1. Given an array A of n integers and an integer X, find indices i and j, where $1 \leq i, j \leq n$, and $i \neq j$, such that, $X = A[i] + A[j]$. The $O(n^2)$ algorithm is obvious. Give an algorithm that does this in $O(nlgn)$. The algorithm returns (0, 0) if no (i, j) satisfy the summation condition.

2. Give an $O(n \lg n)$ time algorithm that determines whether two sets S_1 and S_2 of n numbers each, are disjoint or not. (Two sets are disjoint if their intersection is an empty set). This problem is very similar to the previous problem.

3. Given an array A of n distinct integers, write an algorithm that finds the minimum and the second-minimum elements of A. It is easy to do this in $2n$ steps: simply pass the array twice and find the numbers. Your algorithm should do this in $n + lgn$ steps.

4. You have an unsorted array A containing n non-negative integers in the range 1 … m, not necessarily unique. You want to ask: How many elements in A lie in the range between numbers a and b (including a and b). Assume that a and b are also elements of A themselves. You first need to do some preprocessing on this array. You are allowed to use extra storage. Now given any numbers a and b your algorithm should tell the number of elements between them in $O(1)$. What preprocessing will you do for this purpose? Your preprocessing should be in $O(n)$ time.

5. You have an array A of n elements, but only $O(lgn)$ elements in this array are unique, the rest are duplicates of these elements. Give an algorithm that can sort A in time$O(nlglgn)$.

6. You have an array A of n images. Being images you can compare them for equality, so you can use $A[i] == A[j]$, but there is no way to make comparisons like $A[i] \leq A[j]$ or $A[i] \geq A[j]$, because there is no sense of "order" in images. In short the array A **cannot** be sorted. Now, we say that the array has a *popular element* p, if p occurs more than $\frac{n}{2}$ times in the array A. Obviously it is also possible that A has no *popular element*. Write an algorithm that finds whether A has a popular element or not. If yes, it returns the popular element, otherwise it returns NULL. Write a divide and conquer algorithm which performed this task in $O(n)$.

7. Given an array A of n real numbers (both positive and negative possible), find the maximum sum of any contiguous subarray. Write an algorithm to accomplish this task in $O(nlgn)$. Note that the $O(n^2)$ algorithm is obvious. For example, if A is $\{1, 5, -2, 4, 5, -4, 2, -10\}$, the required subarray is $\{1, 5, -2, 4, 5\}$.

8. You have an array A of n integers which was first sorted and then rotated to the right k times. So if A was $\{1, 2, 3, 4, 5, 6, 7, 8\}$ and k was 3, then A became $\{6,7,8,1,2,3,4,5\}$. Given this rotated array (you do not know the value of k); write an $O(lgn)$ time algorithm to find the maximum element in A. You could try finding an $O(n)$ algorithm first.

9. You are given a sorted sequence S of n distinct integers. Give an $O(lgn)$ algorithm that finds an index i, if it exists, such that $S[i] = i$, and where $1 \leq i \leq n$ If no such index exists, the algorithm simply returns 0.

10. You are given two sorted arrays A and B, of sizes n and m respectively. A third array C is made of the union (in the sense of Set Union) of these two arrays. Give an algorithm that finds the k^{th} smallest element of C in $O(\lg n + \lg m)$.

11. Consider the summations: $X = x_1 + x_2 + \cdots + x_n$, $Y = y_1 + y_2 + \cdots + y_n$. Design an algorithm that finds indices i and j such that swapping x_i with y_j makes the summations X and Y equal, i.e. $X - x_i + y_j = Y - y_j + x_i$, if such x_i and $y_j$exist. The asymptotic running time of your algorithm should be faster than $O(n^2)$.

