Name 1: Abrar, Md Hasin **Access ID:** mqa6002
Name 2: Rahman Hera, Mahmudur **Access ID:** mbr5797
Name 3: Zahin, Tasfia **Access ID:** tkz5115

Problem 1 **Points:**

Let us assume that $\Theta(n) = cn$ for some constant $c > 0$, and $n = \frac{1}{b^k}$ where $k \in \mathcal{N} = \{1,2,3,...\}$. These assumptions simply make the analysis easier, while it can easily be extended to the general case.

We will prove the closed form of the given recurrence for $a < b$. The case where $a > b$ is analogous with switching the roles of a and b. We will prove the case where $a = b$ separately.

Case I: $a < b$**:** We expand each $T(.)$ as follows.

$$\begin{aligned}
T(n) &= cn + T(an) + T(bn) \\
&= cn + acn + bcn + T(a^2n) + 2T(abn) + T(b^2n) \\
&= cn + cn(a+b) + T(a^2n) + 2T(abn) + T(b^2n) \\
&= cn + cn(a+b) + a^2cn + b^2cn + 2abcn + T(a^3n) + 3T(a^2bn) + 3T(ab^2n) + T(b^3n) \\
&= cn + cn(a+b) + cn(a+b)^2 + T(a^3n) + 3T(a^2bn) + 3T(ab^2n) + T(b^3n) \\
&\;\;\vdots \\
&= cn + cn(a+b) + cn(a+b)^2 + ... + cn(a+b)^{k-1} + \sum_{i=0}^{k} \binom{k}{i} T(a^{k-i}b^in)
\end{aligned}$$

Let us look into the terms within the summation. We notice that when $i = k$, we have $T(b^kn) = T(1) = 1$. For any other term in the summation, we have $T(a^{k-i}b^in)$ where $i < k$. If we take ratio of the size of the subproblems, we get the following.

$$\frac{a^{k-i}b^in}{b^kn} = \frac{a^{k-i}}{b^{k-i}} = \left(\frac{a}{b}\right)^{k-i} < 1 \text{ when } i < k$$

Therefore, except for $i = k$, all other terms in the summation have $T(\text{something smaller than } 1)$, which is actually 0. These are here only because we expanded each of the $T(.)$, no matter what. We have expanded into smaller subproblems even when the size of the problem has been broken down to the base case. During actual execution, these will not exist. Therefore, we have:

$$\begin{aligned}
\sum_{i=0}^{k} \binom{k}{i} T(a^{k-i}b^in) &= \sum_{i=0}^{k-1} \binom{k}{i} T(a^{k-i}b^in) + T(b^kn) \\
&= \sum_{i=0}^{k-1} \binom{k}{i} \times 0 + 1 \\
&= 1
\end{aligned}$$

Plugging this back to the original recurrence, we have:

$$T(n) = cn + cn(a+b) + cn(a+b)^3 + ... + cn(a+b)^{k-1} + \Theta(1)$$

- **If** $a+b<1$, then the higher powers of $a+b$ will be dominated by the lower powers. Therefore, the most dominant term will be cn. Therefore, $T(n)=\Theta(n)$.
- **If** $a+b=1$, then we have: $T(n)=cn+cn+cn+...+cn=ckn$. However, $k=\log_{1/b} n=\Theta(\log n)$. Hence, $T(n)=\Theta(cn\log n)=\Theta(n\log n)$.

Case II: $a>b$**:** The analysis is exactly the same after we swap a and b.

Case III: $a=b$**:** This can be solved using classic Master's Theorem. We have: $T(n)=\Theta(n)+2T(\frac{n}{1/a})$ where $n>1$, and $T(n)=1$ when $n=1$.

- If $a+b<1$, then $a=b<1/2$. Thus, $1/a>2$. Therefore, $\log_{1/a}2<\log_2 2$. Because we have that $\log_{1/a}2<1$, the $\Theta(n)$ term dominates, and $T(n)=\Theta(n)$.
- If $a+b=1$, then $a=b=1/2$, and $\log_{1/a}2=1$. Therefore, by Master's Theorem, $T(n)=\Theta(n\log n)$.

