Problem 1 (20 points).

You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \cdot \log n)$.

Solution: We can use existing sorting algorithm (for example, merge-sort) to solve this problem. For the given array $S = (s_1, s_2, \cdots, s_n)$, we create a new array X that remembers the index of each element, $X = ((s_1, 1), (s_2, 2), \cdots, (s_k, k), \cdots, (s_n, n))$. In the new array X, each element is a pair: the first number of $X[k]$ is s_k and the second number of $X[k]$ is k, i.e., the index of s_k. Then we can use merge-sort to sort X. Notice that merge-sort is a comparison-based sorting algorithm, that is, we can (and we need to) define how to compare two elements in the array. In X, we use the first number of each pair to compare, i.e., we define $(s_i, i) < (s_j, j)$ if and only if $s_i < s_j$. Let X_1 be the sorted array of X. So now elements of X_1 are sorted by its first number of each pair, and identical first numbers must be adjacent to each other. Then we can traverse X_1 to remove all repeats, and we can do that in linear time. Let X_2 be the array that all repeats are removed. Now we need to transform back to original order. To do that we again use existing merge-sort algorithm, but now we need to use the second-number (i.e., the index) as the comparison function, i.e., now we define $(s_i, i) < (s_j, j)$ if and only if $i < j$. Let X_3 be the sorted list; we then extract the first numbr in each pair of X_3, which will be the final answer.

In this algorithm, we use two rounds of merge-sort, which takes $O(n \cdot \log n)$ in total. Other operations can be done in linear time. So overall this algorithms runs in $O(n \cdot \log n)$ time.

Problem 2 (20 points).

Assume you have functions f and g such that $f(n)$ is $O(g(n))$. For each of the following statements, decide whether you think it is true or false and give a proof or counterexample.

(a) $\log_2 f(n) = O(\log_2 g(n))$.

(b) $2^{f(n)} = O(2^{g(n)})$.

(c) $(f(n))^2 = O((g(n))^2)$.

Solution:

(a) This is false in general, since it could be that $g(n) = 1$, $f(n) = 2$, in which case $\log_2 g(n) = 0$, where we cannot write $\log_2 f(n) \leq c \cdot g(n)$. On the other hand, if we simply require $g(n) > 2$ for all n beyond n_1, the statement holds. Since $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$, we have $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (\log_2 c) \cdot (\log_2 g(n))$ once $n \geq \max\{n_0, n_1\}$.

(b) This is false. Let $f(n) = 2n$ and $g(n) = n$. Then $2^{f(n)} = 4^n$ and $2^{g(n)} = 2^n$, and $4^n \neq O(2^n)$.

(c) This is true. Assume we have an constant c such that $f(n) \leq c \cdot g(n)$. Then we can find constant c^2 such that $(f(n))^2 \leq c^2 \cdot (g(n))^2$ which implies $(f(n))^2 = O((g(n))^2)$.

Problem 3 (20 points).

For each pairs of functions below, indicate one of the three: $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$.

1. $f(n) = 100 \cdot n$; $g(n) = 0.01 \cdot n$
2. $f(n) = n^{1.01}$; $g(n) = n^{0.99}$
3. $f(n) = n$; $g(n) = n^{0.99} \cdot (\log n)^2$

4. $f(n) = 100 \cdot \log n;\ g(n) = n^{0.01}$
5. $f(n) = \log n^2;\ g(n) = \log n^3$
6. $f(n) = (\log n)^{\log n};\ g(n) = n$
7. $f(n) = 2^{\log n};\ g(n) = n^2$
8. $f(n) = 2^{n \cdot \log n};\ g(n) = 3^n$
9. $f(n) = n^n;\ g(n) = n!$
10. $f(n) = \log(n^n);\ g(n) = \log(n!)$

Solution:

1. $f = \Theta(g)$
2. $f = \Omega(g)$
3. $f = \Omega(g)$, compare $n^{0.01}$ and $\log n$
4. $f = O(g)$
5. $f = \Theta(g)$
6. $f = \Omega(g)$, let $n = e^a$
7. $f = O(g)$
8. $f = \Omega(g)$
9. $f = \Omega(g)$
10. $f = \Theta(g)$

Problem 4 (20 points).
Solve each of the following recursions.