12. You have an unsorted array A containing n integers. Your goal is to search for an integer *key* in this array. One way is to go for the simple linear search, which will solve the problem in $O(n)$, the other is to sort the array first and then do a binary search, which will mean a running time of $O(nlgn)$. But you will do this using a randomized algorithm.

    a. Write an algorithm $RandomSearch(A, left, right, key)$ which uses a uniform random number generator $rand(left, right)$, to generate a random index r between *left* and *right* (including possibly *left* and *right*) and checks whether $A[r] = key$. If not, it repeats the process until all indices have been checked for the *key*, or until *key* is found. Note that on each call to $rand(left, right)$ the index is chosen from between *left* and *right* and so some values of r might repeat. What is the **expected** running time, $E[T(n)]$ of this algorithm. (Give an upper bound). Assume in this case that there is only one index r for which $A[r] = key$. In this case $E[T(n)]$ basically means the expected number of comparisons you must make before finding the right *key*.

    b. Now suppose that there are some $m \geq 1$ indices in A for which $A[r] = key$. Find $E[T(n, m)]$ for this situation. This should be a function of n and m.

    c. Now suppose that there are no indices r for which $A[r] = key$, (the *key* does not exist in A). What is $E[T(n)]$ in this case. Note that in this case $E[T(n)]$ is basically the expected number of comparisons you must make before all indices in A have been checked.

13. For the same problem as in 9; consider a randomized algorithm $PermutedSearch$. This algorithm works as follows:

i. Randomly permute the elements of array A.
ii. Run simple linear search on A to find *key*.

Given that some $m \geq 1$ elements in A contain *key*, what are the worst-case and average-case running times of $PermutedSearch$.

14. Prove that a geometric series $\sum_{i=0}^{n} b^i$ is bounded from above and below by a constant multiple of its largest term. Do this for the three cases where $b < 1$, $b > 1$ and $b = 1$.

15. Solve the following recurrences without using Master Theorem:

i. $T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$
ii. $T(n) = 5T\left(\frac{n}{5}\right) + O(\frac{n}{\lg n})$
iii. $T(n) = T(n-1) + O(\frac{1}{n})$
iv. $T(n) = \sqrt{n}T(\sqrt{n}) + O(n)$
v. $T(n) = T(n-1) + O(c^n)$, where $c > 1$, is a constant.

16. Many times during the analyses of algorithms we have done so far in the class, we have assumed that n is a power of some constant b, say 2, or 5 or any other positive constant. We have said that this assumption will not affect our analysis. In this problem you will prove the validity of this assertion.

i. Show that for any positive integer n and any base b, there must be some power of b lying in the range $[n, bn]$.
ii. Now that you have shown this fact, prove that when analyzing any algorithm for a problem for size n, taking the assumption that $n = m^b$ for some positive integer m and constant b, will not affect the asymptotic running time of the algorithm.
iii. Give an example of an analysis where it is handy to assume that n is a power of 2. We know that merge sort is one such example, where if n is an exact power of 2, we have a clean $O(nlgn)$ running time. Give another example, maybe from the algorithms you've done in class.

17. The greatest common divisor, $\gcd(a, b)$, of two positive integers a and b is the greatest integer that divides both a and b. One way of finding $\gcd(a, b)$ is the simple high-school method you all are familiar with. In this problem we see a method that employs divide and conquer.

i. Prove using induction, that:

$$\gcd(a,b) = \begin{cases} 2\gcd\left(\frac{a}{2},\frac{b}{2}\right) \text{ if } a,b \text{ are even} \\ \gcd\left(a,\frac{b}{2}\right) \text{if } a \text{ is odd}, b \text{ is even} \\ \gcd\left(\frac{a-b}{2},b\right) \text{ if } a,b \text{ are odd} \end{cases}$$

ii. Give a divide and conquer algorithm for finding the $\gcd(a,b)$ using the method given above.

iii. Given that both a and b are n-bit integers, what is the complexity of your algorithm (the upper bound) as a function of n. Note that the complexity of the high-school algorithm (called Euclid's algorithm for the greatest common divisor) is $O(n^3)$.

18. Using your knowledge of Min-Heap, from your data structures class, devise an algorithm that merges k sorted arrays, each of size n into a single sorted array A of size nk, in time $O(n \lg k)$. Does this mean you can improve the Merge step in MergeSort?

19. Suppose you have an unsorted array C of colors *red*, *green* and *blue*. You want to sort this array so that all *reds* are before all *greens*, followed by all *blues*. Only operations available to you for this purpose are: $C[i] == c$, where c is one of the three colors, and $Swap(C,i,j)$ which swaps the colors at indices i and j in C. You algorithm should run in $O(n)$.

