Assignment #2 CS4/531

Due Date: Monday, Oct 5. 2015

UNSUPPORTED SOLUTIONS RECEIVE NO CREDIT.

Total points: 46

- You MUST turn in your HW by 9:10am on Oct 5. After that, I will NOT accept your HW. This rule will be **STRICTLY ENFORCED.**
- Please PRINT YOUR LAST NAME, FIRST NAME and UB number on the first page.
- Write solution of each problem on a separate sheet. Staple them in the order of problem numbers.
- If your homework solution deviates significantly from these guidelines, TA may deduct up to 20% of the points.

Recall that: A basic arithmetic operation is one of the following: $x + y$, $x - y$, $x \times y$, x/y (which returns the integer part of x divided by y); and ($x \bmod y$) (which returns the remainder of x divided by y). But, x^y is NOT a basic arithmetic operation. Although most programming languages allow this operation, x^y is calculated by a subroutine, not by a CPU instruction.

1. (6 pts) Let a be a real number and N a positive integer. We want to compute a^N. (Recall that computing a^N is a basic operation for the RSA public key crypto-system). This, of course, can be easily done by $N - 1$ multiplications. However, this is NOT a polynomial time algorithm. (Let n is the number of bits needed to represent N. Then the value of N is $\Theta(2^n)$. The algorithm takes $\Theta(N) = \Theta(2^n)$ time.)

Describe an algorithm that computes a^N using $O(\log N)$ basic arithmetic operations.

2. (8 pts) In some applications (such as cryptography), we need to use very long integers. These integers cannot be stored in an *int* type variable. So we must use an array $A[1..n]$ to represent such long integers. (For simplicity, each element in A is used for 1 digit, and $A[n]$ is the most significant digit.) (For example, in the Crypto++ library widely used in cryptography applications, a special class is defined for such integers.) The basic arithmetic operations for such long integers can no longer be computed in constant time.

Let $A[1..n]$ and $B[1..n]$ be two n-digit integers. It is easy to see the sum of A and B can be computed in $\Theta(n)$ time. The multiplication procedure (you learned from elementary school) takes $\Theta(n^2)$ time.

Describe an algorithm for multiplication, with $O(n^{\log_2 3}) = O(n^{1.585})$ run time. (The algorithm is similar to Strassen's algorithm in nature. But much more simpler.)

3. (8 pts) Let $A[1..n, 1..n]$ be an $n \times n$ array, containing n^2 **distinct numbers**. Suppose that $A[1, 1]$ is located at the top-left corner. For each entry $A[i, j]$, its *up-neighbor, down-neighbor, left-neighbor, right-neighbor* are $A[i-1, j], A[i+1, j], A[i, j-1], A[i, j+1]$, respectively. (Note that if $A[i, j]$ is located at the border of $A[*, *]$, some of its neighbors may not exist. For example, $A[1, 1]$ has no up-neighbor, nor left-neighbor.)

An entry $A[i, j]$ is called a *local minimum* if its value is less than the values of all of its neighbors.

Describe an algorithm that finds a local minimum in $A[*,*]$ in $O(n)$ time. (Note that $A[*,*]$ may have more than one local minimums. You **only need to find one of them**).

Hint: The algorithm is **divide-and-conquer**. Since we want an $O(n)$ time algorithm, we can only look at $O(n)$ entries in $A[*,*]$.

Scan the middle row and the middle column of A. Thus we look at about $2n$ ($2n-1$ to be precise) elements in A. Let $x = A[i,j]$ be the minimum value among all entries we examined. (Here either $i = n/2$ or $j = n/2$). The mid row and the mid column of A divide A into four quadrants A_1, A_2, A_3, A_4. The idea of the algorithm is that, by looking at the neighbors of $A[i,j]$, we want to determine one quadrant A_i ($i = 1,2,3,4$) that must contain a local minimum element. Then in the next step, we repeat the process in that quadrant A_i.

4. (8 pts) Let $F(n)$ be a function defined on the integers $\{1, 2, \ldots, n\}$. $F(n)$ is said **bitonic** if its values first strictly increasing, then strictly decreasing. $F(k)$ is the maximum value of F if $F(k) > F(i)$ for any i ($1 \le i \le n$ and $i \ne k$). $F(i)$ can be computed by calling a procedure $\mathbf{F}(i)$.

We need to compute the maximum value $F(k)$. This can easily done by calling $\mathbf{F}(i)$ n times and keep the maximum value. However, the calling of $\mathbf{F}(i)$ takes a long time.

Describe an algorithm that finds the maximum value $F(k)$ and evaluates $F(i)$ at most $O(\log n)$ times.

5 (8 pts). Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points on 2D plane. We say p_1 is *dominated by* p_2 if $x_1 \le x_2$ and $y_1 \le y_2$. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points. The set of maximum points of P is defined to be:

$\max(P) = \{p_i \in P \mid p_i$ is **not** dominated by any other points in $P\}$.

For example, if $P = \{(1,1), (1,2), (1,3), (2,1)(3,2)\}$, then $\max(P) = \{(1,3), (3,2)\}$.

(a) Show that if the points in $\max(P)$ are sorted in increasing x-coordinate, then they are sorted in decreasing y-coordinate.

(b) Suppose that the points in P are given in increasing x-coordinate. Describe an $O(n)$ time algorithm for finding $\max(P)$. (To simplify the problem, you may assume no two points in P have the same x-coordinate.)

Note: There are several solutions. One of them is by using divide and conquer strategy, and similar to the algorithm for solving the closest pair of points problem.

6 (8 pts). Textbook, page 109, Problem 4-5.

Read and understand the problem. We assume "more than $n/2$ chips are good", which means $> \frac{n}{2}$ chips are good. (For example, if $n = 1000$, then at least 501 chips are good. If $n = 999$ then at least 500 chips are good.

Under this assumption, solve the following two problems.

(b) the part (b) in the book.

(c) Describe an algorithm that identify good chips using $\Theta(n)$ pairwise chip tests. (This is basically the part (c) in the book, but slightly easier.)