Problem 1 (10 points).

Consider recurrence relation $T(n) = \Theta(n) + T(a \cdot n) + T(b \cdot n)$, $T(1) = 1$, $0 < a < 1$, $0 < b < 1$. Prove the following:

1. $T(n) = \Theta(n)$, if $a + b < 1$.
2. $T(n) = \Theta(n \cdot \log n)$, if $a + b = 1$.

Solution. Assume that the term $\Theta(n)$ in the recursion admits a lower bound of $d_1 n$ and upper bound of $d_2 n$, for large enough n. That is $T(n) \geq d_1 n + T(an) + T(bn)$ and $T(n) \leq d_2 n + T(an) + T(bn)$.

1. We first prove that $T(n) = O(n)$, by induction. Assume that $T(n) \leq c_2 n$ for large enough n, where $c_2 = d_2/(1-a-b)$. Now we prove that $T(n+1) \leq c_2(n+1)$. We can write

$$T(n+1) \leq d_2(n+1) + T(a(n+1)) + T(b(n+1)).$$

We have $a(n+1) \leq n$ and $b(n+1) \leq n$ for large enough n as $a < 1$ and $b < 1$. By the inductive assumption we have

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + c_2 a(n+1) + c_2 b(n+1) \\ &\leq d_2(n+1) + c_2(a+b)(n+1) \\ &= (d_2 + c_2(a+b))(n+1) \\ &= c_2(n+1). \end{aligned}$$

It's easy to verify the last equation, i.e., $d_2 + c_2(a+b) = c_2$ with $c_2 = d_2/(1-a-b)$. In fact this is where we determine the value of c_2. You can also see why $a + b < 1$ is required here.

We then prove that $T(n) = \Omega(n)$, by induction. Assume that $T(n) \geq c_1 n$ for large enough n, where $c_1 = d_1/(1-a-b)$. Now we prove that $T(n+1) \geq c_1(n+1)$. We can write

$$\begin{aligned} T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\geq d_1(n+1) + c_1 a(n+1) + c_1 b(n+1) \\ &\geq d_1(n+1) + c_1(a+b)(n+1) \\ &= (d_1 + c_1(a+b))(n+1) \\ &= c_1(n+1). \end{aligned}$$

The last equation holds as $d_1 + c_1(a+b) = c_1$ with $c_1 = d_1/(1-a-b)$.

2. We first prove that $T(n) = O(n \log n)$, by induction. Assume that $T(n) \leq c_2 n \log n$ for large enough n, where $c_2 = -d_2/(a \log a + b \log b)$. Now we prove that $T(n+1) \leq c_2(n+1)\log(n+1)$. We can write

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\leq d_2(n+1) + c_2 a(n+1)\log(a(n+1)) + c_2 b(n+1)\log(b(n+1)) \\ &\leq d_2(n+1) + c_2(a\log a + b\log b)(n+1) + c_2(a+b)(n+1)\log(n+1) \\ &= (d_2 + c_2(a\log a + b\log b))(n+1) + c_2(n+1)\log(n+1) \\ &= c_2(n+1)\log(n+1). \end{aligned}$$

The last equation holds as $d_2 + c_2(a\log a + b\log b) = 0$ with $c_2 = -d_2/(a\log a + b\log b)$.

We then prove that $T(n) = \Omega(n \log n)$, by induction. Assume that $T(n) \geq c_1 n \log n$ for large enough n, where $c_1 = -d_1/(a \log a + b \log b)$. Now we prove that $T(n+1) \geq c_1(n+1)\log(n+1)$. We can write

$$\begin{aligned}
T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\
&\geq d_1(n+1) + c_1 a(n+1)\log(a(n+1)) + c_1 b(n+1)\log(b(n+1)) \\
&\geq d_1(n+1) + c_1(a\log a + b\log b)(n+1) + c_1(a+b)(n+1)\log(n+1) \\
&= (d_1 + c_1(a\log a + b\log b))(n+1) + c_1(n+1)\log(n+1) \\
&= c_1(n+1)\log(n+1).
\end{aligned}$$

The last equation holds as $d_1 + c_1(a\log a + b\log b) = 0$ with $c_1 = -d_1/(a\log a + b\log b)$.

Problem 2 (10 points).

Let S be an array with n distinct positive integers. We say two indices (i, j) form an inversion of S if we have $i < j$ and $S[i] > S[j]$. Design a divide-and-conquer algorithm to count the number of inversions in S. Your algorithm should run in $O(n \log n)$ time.