Problem 2 **Points:**

To count the number of inversions in an array, at first we note that we can divide the array into two halves, and count the inversions in each of the halves recursively. Then we count the inversions across the two halves. Summing all three counts will give the total inversion count.

Algorithm 1: GetInversionCount

Intput:
$S[1..n]$: an array of n numbers
Output:
$S[1..n]$: an array of n sorted numbers in descending order
Count: the number of inversions in S

if $n \leq 1$ **then**
  return S, 0;
end
$S_1, Count_1 \leftarrow GetInversionCount\,(S[1,2,\ldots,\lfloor n/2 \rfloor])$;
$S_2, Count_2 \leftarrow GetInversionCount\,(S[\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2,\ldots,n])$;
$S, Count_3 \leftarrow CalculateInversionFromSortedArray(S_1, S_2)$;
$Count \leftarrow Count_1 + Count_2 + Count_3$;
return S, *Count*;

In the following algorithm, we calculate the inversion count between two arrays sorted in descending order given they were splitted from one single array. As a result, from the parameter list, the first array's elements will always have smaller indices than the second array's.

Algorithm 2: CalculateInversionFromSortedArray

Intput:
$A[1..m]$: a sorted array of m numbers in descending order
$B[1..n]$: a sorted array of n numbers in descending order
Output:
$C[1..m+n]$: an array of $m+n$ sorted numbers in descending order
Count: the number of inversions between A and B

```
C ← ϕ, k_A ← 1, k_B ← 1, Count ← 0;
A[m+1] ← −∞, B[n+1] ← −∞;
for k ← 1 to m+n do
    if A[k_A] > B[k_B] then
        C[k] ← A[k_A];
        k_A ← k_A + 1;
        Count ← Count + n − k_B + 1;
    end
    else
        C[k] ← B[k_B];
        k_B ← k_B + 1;
    end
end
return C, Count;
```

Correctness

At first we prove the correctness of *CalculateInversionFromSortedArray*, and then prove the correctness of *GetInversionCount*.

To prove the the correctness of *CalculateInversionFromSortedArray*, we show that the following holds.
Loop Invariant: For any k, at the end of the k^{th} iteration, C stores the k largest elements in A and B in descending order. k_A points to the largest element in $A-C$ and k_B points to the largest element in $B-C$. *Count* stores the total number of inversions corresponding to the k elements in C.

Base Case: At the beginning of the 1^{st} iteration, C is empty, *Count* is 0, k_A and k_B point to the first element in A and B respectively. As A and B are sorted in descending order, the first element corresponds to the largest element.
At the beginning of the iteration, if the element at k_A is larger than the element at k_B, the larger element is stored in C, and k_A is incremented by 1. As the elements are in descending order in both A and B, $A[k_A]$ being larger than $B[k_B]$ means that $A[k_A]$ is also larger than all other elements to the right of $B[k_B]$. So, the number of inversions corresponding to $A[k_A]$ can be found from $n-k_B+1$. We increase *Count* by this value. On the other hand, if the element at k_B is larger than the element at k_A, the larger element is stored in C, and k_B is incremented by 1. As the elements in B have higher indices than A in the original array (from where they were split), there is no inversion in this case. So, value of *Count* does not change, that is it remains 0.
So, after the end of the 1^{st} iteration, C stores the largest element in A and B. k_A and k_B points to the largest element in $A-C$ and $B-C$ respectively. Also, *Count* holds the inversion count corresponding to the element in C.

Induction Step: We assume the loop invariant is true for step m. Now, at step $m+1$, if $A[k_A]$ is larger than $B[k_B]$, we store the element corresponding to k_A in C, and increment k_A by 1. This implies C remains sorted in descending order, and k_A points to the largest element in $A-C$. Also, as B is in descending order, $A[k_A]$ is larger than the rest of the elements in B. So, we increase *Count* by the number of elements to the right of $B[k_B]$ including it which is found by $n-k_B+1$.
If $A[k_A]$ is smaller than $B[k_B]$, we store $B[k_B]$ in C, and increment k_B by 1. This also implies C remaining sorted in descending order, and k_A pointing to the largest element in $B-C$. Since the elements in B have higher indices than A in the original array, there is no inversion, and value of *Count* does not change.
So, at the end of the $m+1$ iteration, C stores the largest $m+1$ elements in A and B. k_A and k_B points to the largest element in $A-C$ and $B-C$ respectively. Also, *Count* holds the inversion count corresponding to the elements in C.
So, the loop invariant holds for all the iterations.

