
- Solve $T(n) = T(n-1) + n^2$ using the recurrence tree method. In particular, what is big-O of $T(n)$. Hint: use the formula $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$. The work done at level $i = 0, 1, 2, \ldots, n$ is $(n-i)^2$. So the running time is at most $\sum_{i=0}^{n}(n-i)^2 = \sum_{i=0}^{n} i^2 = O(n^3)$.
- Solve $T(n) = 4T(n-1) + 99$ using the recurrence tree method. The work done at level $i = 0, 2, \ldots, n$ is 99^i. Hence, the running time is $\sum_{i=1}^{n} 4^i = O(4^n)$. Here we use the fact that $4^0 + 4^1 + \ldots + 4^n = O(4^n)$.
- Solve $T(n) = T(n/6) + T(n/20) + O(n)$ using the recurrence tree method.


c: constant.
work done:
n
n/6
n/20
n/36
n/120
n/120
n/400
$\le c \cdot n$
$\le c\frac{n}{6} + c\frac{n}{20} = c\frac{13}{60} \cdot n.$
$\le c \cdot \frac{169}{3600} n = c \cdot \left(\frac{13}{60}\right)^2 n.$

running time $\le c \cdot n \sum_{i=0}^{\infty} \left(\frac{13}{60}\right)^2 = cn \cdot \frac{1}{1 - \frac{13}{60}} = O(n)$

- Show that $2^{2n} \neq O(2^n)$. The "textbook" way to solve it is that suppose there exists constants c, n_0 such that $2^{2n} \le c2^n$ for all $n \ge n_0$. Note that $2^{2n} \le c2^n$ implies $c \ge 2^n$ which means c cannot be a constant since it has to grow larger as n grows. Thus, we have a contradiction.
  Another way to show it is that $\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} \frac{2^n}{1} = \infty$.
- Problem 6 (page 49): Solve B, E, H using master theorem.
  - $B = O(n)$ since $a < b^\alpha$ (where $a = 2, b = 4, \alpha = 1$).
  - $E = O(n \log n)$ since $a = b^\alpha$ (where $a = 3, b = 3, \alpha = 1$).
  - $H = O(n^{\log_2 4}) = O(n^2)$ since $a > b^\alpha$ (where $a = 4, b = 2, \alpha = 1$).

(2) **Question 2:** Problem 37 (page 64).

$MaxSubtree(v)$ returns the depth of the largest complete tree rooted at v. Note that $MaxSubtree(v)$ is equal to the minimum between $MaxSubtree(leftchild(v))$ and $MaxSubtree(rightchild(v))$ plus 1.

We use a table $Depth[1 \ldots n]$ where $Depth[v]$ stores the maximum depth of the tree rooted at v.

(a) If v is a leaf node that has no children (if the tree is represented by linked list then $leftchild(v) = rightchild(v) = NULL$), then return 1.
(b) $x, y \leftarrow 0$
(c) If v has a left child ($leftchild(v) \neq NULL$), then $x \leftarrow MaxSubtree(leftchild(v))$.
(d) If v has a right child ($rightchild(v) \neq NULL$), then $y \leftarrow MaxSubtree(rightchild(v))$.
(e) $Depth[v] = \min\{x, y\} + 1$.

(f) Return $\min\{x, y\} + 1$.

Start with the root r, call $MaxSubtree(r)$.

At the end, find v such that $Depth[v]$ is largest. Return v and $A[v]$. This takes $O(n)$ time. Another solution is to do the following: For $i = 1$ to n: $Depth[i] \leftarrow MaxTree(i)$. Then the running time is $O(n^2)$ which is also acceptable.

Running time: each node of the tree requires $O(1)$ running time before and after making the recursive call. Therefore, the running time is $O(n)$ if the tree has n nodes.

(3) **Question 3:** Problem 21, part a (page 55).

As discussed in class, assume n is power of 2 and define the median to be the one with rank n (i.e., nth smallest) in $A \cup B$. Note that $A \cup B$ has $2n$ elements. The median must be larger than exactly $n - 1$ elements and smaller than n other elements. The problem also assumes that all elements are distinct.

Base case: If $n = 1$, return the smaller between $A[1]$ and $B[1]$.

Let $C = A[1 \dots n/2]$ and $D = A[(n/2 + 1) \dots n]$. Let $E = B[1 \dots n/2]$ and $F = B[(n/2 + 1) \dots n]$. Consider $A[n/2]$ and $B[n/2]$.