Solution. We can count the number of inversions using a divide-and-conquer algorithm similar to merge-sort. We define a recursive function (S', N) = count-inversions (S) with an array S as input and returns a pair (S', N) where S' represents the sorted array of S and N represents the number of inversions in S. To implement count-inversions, we will first divide the array S into two parts $S_1 = S[1 \cdots n/2]$ and $S_2 = S[n/2+1, \cdots, n]$, and then recursively call the count-inversions function on S_1 and S_2, which will get us the sorted subarray and number of inversions of S_1 and S_2. Notice that, the total number of inversions in S, is equal to the number of inversions in S_1, plus the number of inversions in S_2, and plus the number of inversions that span S_1 and S_2. The third item can be calculated in a way similar to merging two sorted arrays, where we merge them while counting the number of inversions. The running time of this algorithm can be analyzed in exactly the same way as we did for merge-sort, which is also $O(n \log n)$.

```
function count-inversions (S)
    if (|S| = 1): return (S, 0);
    (S_1, N_1) = count-inversions (S[1 ··· n/2]);
    (S_2, N_2) = count-inversions (S[n/2 + 1 ··· n]);
    (S_merged, N_3) = merge-counting-two-sorted-arrays (S_1, S_2);
    return (S_merged, N_1 + N_2 + N_3)
end function;
```

```
function merge-counting-two-sorted-arrays (S_1, S_2)
    let k_1 = 1, k_2 = 1, N = 0, S_merge = []
    While (k_1 ≤ |S_1| and k_2 ≤ |S_2|):
        if(S_1[k_1] < S_2[k_2]):
            S_merge[k_1 + k_2 − 1] = S_1[k_1]
            K_1 = K_1 + 1
        else:
            S_merge[k_1 + k_2 − 1] = S_2[k_2]
            K_2 = K_2 + 1
            N = N + |S_1| − k_1 + 1
    While (k_1 ≤ |S_1|):
        S_merge[k_1 + k_2 − 1] = S_1[k_1]
        K_1 = K_1 + 1
    While (k_2 ≤ |S_2|):
        S_merge[k_1 + k_2 − 1] = S_2[k_2]
        K_2 = K_2 + 1
    return (S_merged, N)
end function;
```

Problem 3 (10 points).

Let S be an array with n distinct positive integers. We say two indices (i, j) form a big-inversion of S if we have $2 \cdot i < j$ and $S[i] > 2 \cdot S[j]$. Design a divide-and-conquer algorithm to count the number of big-inversions in S. Your algorithm should run in $O(n \log n)$ time.

Solution 1. We can count the number of big-inversions by designing a divide-and-conquer algorithm as we did in problem 2. We divide the array into two parts: $S_1 = S[1 \cdots n/2]$ and $S_2 = S[n/2+1, \cdots, n]$. Because for a big-inversion (i, j), i and j can't both come from S_2. So the number of big-inversions in S is equal to the number of big-inversions in S_1 plus the number of big-inversions span S_1 and S_2. We define a recursive function count-big-inversions (S) that returns the number of big-inversions in S. We can recursively call this count-big-inversions function on S_1 to get the number of big-inversions in S_1. The difficult part comes from counting the big-inversions span S_1 and S_2. To do it, we create a new array: we double the indices for all the elements in S_1, double the value for all the elements in S_2, and then merge them into a new array by their indices. For example, assume $S = [10, 8, 6, 12, 3, 4, 2, 4]$, then

$$S_1 : \{S[1] = 10, S[2] = 8, S[3] = 6, S[4] = 12\}$$

$$S_2 : \{S[5] = 3, S[6] = 4, S[7] = 2, S[8] = 4\}$$

After doubling the indices of elements in S_1, and the values of elements in S_2, they become:

$$S_1 : \{S[2] = 10, S[4] = 8, S[6] = 6, S[8] = 12\}$$

$$S_2 : \{S[5] = 6, S[6] = 8, S[7] = 4, S[8] = 8\}$$

Then we can merge the two array by their indices:

$$A : \{S[2] = 10, S[4] = 8, S[5] = 6, S[6] = 8, S[6] = 6, S[7] = 4, S[8] = 8, S[8] = 12\}$$

$$A : \{10, 8, 6, 8, 6, 4, 8, 12\}$$

Now we relate the number of big-inversions of S that span S_1 and S_2 with the number of inversions (defined in Problem 2) in A. Keep in mind that A consists of elements originate from S_1 and elements originate from S_2. A pair (i, j) that i is from S_1 and j is from S_2 form an inversion of A if and only if (i, j) form a big-inversion of S that span S_1 and S_2. But not all the inversions of A are paired by two elements such that the first is from S_1 and the second is from S_2. To resolve this, we first count the inversions of A such that the second element is from S_2. Specifically, we slightly modify our count-inversions function used in Problem 2. We mark all the elements in A that come from S_2. In the count-inversions function, only the pairs with its second element being marked will be counted. We define this new inversions-counting algorithm as count-marked-inversions which takes $O(n \log n)$ time. Second, we count the number of inversions in A such that both elements are from S_2: the number of such inversions is exactly the number of inversions in S_2, which can be found by calling count-inversions (S_2). Combined, the number of big-inversions of S that span S_1 and S_2 can be calculated by: count-marked-inversions (A) - count-inversions (S_2).