Now we prove the correctness of *GetInversionCount*.

Base Case: When $|S|=1$, *GetInversionCount* returns 0, which confirms the base case.

Induction Step: For $|S|>1$, S is divided into two parts, S_1 and S_2, on which further recursions are called. Since S_1 receives the first half and S_2 receives the second half of S, any index of S_1 will be smaller than any index of S_2. For the induction step, we can assume that the recursion calls on S_1 and S_2 return two sorted arrays with their inversion counts in $Count_1$ and $Count_2$ respectively. We only need to calculate the inversion counts across these two sorted arrays to get the total count. This is done using *CalculateInversionFromSortedArray* algorithm and stored in $Count_3$. So, the total number of inversions for the merged array of S_1 and S_2 would be the summation of $Count_1$, $Count_2$ and $Count_3$. Since we already proved *CalculateInversionFromSortedArray* algorithm to be correct, *Count* returned should contain the total number of inversions in S, proving this algorithm correct.

Runtime

Let $T'(m,n)$ be the runtime of the merging step *CalculateInversionFromSortedArray* where $|A| = m$ and $|A| = n$.

$$\begin{aligned} T'(m,n) &= 2(m+n)+3(m+n)+7 \\ &\leq \theta(m+n)+\theta(1) \\ &= \theta(m+n) \end{aligned}$$

Let $T(n)$ be the runtime of *GetInversionCount* where $|S|=n$.

$$\begin{aligned} T(n) &= 1+T(\frac{n}{2})+T(\frac{n}{2})+\theta(\frac{n}{2}+\frac{n}{2}) \\ &= 1+2T(\frac{n}{2})+\theta(n) \end{aligned}$$

So, the runtime of *GetInversionCount* can be written as:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \theta(n) & \text{if } n \geq 2 \end{cases}$$

We know from the Master's Theorem that if the running time of an algorithm takes the following general form,

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ aT(\frac{n}{b}) + \theta(n^d) & \text{if } n \geq 2 \end{cases}$$

the closed form of $T(n)$ would be:

$$T(n) = \begin{cases} \theta(n^d \log n) & \text{if } d = \log_b a \\ \theta(n^d) & \text{if } d > \log_b a \\ \theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Relating the running time of the *GetInversionCount* algorithm with the general form, we get $a = 2$, $b = 2$, and $d = 1$.

$\log_b a = \log_2 2 = 1 = d$

$\therefore\ T(n) = \theta(n^d \log n) = \theta(n^1 \log n) = \theta(n \log n)$

Problem 3 **Points:**

We assume a 1-based index. We notice that if n is even, then $S[n]$ can form a big-inversion with the numbers in $S[1..n/2-1]$. If n is odd, then $S[n]$ can form a big-inversion with the numbers in $S[1..\lceil n/2 \rceil - 1]$. Therefore, $S[n]$ needs to be compared with all of $S[1..\lceil n/2 \rceil - 1]$. If we can count the number of elements in $S[1..\lceil n/2 \rceil - 1]$ that are greater than $2S[n]$, then we have the number of big-inversions where $S[n]$ is involved.

Algorithm 3 separates out $S[n]$ and counts the number of big-inversions in $S[1..n-1]$ recursively. The recursive call returns a binary-search tree containing the elements in $S[1..\lceil n/2 \rceil - 1]$. Therefore, by querying $2A[n]$ in the tree, the algorithm counts the number of big-inversions where $S[n]$ is involved.

If n is even, it means that the next element $A[n+1]$ will need $A[n/2]$ in the search tree. Therefore, the algorithm inserts that in the BST before returning.