20. For each of the following functions f find a simple function g such that $f(n) = \theta(g(n))$

i. $f(n) = (1000)2^n + 4^n$
ii. $f(n) = n + n\lg n + \sqrt{n}$
iii. $f(n) = \log n^{20} + (\lg n)^{10}$
iv. $f(n) = (0.99)^n + n^{100}$

21. You have an $n \times n$ matrix Z containing integer elements. Elements in each row of the array are in strict increasing order from left to right (means no two consecutive elements are the same). The elements in the columns are strictly in decreasing order from top to bottom. Design an $O(n)$ time algorithm that counts the number of zeros in Z.

22. The way we choose the pivot, in the simplest deterministic version of QuickSort, has an effect on the algorithm's running time on specific inputs. Keeping this in mind answer the following questions:

i. What is the worst-case running time of QuickSort if the middle element of the array is chosen as the pivot? What is the best-case running time in this case? What kind of data makes the best case?
ii. What is the worst-case running time of QuickSort if the left most element of the array is chosen as the pivot? What is the best-case running time in this case?

iii. Using a slightly different strategy, we chose the pivot to be the median of the leftmost, rightmost and the middle element of the array. What is the worst-case running time of QuickSort in this case? What is the best case running time?

iv. We know from our analysis of RandomizedQuickSort that every second pivot is "expected" to be good. This property ensures an average-case running time (the expected running time) of $O(n \lg n)$. What is the average case running time of RandomizedQuickSort, if we pick three elements at random from the array and choose their median as the pivot?

23. Find a tight asymptotic upper bound on each of the following summations:

i. $\sum_{k=1}^{n} \frac{1}{k^2}$

ii. $\sum_{k=0}^{\lg n} \frac{n}{2^k}$

iii. $\sum_{k=0}^{\infty} \frac{2^k}{k^2}$

iv. $\sum_{k=1}^{n} \frac{1}{k}$

v. $\sum_{k=1}^{n} k^r (\lg k)^s$ where $r \geq 0, s \geq 0$ are constants.

24. Consider distinct real numbers $x_1, x_2 \ldots, x_n$ and weights $w_1, w_2 \ldots, w_n$ where each $w_i > 0$; such that $\sum_{i=1}^{n} w_i x_i = 1.0$. You task is to design an algorithm that finds a weighted-median m, satisfying the condition: $\sum_{x_i<m} w_i < 0.5$ and $\sum_{x_i>m} w_i > 0.5$.

i. Give an algorithm that runs in worst-case $O(n \lg n)$

ii. Give an algorithm that runs in worst-case $O(n)$

25. Recall that while proving a lower bound of $\Omega(n \lg n)$ on the worst-case of comparison-based sorting we showed that $\lg(n!) = O(n \lg n)$. Now show that in fact, $\lg(n!) = \theta(n \lg n)$.

26. In the Selection algorithm we did in the class to find the i^{th} smallest element from an array of n elements in average-case time of $O(n)$, we made an assumption while writing pseudo-code, that all numbers are distinct. Modify the pseudo-code, such that is handles the case when numbers in the array are not unique.

27. Dry-run the QuickSort algorithm (where pivot is always the rightmost element) on the following array: $[44, 75, 23, 43, 55, 12, 64, 77, 33, 41, 19, 51, 62, 33, 72, 30]$

28. Dry-run the MergeSort algorithm on the array given below. You can make a tree to show all steps of Merge: $[44, 75, 23, 43, 55, 12, 64, 77, 33, 41, 19, 51, 62, 33, 72, 30]$

29. Dry-run the deterministic Select algorithm, the one with Median-Of-Medians, on the following array: $[44, 75, 23, 43, 55, 12, 64, 77, 33, 41, 19, 51, \ 33, 72, 30]$

30. Explain briefly why RandomizedQuickSort, with average-case running time of $O(n \lg n)$, is quicker in practice than MergeSort, Heapsort, and deterministic QuickSort, all with worst-case running times of $O(n \lg n)$. Give separate, and clear reasons for each kind of sort.