Design and Analysis of Computer Algorithms
Assignment #1 Sketch of Solution
Due: March 24, 2007

1. Prove that
$$\sum_{i=1}^{n} \left\lceil \frac{i(i+1)}{2} \right\rceil r^i < \frac{1}{(1-r)^3} \text{ for all } n \geq 1 \text{ and } 0 < r < 1..$$

Answer:
Note that the sum may be written

$$\sum_{i=1}^{n} (1 + 2 + \cdots + i) r^i.$$

Then the term in the sum is

$$\begin{array}{lccccccc}
i = 1, & r^1 & & & & & & \\
i = 2, & r^2 & 2r^2 & & & & & \\
\vdots & \vdots & \vdots & & & & & \\
i = k, & r^k & 2r^k & \cdots & kr^k & & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & & & \\
i = n, & r^n & 2r^n & \cdots & kr^n & \cdots & nr^n &
\end{array}$$

If we sum the terms in the diagonal direction ($\searrow$), the sum beginning at r^1 is

$$r^1 + 2r^2 + 3r^3 + \cdots + nr^n.$$

According to the following inequality

$$\sum_{i=1}^{n} ir^i < \frac{r}{(1-r)^2}, \tag{1}$$

$$r^1 + 2r^2 + 3r^3 + \cdots + nr^n < \frac{r}{(1-r)^2}.$$

The sum of the next-highest diagonal is

$$r^2 + 2r^3 + 3r^4 + \cdots + (n-1)r^n.$$

Again, using Eq. (1), we can derive

$$r^2 + 2r^3 + 3r^4 + \cdots + (n-1)r^n < \frac{r^2}{(1-r)^2}.$$

In general, the ith diagonal is less than $\frac{r^i}{(1-r)^2}$. Therefore, the sum of all the terms is less than

$$\sum_{i=1}^{n} \frac{r^i}{(1-r)^2} = \frac{1}{(1-r)^2} \sum_{i=1}^{n} r^i < \frac{1}{(1-r)^2} \frac{1}{(1-r)} = \frac{1}{(1-r)^3}.$$

2. Let $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ be a polynomial in n of degree k with $a_k > 0$. Prove that $p(n)$ is in $\Theta(n^k)$.

   Answer:

$$\begin{aligned} f(n) &\leq \sum_{i=0}^{k} |a_k| n^i \\ &\leq n^k \sum_{i=0}^{k} |a_i| n^{i-k} \\ &\leq n^k \sum_{i=0}^{k} |a_i|, \text{ for } n \geq 1 \end{aligned}$$

   So, $f(n) = O(n^k)$. Please prove the other direction by yourself.

3. Prove or disprove:

$$\sum_{i=1}^{n} i^2 \in \Theta(n^2).$$

   Answer:

$$\sum_{i=1}^{n} i^2 = \Theta(n^3) \neq \Theta(n^2).$$

4. Find the asymptotic order of the following recurrence relations, where $T(1) = O(1)$, c is a constant, and the recurrence is for $n > 1$. Please verify your answer.

   a. $T(n) = 2T(n/2) + c$
   b. $T(n) = 4T(n/2) + cn$
   c. $T(n) = 4T(n/2) + cn^3$
   d. $T(n) = 4T(n/3) + cn$

   Answer:

   a. We guess $T(n) = O(n)$ and let $T(n) \leq an - c$, where a is a constant. Suppose $T(\frac{n}{2}) \leq \frac{an}{2} - c$. Then, consider

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + c \\ &\leq 2(\frac{an}{2} - c) + c \\ &= an - c \end{aligned}$$

   Therefore, $T(n) = O(n)$.

   b. We guess $T(n) = O(n^2)$ and let $T(n) \leq an^2 - cn$. Suppose $T(\frac{n}{2}) \leq a(\frac{n}{2})^2 - \frac{cn}{2}$. Then, consider

$$\begin{aligned} T(n) &= 4T(\frac{n}{2}) + cn \\ &\leq 4(\frac{an^2}{4} - \frac{cn}{2}) + cn \\ &= an^2 - cn \end{aligned}$$

Therefore, $T(n) = O(n^2)$.