1. $T(n) = 8 \cdot T(n/2) + 1000 \cdot n^2$
2. $T(n) = 2 \cdot T(n/2) + 10 \cdot n$
3. $T(n) = 2 \cdot T(n/2) + n^{0.5}$
4. $T(n) = 2 \cdot T(n/2) + n^{1.5}$
5. $T(n) = 4 \cdot T(n/2) + n \cdot \log(n)$

Solution

1. $a = 8,\ b = 2,\ \log_b a = 3$, Thus $T(n) = \Theta(n^3)$
2. $a = 2,\ b = 2,\ \log_b a = 1$, Thus $T(n) = \Theta(n \cdot \log(n))$

3. $a = 2,\ b = 2,\ \log_b a = 1$, Thus $T(n) = \Theta(n)$
4. $a = 2,\ b = 2,\ \log_b a = 1$, Thus $T(n) = \Theta(n^{1.5})$
5. $a = 4,\ b = 2,\ \log_b a = 2$, Thus $T(n) = \Theta(n^2)$

Problem 5 (20 points).

The Fibonacci sequence is a set of numbers that starts with a one or a zero, followed by a one, and proceeds based on the rule that each number is equal to the sum of the preceding two numbers.

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

Can you find a Divide and Conquer algorithm to find $F(n)$ such that the sub-problem is half the size? (You only need to give the recursive relationship and a brief explanation)

Solution:
Any recursive relationship like below will be accepted:

$$F(n) = F(\frac{n}{2}-1)\cdot F(\frac{n}{2}) + F(\frac{n}{2}+1)\cdot F(\frac{n}{2}) \tag{2}$$

$$F(2n) = F(n-1)\cdot F(n) + F(n+1)\cdot F(n) \tag{3}$$

$$F(2n) = 2\cdot F(n+1)\cdot F(n) - 2\cdot F(n)\cdot F(n) \tag{4}$$

$$F(2n+1) = F(n+1)^2 + F(n)^2 \tag{5}$$

Sample Explanation: (other explanation will also be accepted)

The Fibonacci sequence can be interpreted as a combination problem:

Let's say Amy is trying to climb the stairs, she can walk up either one or two stairs at a time. How many ways are there for her to climb N stairs? Let's think about the last step before Amy reaches the Nth stair. She might either be at the $N-1th$ stair and then walked up one stair, or at the $N-2th$ stair and walked up two stairs. So clearly $F(n) = F(n-1) + F(n-2)$, its basically Fibonacci sequence where $F(0) = 1, F(1) = 1, F(2) = 2$.

Now lets say there is an apple on $\frac{N}{2}th$ stair, if Amy stepped on it, she will pick it up. Obviously, when she reaches the $N-1th$ stair, she either have the apple or she doesn't. If she has the apple, she must first climb $\frac{N}{2}$ stairs and then climbed another $\frac{N}{2}-1$ stairs If she does not have the apple, she must first climb $\frac{N}{2}-1$ stairs, then walk up two, then climb another $\frac{N}{2}-2$ stairs

Thus $F(n) = F(\frac{n}{2}-1)\cdot F(\frac{n}{2}) + F(\frac{n}{2}+1)\cdot F(\frac{n}{2})$.

Problem 6 (15 points).

Prove that, at iteration k in the Graham-Scan algorithm, when we find $p_b \to p_a \to p_k$ is turning "left", where p_a and p_b are the top and 2nd top elements in stack S, respectively, then p_k together with all points in S form the convex hull of $\{p_1, p_2, \cdots, p_k\}$.

Solution: To prove that points in $S \cup \{p_k\}$ forms the convex hull of $\{p_1, p_2, \cdots, p_k\}$, we need to, by definition, verify that 1) $S \cup \{p_k\}$ form a *convex* polygon, 2) the polygon formed by $S \cup \{p_k\}$ contains all other points, and 3) such polygon is the smallest. To verify 1), we need to show that every 3 consecutive points in $S \cup \{p_k\}$ are turning left, and this is guaranteed by the algorithm. To verify 2), we notice that the algorithm guarantees that all the points that are popped out from the stack are *within* some triangle formed by points in $S \cup \{p_k\}$. To verify 3), we note that vertices of the polygon, i.e., $S \cup \{p_k\}$, are from $\{p_1, p_2, \cdots, p_k\}$ and therefore it must be the smallest.