Case 1: If $A[n/2] < B[n/2]$, then the median must be in $D \cup E$. The reason is as follows. Any element in C is smaller than $n/2$ elements in A and $n/2 + 1$ elements in B. Therefore, it is smaller than $n + 1$ elements and cannot be the median. Any element in F is larger than $n/2$ elements in B and $n/2$ elements in A. Hence, it is larger than n elements in total and cannot be the median. Hence, the median must be in $D \cup E$.

Next, we need to show that the median of $A \cup B$ is also the median of $D \cup E$. Let m be the median of $A \cup B$. Let e be the number of elements in E that are smaller than m and let a be the number of elements in A that are smaller than m. Note that since m is the median, $a + e = n - 1$. The rank of m in $D \cup E$ is

$$(a - n/2) + e = (a + e) - n/2 = n - 1 - n/2 = n/2 - 1.$$

Hence, m is the median in $D \cup E$.

Case 2: If $A[n/2] > B[n/2]$: is exactly symmetric to the first case. In this case, the median of $A \cup B$ must be the median in $C \cup F$.

Using the above, we have the following algorithm: $Median(A, B)$:

```
If $n = 1$, return $A[1]$.
If $A[n/2] < B[n/2]$, return $Median(D, E)$
Else, return $Median(C, F)$.
```

The running time is given by $T(n) = T(n/2) + O(1)$ which gives $O(\log n)$.

(4) **Question 4:** The majority element is the element that occurs more than half of the size of the array ($\lceil (n/2) \rceil$). Explain how to find the majority element in an array of n numbers in $O(n)$ time. If there is no majority element, the algorithm should also report "no majority element". Hint: there is a two line solution based on what we covered in class.

We can use the QuickSelect algorithm that finds the median in $O(n)$ time. Note that if there is a majority element, it must be the median. On the other hand, the median may not be a majority (in the case that the majority does not exist). Hence, after finding the median, we count the occurrences of the median, if it occurs more than $\lceil (n/2) \rceil$, return the median. Otherwise return "no majority element". The second step takes $O(n)$ time.

(5) **Question 5:** Problem 3a (page 124)

Let $T[i]$ be the maximum sum of a contiguous subarray ending at $A[i]$. There are only two cases, a) if the subarray does not start at $A[i]$, then the sum is equal to the maximum sum a contiguous subarray ending at $A[i-1]$ plus $A[i]$ or b) the sum starts and ends at $A[i]$.

$$T[i] = \max\{T[i-1] + A[i], A[i]\}.$$

Clearly, $T[1] = A[1]$. We can fill the table from left to right: For $i = 2$ to n: Fill $T[i]$. The running time is $O(n)$. Then return the maximum in T.

(6) **Extra credit:** Problem 21, part b (page 55) First, assume n, m are a power of 2.

We know that we can find median for the case $m = n$ using part (a).

If $k = n/2$, then we simply use the median algorithm (we can add the same number of ∞ and $-\infty$ to the smaller array).

If $n < n/2$, we can add $n/2 - k$ elements $-\infty$ to A. Then the problem becomes finding the median again. We can use part (a) to solve this.

If $n > n/2$, we add $k - n/2$ elements ∞ to A. Then the problem becomes finding the median again. We again use part (a) to solve this.

or Problem 3b (page 124) . If you do both, you will get 20 extra points. Let $P[i]$ be the maximum product of a contiguous subarray ending at $A[i]$. Let $N[i]$ be the maximum negative product of a contiguous subarray ending at $A[i]$.

If $P[1] = A[1]$ if $A[1] \geq 0$. Otherwise, $P[1] = NULL$. If $P[1] = A[1]$ if $A[1] < 0$. Otherwise, $P[1] = NULL$.

$$P[i] = \begin{cases} \begin{cases} P[i-1] \times A[i] & \text{If } A[i] \geq 0 \text{ and } P[i-1] \neq NULL \\ A[i] & \text{If } A[i] \geq 0 \text{ and } P[i-1] = NULL \end{cases} \\ N[i-1] \times A[i] & \text{If } A[i] < 0 \text{ and } P[i-1] \neq NULL \\ NULL & \text{otherwise .} \end{cases}$$

Similarly,

$$N[i] = \begin{cases} \begin{cases} P[i-1] \times A[i] & \text{If } A[i] < 0 \text{ and } P[i-1] \neq NULL \\ A[i] & \text{If } A[i] < 0 \text{ and } P[i-1] = NULL \end{cases} \\ \\ N[i-1] \times A[i] & \text{If } A[i] \geq 0 \text{ and } P[i-1] \neq NULL \\ NULL & \text{otherwise .} \end{cases}$$

In the end, return the maximum of P, if all entries of P are NULL, return 1.