Problem 1 (30 points). For each pairs of functions below, indicate one of the three: $f = O(g)$, $f = \Omega(g)$, or $f = \Theta(g)$.

1. $f(n) = 100 \cdot n$, $g(n) = n + 10^4$
2. $f(n) = n^{1.01}$, $g(n) = n^{0.99} \cdot \log n$
3. $f(n) = n$, $g(n) = n^{0.99} \cdot (\log n)^2$
4. $f(n) = 100 \cdot \log(100 \cdot n^3)$, $g(n) = (\log n)^2$
5. $f(n) = n^2 \cdot \log n^2$, $g(n) = n \cdot (\log n)^3$
6. $f(n) = n^{2.01} \cdot \log n^3$, $g(n) = n^2 \cdot (\log n)^2$
7. $f(n) = (\log n)^{\log n}$, $g(n) = n^2$
8. $f(n) = (\log n)^{\log n}$, $g(n) = n^{\log \log n}$
9. $f(n) = (n \cdot \log n)^{\log n}$, $g(n) = n^{(\log n)^2}$
10. $f(n) = n^2 \cdot 2^n$, $g(n) = 3^n$
11. $f(n) = 2^{n \cdot \log n}$, $g(n) = 3^n$
12. $f(n) = n!$, $g(n) = (\log n)^n$
13. $f(n) = n!$, $g(n) = n^n$
14. $f(n) = \sum_{k=1}^{n} (1/k)$, $g(n) = \log n$
15. $f(n) = \sum_{k=1}^{n} k^2$, $g(n) = n^2 \cdot \log n$

Solution:

1. $f = \Theta(g)$.
2. $f = \Omega(g)$. $f(n) = n^{0.99} \cdot n^{0.02}$, then use the fact that polynomial always dominates logarithmic.
3. $f = \Omega(g)$.
4. $f = O(g)$.
5. $f = \Omega(g)$.
6. $f = \Omega(g)$.
7. $f = \Omega(g)$. Take logarithmic on both: $\log f(n) = \log n \cdot \log \log n$, and $\log g(n) = 2 \cdot \log n$.
8. $f = \Theta(g)$. Note that $\log f(n) = \log n \cdot \log \log n = \log g(n)$.
9. $f = O(g)$.
10. $f = O(g)$.
11. $f = \Omega(g)$.
12. $f = \Omega(g)$. $\log n! = \sum_{k=1}^{n} \log k = \Theta(n \cdot \log n)$.
13. $f = O(g)$.

14. $f = \Theta(g)$.
15. $f = \Omega(g)$. $f(n) = \Theta(n^3)$.

Problem 2 (20 points). Solve each of the following recursions.

1. $T(n) = 9 \cdot T(n/2) + n^3$
2. $T(n) = 8 \cdot T(n/3) + n^2$
3. $T(n) = 2 \cdot T(n/4) + n^{0.5}$
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. $T(n) = O(n^{\log_2 9})$.
2. $T(n) = O(n^2)$.
3. $T(n) = O(n^{0.5} \cdot \log n)$.
4. $T(n) = O(n^{1.5})$.
5. $T(n) = O(n^2)$. We can solve this recursion in two ways.