c. We guess $T(n) = O(n^3)$ and let $T(n) \leq 2cn^3$. Suppose $T(\frac{n}{2}) \leq 2c(\frac{n}{2})^3 = \frac{cn^3}{4}$. Then, consider

$$\begin{aligned} T(n) &= 4T(\frac{n}{2}) + cn^3 \\ &\leq 4\frac{cn^3}{4} + cn^3 \\ &= 2cn^3 \end{aligned}$$

Besides, according to the recurrence relation, we can derive $T(n) = \Omega(n^3)$. Therefore, $T(n) = \Theta(n^3)$.

d. $T(n) = \Theta(n^{\log_3 4})$. Using substitution method is not easy to verify. For this problem, we can use recursion tree. Actually, this is the example we discuss in class.

5. Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is *unimodal*: For some index p between 1 and n, the values in the array entries increase up to the position p in A and then decrease the remainder of the way until position n. Actually, $A[p]$ is the "peak entry" of A. Show how to find the entry p by reading at most $O(\log n)$ entries of A. **Note:** You need to write down your idea, give the pseudocode of your algorithm, show the correctness, and analyze the time complexity of your algorithm.

Answer:

1^o *The idea*: If one needs to compute something using only $O(\log n)$ operations, a useful strategy is to perform a constant amount of work, throw away half the input, and continue recursively on what's left. This is like the binary search and can be viewed as a divide-and-conquer approach. According to this thought, suppose $T(n)$ is the running time on the input of size n, one can derive the following equation

$$T(n) \begin{cases} T(\frac{n}{2}) + c, & n > 2; \\ c, & n \leq 2. \end{cases} \tag{2}$$

If we want to set ourselves up to use Eq. (2), we could probe the midpoint of the array and try to determine whether the "peak entry" p lies before or after this midpoint.

2^o *Approach*: Based on the idea presented above, we can have the following approach. To determine the "peak entry" p, we recursively consider the middle entry of the considered array. Suppose we consider the array from left to right. If the middle entry is on the "up-slope", the we discard the left part and continue to consider the right part of the array. If the middle entry is on the "down-slope", the we discard the right part and continue to consider the left part of the array. The other case is that the middle entry is the peak entry. Hence, for a considered middle entry, say $A[\frac{n}{2}]$, there are three possibilities.

- If $A[\frac{n}{2}-1] < A[\frac{n}{2}] < A[\frac{n}{2}+1]$, $A[\frac{n}{2}]$ is before the peak entry and so we recursively continue on $A[\frac{n}{2}+1, \cdots, n]$.
- If $A[\frac{n}{2}-1] > A[\frac{n}{2}] > A[\frac{n}{2}+1]$, $A[\frac{n}{2}]$ is after the peak entry and so we recursively continue on $A[1, \cdots, \frac{n}{2}-1]$.

- Finally, if $A[\frac{n}{2}]$ is larger than both $A[\frac{n}{2}-1]$ and $A[\frac{n}{2}+1]$, the peak entry is $A[\frac{n}{2}]$ and we are done.

3^o *Correctness*: The correctness comes directly from the property of the input array. The entries in the discarded part of the array are definitely not the peak entry.

4^o *Time Analysis*: The time analysis is similar to the binary search. At each iteration, one only needs constant number of operations, c, to decide if a recursion continues by the three possibilities. Hence, the recurrence relation is as Eq. (2) and the running time is $\mathrm{O}(\log n)$.

6. Given a sequence of n distinct numbers $a_1, a_2, \ldots, a_n$, we define a *significant inversion* to be a pair $i < j$ such that $a_i > 2a_j$. Give an $\mathrm{O}(n \log n)$ algorithm to count the number of significant inversions in the given sequence of n distinct numbers $a_1, a_2, \ldots, a_n$.
Answer: This problem can be solved by revising the Merge Sort algorithm which uses the divide-and-conquer strategy. We first recursively partition the sequence of n elements into two subsequences having the same (or nearly equal) size until only one element is left in each subsequence. Then, the merging process takes place. We can use the merge process in the merge sort and record the number of significant inversions during the merge. The correctness comes from that the elements in the left subsequence have smaller indices than the elements on the right subsequence. We can add a judgement to see if $a_i > 2a_j$ during the merging process.