Algorithm 3: GetBigInversionCount

Intput:
$S[1..n]$: an array of n numbers
Output:
count: the number of big inversions in S
BST: a binary search tree of the elements in $S[1..\lfloor n/2 \rfloor]$, every node in this tree also stores the size of its right-subtree

if $n = 1$ **then**
  return 0, *NULL*;
end
small-BST, $C_1 \leftarrow$ ***GetBigInversionCount***($S[1..n-1]$);
$C_2 \leftarrow$ ***GetCountOfLargerNumbers***(*small-BST*, $2A[n]$);
if *n is even* **then**
  BST $\leftarrow$ ***insert***(*small-BST.root*, A[n/2]);
  rotate and balance the height of the tree in constant time;
else
  BST $\leftarrow$ *small-BST*;
end
return *BST*, $C_1 + C_2$;

This algorithm uses the following routines.

Algorithm 4: insert

Intput:
root: root of a binary search tree, each node stores size of right-subtree
x: new key to be stored in the tree
Output:
root: root of a new binary search-tree that stores *x*, and updates the size of right subtree in each node

```
if root = NULL then
    root ← create a new node with key=x;
    root.rightSubtreeSize = 0;
    return root;
end
if x ≤ root.key then
    root.left = insert(root.left, x);
else
    root.right = insert(root.right, x);
    root.rightSubtreeSize = root.rightSubtreeSize + 1;
end
```

The routine to count the number of nodes larger than a given value in a binary-search-tree is as follows:

Algorithm 5: GetCountOfLargerNumbers

Intput:
BST: a binary search tree, each node stores size of right-subtree
x: a number
Output:
count: number of keys in the BST that is larger than *x*

```
root ← BST.root;
if root = NULL then
    return 0;
end
count ← 0;
while root is not NULL do
    if root.key > x then
        count ← count + root.rightSubtreeSize + 1;
        root ← root.right;
    else
        root ← root.left;
    end
end
return count;
```

Correctness

We prove the correctness using induction. When there is only one element, there is no big-inversion. Therefore, the algorithm calculates the number of big-inversions when $n = 1$ correctly.

Let the algorithm calculate the number of big-inversions when $n = m$ correctly. This means that the algorithm will also construct a binary search tree with the elements in $A[1..\lfloor m/2 \rfloor]$.

Now, when $n = m+1$, the algorithm simply counts the number of keys $> 2A[m+1]$ in the search tree. We need to prove that it suffices to query the elements in $A[1..\lfloor m/2 \rfloor]$.

Remember that $A[n]$ needs to be compared with $A[1..\lceil n/2 \rceil - 1]$. Therefore, setting $n = m+1$, the last index that we need to compare is: $\lceil \frac{m+1}{2} \rceil - 1$. Considering both even and odd cases, we can show that $\lceil \frac{m+1}{2} \rceil - 1 = \lfloor m/2 \rfloor$. Therefore, the algorithm correctly compares $2A[m+1]$ with all elements where big-inversion is possible. Given that ***GetCountOfLargerNumbers*** correctly counts the number of keys greater than $2A[m+1]$, the algorithm correctly calculates the number of big-inversions where $A[m+1]$ is involved.

Finally, before returning, the algorithm inserts $A[\lfloor (m+1)/2 \rfloor]$ in the BST if $m+1$ is even. If $m+1$ is odd, $A[\lfloor (m+1)/2 \rfloor]$ is already in the tree.

Correctness of GetCountOfLargerNumbers and insert

The insert routine is standard insertion into binary search tree. The only thing we do is, we increment the size of the right subtree when we insert a subtree to the right. Doing so in every insert operation will maintain this count correctly. By using this count, we can calculate the number of elements greater than a given value x easily. If we see that x is smaller than a key, then it must also be smaller than everything to the right of that key.

Therefore, because GetCountOfLargerNumbers and insert both are correct, we can argue that the algorithm will generate a correct output for $m+1$. By induction, we argue that the algorithm is always correct.

Running time

If a binary-search tree has n elements, then ***GetCountOfLargerNumbers*** and ***insert*** both require $O(\text{tree-height})$ steps. This is because at every node, we either go right or left. Therefore, in worst case, we have to traverse the longest path to a leaf node. Because after every insert operation we balance the tree, the height of the tree is always $O(\log n)$. Therefore, ***GetCountOfLargerNumbers*** and ***insert*** both require $O(\log n)$ steps.