   Approach 1: use the recursion tree. Similar to the proof of master's theorem, we build the recursion tree and compute the sum of the cost of all nodes in the tree. The recursion tree has $(\log n + 1)$ levels (the base of this logarithm is $b = 2$), and the k-th layer contains $a^k = 4^k$ nodes, and the cost for each node at the k-th layer is $n/b^k \cdot \log(n/b^k) = n/2^k \cdot \log(n/2^k)$. Therefore, the total cost is $T(n) = \sum_{k=0}^{\log n} 4^k \cdot n/2^k \cdot \log(n/2^k) = \sum_{k=0}^{\log n} 2^k \cdot n \cdot (\log n - k) = n \cdot \log n \cdot \sum_{k=0}^{\log n} 2^k - n \cdot \sum_{k=0}^{\log n} k \cdot 2^k$. Let $X = \sum_{k=0}^{\log n} 2^k$ and $Y = \sum_{k=0}^{\log n} k \cdot 2^k$. X is the sum of a geometric series: $X = (1 - 2^{1+\log n})/(1-2) = 2 \cdot n - 1$. We now compute Y. Write $2 \cdot Y = \sum_{k=0}^{\log n} k \cdot 2^{k+1} = \sum_{k=1}^{1+\log n} (k-1) \cdot 2^k = \sum_{k=1}^{\log n} (k-1) \cdot 2^k + \log n \cdot 2^{1+\log n} = \sum_{k=1}^{\log n} (k-1) \cdot 2^k + 2 \cdot n \cdot \log n$. Then we have $2 \cdot Y - Y = 2 \cdot n \cdot \log n - \sum_{k=1}^{\log n} 2^k = 2 \cdot n \cdot \log n - 2 \cdot (1 - 2^{\log n})/(1-2) = 2 \cdot n \cdot \log n - 2 \cdot n + 2$. We have $T(n) = n \cdot \log n \cdot (2 \cdot n - 1) - n \cdot (2 \cdot n \cdot \log n - 2 \cdot n + 2) = 2 \cdot n^2 - n \cdot \log n - 2 \cdot n = O(n^2)$.

   Approach 2: use lower and upper bounds. In order to use master's theorem, we can bound $n \log n$ with two polynomials, for example, $n \le n \cdot \log n \le n^{1.01}$. Let $T_1(n) = 4 \cdot T_1(n/2) + n$ and $T_2(n) = 4 \cdot T_2(n/2) + n^{1.01}$. We have $T_1(n) \le T(n) \le T_2(n)$. Now apply master's theorem to solve $T_1(n)$ and $T_2(n)$: $T_1(n) = O(n^2)$ and $T_2(n) = O(n^2)$. Therefore we have $T(n) = O(n^2)$.

Problem 3 (10 points). You are given an array with n integers in which some integers might appear more than once. Design an algorithm to remove all integers that appear more than once. For example, if you are given $(6,3,4,2,4,3,5,3)$, then your algorithm should return $(6,2,5)$. Your algorithm should run in $O(n \cdot \log n)$ time.

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 4 (20 points). You are given m sorted arrays, each of which contains n integers. You are asked to merge them into a single sorted array. You may assume that $m = 2^k$.

1. Consider the following recursive algorithm: use the linear-time algorithm (the "merge" step in the divide-and-conquer merge-sort algorithm) to merge the first 2 sorted arrays to obtain a single sorted array, then use the same algorithm (again the "merge" step in merge-sort) to merge this sorted array with the third array, and so on, until all arrays are processed. Analyze the running time of this algorithm.

2. Design a faster algorithm using divide-and-conquer technique, and give its running time.

Solution: First, recall the merge (X, Y) function, which takes two *sorted* lists X and Y as input and outputs the sorted list of all elements in X and Y, takes $O(|X| + |Y|)$ time.

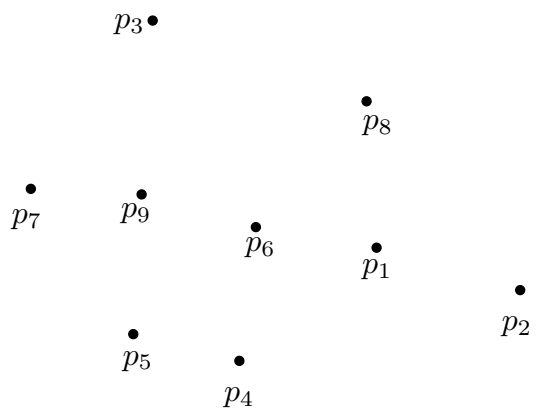
1. We need to consider how many merge are required, and for each calling of merge, what are the lengths of the two input arrays. The first calling of merge is to combine the first two given arrays, each of which has length of n. Therefore, the first calling of merge takes $2 \cdot n$ time. The second calling of merge takes this new array, which has length of $2 \cdot n$ and the third given array, which has length of n, as input. Therefore, the second calling of merge takes $2 \cdot n + n = 3 \cdot n$ time. The general case of the k-th calling of merge takes the output of the $(k-1)$-th calling of merge and the k-th given array, which have length of $k \cdot n$ and n respectively, as input, and therefore its running time is $k \cdot n + n = (k+1) \cdot n$. We in total requires calling merge $(m-1)$ times. So the total running time will be $\sum_{k=1}^{m-1} (k+1) \cdot n = O(m^2 \cdot n)$.