The total number of big-inversions in A can be represented with the formular: count-big-inversions (S) = count-big-inversions (S_1) + count-marked-inversions (A) - count-inversions (S_2).

```
function count-big-inversions (S)
    if (|S| = 1): return (0);
    N1 = count-big-inversions (S[1 ··· n/2];)
    A = double-and-merge (S[1 ··· n/2], S[n/2 + 1 ··· n]);
    N2 = count-marked-inversions (A);
    N3 = count-inversions (S[n/2 + 1 ··· n]);
    return N1 + N2 − N3
end function;
```

Note that count-marked-inversions and count-inversions takes $O(n \log n)$ time. Let $T(n)$ be the running time of count-big-inversions:

$$\begin{aligned}
T(n) &= T(n/2) + O(n \log n) + O(n) + O(n \log n) \\
&= T(n/2) + O(n \log n) \\
&= T(n/4) + O(n \log n) + O(n/2 \log n/2) \\
&= O(n \log n) + O(n/2 \log n/2) + O(n/4 \log n/4) + \cdots + O(1 \log 1) \\
&\leq O(n \log n) + O(n/2 \log n) + O(n/4 \log n/4) + \cdots + O(1 \log n) \\
&\leq O(2n \log n) \\
&= O(2n \log n)
\end{aligned}$$

So $T(n) = O(n \log n)$.

Solution 2. We scan A from left to right. Let the current index be j. Whenever j is even, we push $A[j/2]$ into an *binary search tree*, where each node additionally stores the size of its left sub-tree. Additionally, at step j we search $A[j]$ in the binary search tree to get the size of its left sub-tree (rooted at $A[j]$), which is the number of pairs corresponding to $A[j]$, i.e., the number of is satisfying $2 \cdot i \leq j$ and $A[i] \geq 2 \cdot A[j]$. Adding them together gets the answer. Overall, we perform $n/2$ insertion operations and n search operations in a BST, which leads to $O(n \cdot \log n)$ time complexity.

Problem 4 (10 points).

You are given n points $P = \{p_1, p_2, \cdots, p_n\}$ on 2D plane, represented as their coordinates. You are informed that the convex hull of P contains $O(\log\log n)$ points in P. Design an algorithm to compute the convex hull of P in $O(n \cdot \log\log n)$ time. You may assume that no three points in P are on the same line.

Solution. We first find the point in P with smallest y-coordinate, denoted as p_1^*. Clearly, p_1^* must be on the convex hull of P. Let $C = (p_1^*)$ store the list of points on the convex hull of P found so far. Let $p_0^* = p_1^* - (1,0)$ be a virtual point (not within P) be the point on the left of p_1^*. We then iteratively find all other points on the convex hull of P. In the k-th iteration, $k = 2, 3, \cdots$, for each point $p \in P \setminus \{p_0^*, p_1^*\}$, we compute the angle $\angle p p_{k-1}^* p_{k-2}^*$. We then find the point that maximizes this angle, denoted as p_k^*. Clearly, p_k^* must be on the convex hull of p as well (*Proof.* line $p_k^* p_{k-1}^*$ is an indicator: all other points locate at one side of this line). Therefore, we add p_k^* to C and continue to next iteration. The algorithm terminates when we find p_k^* equals to p_1^*.

Each iteration finds a new point on the convex hull and takes linear time. The total number of iterations equals to the number of points on the convex hull. Hence, the running time is $O(n \cdot s)$, where s is the number of points on the convex hull. This algorithm is *output-sensitive*. When there are $O(\log\log n)$ number of points on the convex hull, as described in this problem, this algorithm runs in $O(n \cdot \log\log n)$ time.

Problem 5 (10 points).

If we add one more partition step in find-pivot function in selecting problem, and use the size 3 for each subarray, the algorithm would become:

```
function find-pivot (A, k)
    Partition A into n/3 subarrays;
    Let M be the list of medians of these n/3 subarrays;
    Partition M into n/9 subarrays;
    Let M' be the list of medians of these n/9 subarrays;
    return selection(M', |M'|/2)
end function;
```

Analyze the running time of the new selection function with the find-pivot function described above.