Except for these routines and the recursive call, the algorithm performs constant amount of work. Therefore, the running-time of Algorithm 3 is:

$$
\begin{aligned}
T(n) &= T(n-1) + O(\log n) \\
&= T(n-2) + O(\log(n-1)) + O(\log n) \\
&= T(n-3) + O(\log(n-2)) + O(\log(n-1)) + O(\log n) \\
&\;\;\vdots \\
&= T(1) + O(\log 2) + O(\log 3) + \ldots + O(\log(n-1)) + O(\log n) \\
&= O(n \log n)
\end{aligned}
$$

The last step follows from the fact that $T(1) = \Theta(1)$.

Problem 4 **Points:**

The point in P with the minimum y coordinate will surely fall in the convex hull of P. Let this point be P_{min}. The second point in the hull would be the point in P that makes the smallest angle with P_{min}. This process is repeated for the second point to get the third point and continued until we get back to the start. Points that do not fall on the perimeter of the hull form larger angles with the current point and are excluded by the algorithm.

Algorithm 6: convex-hull

Intput:
$P[P^1, P^2, ...P^n]$: an array of n points on a 2D plane
Output:
CH(P): Convex Hull of P

$CH \leftarrow [\,],\ i = 1$;
$P_{min} \leftarrow$ point in P with the minimum y coordinate;
$P^* \leftarrow P_{min}$;
$CH[0] \leftarrow P_{min}$;
while *(true)* **do**
  Find P_x that forms the smallest angle with respect to P^*;
  if $(P_x = P_{min})$ **then**
    break;
  end
  $CH[i] \leftarrow P_x$;
  $i \leftarrow i + 1$;
  $P^* \leftarrow P_x$;
end
return CH

Correctness

Loop Invariant: At the beginning of the i^{th} iteration, CH contains i points that fall on the convex hull of P.

Proof: We know that for any point P_i that falls in the convex hull of P, we can draw a line T_i through P_i such that all other points of P fall on one side of T_i.

When $i = 1$, CH contains only one point, P_{min}, with the smallest y coordinate. We can draw a line that goes through P_{min} such that all other points fall above it. So, P_{min} must be on the perimeter of the hull.

In the i^{th} iteration, the algorithm chooses a point P_x that forms the minimum angle with the line T_i. Since all other points are on one side of T_i, P_x must fall on the hull, and P_x must immediately follow P^* in the anti-clockwise ordering of the convex-hull. P_x is added to CH which now has $i + 1$ points that fall on the convex hull of P. Incrementing i to $i + 1$ restores the loop invariant.

Running Time

At every iteration, finding the point that forms the smallest angle with P^* takes $O(n)$ time. Since the convex hull of P contains $O(\log\log n)$ points in P, the while loop will run $O(\log\log n)$ times. Ignoring the steps that run in $O(1)$, this gives an overall complexity of $O(n\log\log n)$.

Problem 5

Points:

Let $T(n)$ be the running time of *slection*(A,k) where $n = |A|$. Then, we have:

$$T(n) = \text{time(find-pivot(A,k))} + \Theta(n) + T(\max(|A_1|,|A_2|))$$

where the partitioned arrays are A_1 and A_2.

Now, let us look at the steps in find-pivot. The first partition takes $\Theta(n)$ steps. Then, median calculation of the $n/3$ subarrays takes $\Theta(3\log 3 \times n) = \Theta(n)$ steps. The next partitioning takes $\Theta(n/3)$ steps. The calculation of medians takes $\Theta(3\log 3 \times n/3) = \Theta(n)$ steps. Overall, we have:

$$\text{time(find-pivot(A,k))} = \Theta(n) + T(n/9)$$

Therefore, the original recurrence is:

$$T(n) = T(n/9) + \Theta(n) + T(\max(|A_1|,|A_2|))$$

Now, let us analyze $|A_1|$. We notice that the pivot will be larger than at least half of the $n/9$ medians, which is $n/18$. Because the subarrays are of size 3, these $n/18$ medians are larger than $n/18$ numbers of the subarrays. Thus, the pivot is larger than at least $2n/18$ numbers. However, all these $2n/18$ numbers themselves are medians. This means that each of these medians are larger than another number in its subarray of size 3. Therefore, the pivot is guaranteed to be larger than at least $2n/18 + 2n/18 = 2n/9$ numbers. Thus, we have: $|A_1| \geq 2n/9$. A similar argument reveals that $|A_2| \geq 2n/9$.