2. We design a divide-and-conquer algorithm for this problem. We define recursive function divide-and-conquer-merge-arrays $(A_1, A_2, \cdots A_m)$ takes m sorted arrays as input, each of which has length of n, and outputs the sorted list of them.

```
function divide-and-conquer-merge-arrays (A_1, A_2, ..., A_m)
    if m = 1, then return A_1;
    X_1 = divide-and-conquer-merge-arrays (A_1, A_2, ..., A_{m/2});
    X_2 = divide-and-conquer-merge-arrays (A_{m/2+1}, A_{m/2+2}, ..., A_m);
    X = merge (X_1, X_2);
    return X;
end function
```

   Notice that X_1 and X_2 each has length of $m \cdot n/2$. therefore the merge within the above algorithm takes $O(m \cdot n)$ time. Let $T(m,n)$ be the running time of divide-and-conquer-merge-arrays when its input contains m sorted arrays, each of which has length of n. We have $T(m,n) = 2 \cdot T(m/2, n) + O(m \cdot n)$. Solving this recursion gives $T(m,n) = O(m \cdot n \cdot \log m)$.

Problem 5 (10 points). Run the Graham-Scan algorithm on the following instance: draw the status of the stack as you process each point.



Solution: The anchor point with lowest y-coordinate will be p_4. Then we sort other points in counter-clockwise order, which will be $(p_4, p_2, p_1, p_8, p_6, p_3, p_9, p_7, p_5)$. Through running the Graham-Scan algorithm, the status of the stack will be (the left side shows the bottom of the stack, and the right side shows the top of the stack):
processing p_4: $[p_4$
processing p_2: $[p_4, p_2$
processing p_1: $[p_4, p_2, p_1$
processing p_8: $[p_4, p_2$
processing p_8: $[p_4, p_2, p_8$
processing p_6: $[p_4, p_2, p_8, p_6$
processing p_3: $[p_4, p_2, p_8$
processing p_3: $[p_4, p_2, p_8, p_3$
processing p_9: $[p_4, p_2, p_8, p_3, p_9$
processing p_7: $[p_4, p_2, p_8, p_3$
processing p_7: $[p_4, p_2, p_8, p_3, p_7$
processing p_5: $[p_4, p_2, p_8, p_3, p_7, p_5$

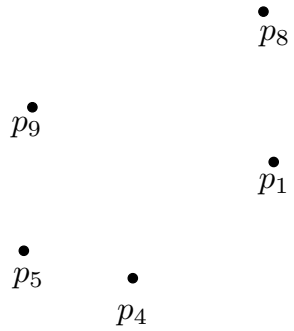
Problem 6 (10 points).

1. Design an instance of the convex hull problem such that the run of the Graham-Scan algorithm on your instance will not execute the *pop* operation. Your instance should contain 5 points on 2D plane and any three of them are not on the same line.

2. Design an instance of the convex hull problem such that the run of the Graham-Scan algorithm on your instance will execute the *pop* operations exactly twice. Your instance should contain 5 points on 2D plane and any three of them are not on the same line.

Solution: This problem uses the following fact: in the run of Graham-Scan algorithm, the points that are poped out from stack are those that are *not* on the convex hull. This fact can be easily obtained from the invariant that the points within the stack are exactly the points on the convex hull.

1. One instance can be (all points are on the convex hull):



2. One instance can be (two points are within the convex hull):