Solution. Let m be the median of array M'. Consider the numbers in M. In half of the $n/9$ subarrays, i.e., those with median that is less than m, there are two numbers (i.e., the median of this subarray and another number) that are guaranteed less than m. This amounts to $2 \cdot (n/9)/2 = n/9$ elements in M that are less than m. Now consider the numbers in A. Recall that each element in M is a median of a subarray of size 3 in A. As we already show that there are $n/9$ elements in M that are less than m, in the corresponding subarrays, there are 2 numbers (the median and another number) that are guaranteed less than m. This amounts to $2 \cdot n/9 = 2n/9$ elements in A that are less than or equal to m. Thus, there are at most $7n/9$ elements are larger than m in A. Symmetrically, there are at most $7n/9$ elements are smaller than m in A.

The recursive call of selection function in above algorithm takes $T(n/9)$ time as $|M'| = n/9$. The recurrence relation for the entire algorithm is: $T(n) \leq \Theta(n) + T(n/9) + T(7n/9)$, which gives $T(n) = \Theta(n)$.

Problem 6 (10 points).

Quicksort is another widely used sorting algorithm. Although its worst-case running time is $\Theta(n^2)$, its average running time is $O(n \log n)$. Quicksort partitions the list A using a random pivot x, like the randomized

algorithm for selection problem. Then, the algorithm recursively sorts elements smaller than x, then x, and then elements larger than x. The pseudocode of quicksort is as follows.

```
function Quicksort (A)
    if |A| ≤ 3 then
        Sort A by brute-force and output the sorted list of A
    else
        Choose a pivot x ∈ A uniformly at random
        for each element a_i of A
            Put a_i in A^- if a_i < x
            Put a_i in A^+ if a_i > x
        end for
        A_1^- = Quicksort(A^-)
        A_1^+ = Quicksort(A^+)
        Return (A_1^-, x, A_1^+)
    end if
end function
```

Show that the EXPECTED running time of the above Quicksort algorithm is $\Theta(n \log n)$

Solution. One solution is at `https://en.wikipedia.org/wiki/Quicksort#Using_recurrences`.

Problem 7 (10 points).

We learned that Graham-Scan algorithm can output the convex hull of a set of points P = $\{p_1, p_2, ..., p_n\}$ on 2D plane in $\Theta(n \cdot \log n)$ time. Prove that $\Theta(n \log n)$ is the best asymptotic running time we can expect, i.e., the OPTIMAL algorithm for the convex hull problem runs in $\Theta(n \cdot \log n)$ time.
Hint: We can prove this by showing that the sorting problem can be *reduced* to the convex hull problem in linear time. Since we know that the optimal algorithm for sorting takes $\Theta(n \log n)$ time, therefore the optimal algorithm for convex hull problem must take $O(n \log n)$ time. To achieve this, you will need to design an algorithm for sorting problem using the convex-hull algorithm. Specifically, given an array $S[1 \cdots n]$ with n distinct positive integers, you need to create an instance of the convex-hull problem (i.e., a set of points P, each of which is represented as coordinates on 2D plane) from S. You then use any algorithm (say, Grahan-Scan algorithm) to compute the convex-hull of P. You finally need to return the sorted list of S.

Complete the two procedures (*Procedure 1* and *Procedure 2*) within the following algorithm. Both *Procedure 1* and *Procedure 2* should run in linear time.

```
function sorting-using-convex-hull (S[1 ··· n])
    Procedure 1: create a set of 2D points P from S;
    C ← Graham-Scan (P); /* points in C will be sorted in counter-clockwise order */
    Procedure 2: compute the sorted list of S from C;
end function
```

Solution.

Without loss of generality, we assume that the given array $S[1 \cdots n]$ contains n distinct positive integers.

Procedure 1: Compute the maximum of S: $S_{max} = max(S[1] \cdots S[n])$; create a series of points $P = \{p_0, p_1, \cdots, p_n\}$ on a 2D plane, where $p_0 = (0,0)$ and $p_i = (\cos \frac{S[i] \cdot \pi}{S_{max}}, \sin \frac{S[i] \cdot \pi}{S_{max}})$, $1 \leq i \leq n$.

Procedure 2: It is guaranteed that C includes all $n+1$ points (see below). Rewrite C so that p_0 is its first point, then sequentially report the numbers corresponding to the points.

Clearly, both Procedure 1 and 2 run in linear time.

Correctness: For every point p_i in P ($i = 1 \cdots n$), we have $||p_0 p_i|| = 1$ and the angle θ_i between $p_0 p_i$ and the x-axis equal to $\frac{S[i] \cdot \pi}{S_{max}}$. Sorting $S[1 \cdots n]$ is equivalent to sorting θ_i ($i = 1 \cdots n$). Apparently all points in P are on the convex-hull of P, because all of the points are on a semi-circle with radius of 1 so it always "turn left" in Graham-Scan algorithm. Thus, any algorithm that computes the convex-hull of P and gives all the points in counter-clockwise essentially sorts all numbers in S.