Because $|A_1| = n - 1 - |A_2|$, we have:

$$|A_1| \leq n - 2n/9 = 7n/9$$

and

$$|A_2| \leq n - 2n/9 = 7n/9$$

Therefore, $\max(|A_1|,|A_2|) \leq 7n/9$. Therefore,

$$T(n) \leq \Theta(n) + T(n/9) + T(7n/9)$$

when $n > 1$. This form of the recurrence matches the form of Problem 1. In our case, we have $a = 1/9$ and $b = 7/9$. Since $a + b < 1$, using the recurrence solution in Problem 1, we have: $T(n) = \Theta(n)$.

Problem 6 **Points:**

We notice that the total number of times we may have to execute the algorithm (including the recursive calls) is $O(n)$. This is because the algorithm selects one pivot element in each run. After partitioning into A^- and A^+, this pivot element is placed where it should be, and it is never considered in future recursive calls.

In each call of the algorithm, we perform some constant work (choosing pivot, calling quicksort recursively, and returning). Overall, these works take $O(n)$ time (since we have an $O(n)$ numbers of calls to the algorithm).

In each call of the algorithm, we also perform some non-constant work. This work consists of checking the pivot with other elements, and placing those in A^- and A^+. More specifically, each $a_i \in A$ is checked for $< x$ and $> x$. Because a pivot will never show up again in future calls, we will determine the number of all such comparisons for all calls to the algorithm. Let X be that number. Then, the running time of the algorithm will be $O(n+X)$. The expected running time will be $O(n+E[X])$.

Now, Let $A'[1..n]$ be the sorted version of A. We define indicator random variable $X_{i,j}$ as follows:

$$X_{i,j} = \begin{cases} 1 & \text{if } A'[i] \text{ and } A'[j] \text{ are compared} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

We notice that:

$$E[X_{i,j}] = \Pr\left[A'[i] \text{ and } A'[j] \text{ are compared}\right] \tag{2}$$

Now, we determine this probability. Let $i < j$. Let us consider the following range of numbers in the sorted array: $A'[i], A'[i+1], A'[i+2], ..., A'[j]$.

We notice that if a pivot is chosen from this range which is neither $A'[i]$ nor $A'[j]$, then these two numbers are put in different partitions, and will never be compared in future. Only when the pivot in this range is either $A'[i]$ or $A'[j]$, we will compare these two numbers. There are total $j-i+1$ numbers in this range, each of which is equally likely to be chosen as a pivot. Therefore, the probability in the RHS of Equation 2 is $2/(j-i+1)$.

Now, we can calculate X as follows: in the partitioning steps of the algorithm, we perform two steps in each of the comparisons ($a_i < x$ is one comparison, $a_i > x$ is the other). Therefore, we need two steps of work whenever we compare two numbers. Therefore, X can be written as follows:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2X_{i,j} \tag{3}$$

Taking expectations, we have:

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 2X_{i,j}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 2E\left[X_{i,j}\right] \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 2\times\frac{2}{j-i+1} \\
&= 4\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{1}{j-i+1} \\
&= 4\sum_{i=1}^{n-1}\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n-i+1}\right) \\
&\leq 4\sum_{i=1}^{n-1}\left(1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n-i+1}\right) \\
&= 4\sum_{i=1}^{n-1}\sum_{k=1}^{k=n-i+1}\frac{1}{k} \\
&\leq 4\sum_{i=1}^{n-1}\int_{k=1}^{k=n-i+1}\frac{1}{k}\,dk \\
&= \sum_{i=1}^{n-1} O(\log(n-i+1)) \\
&= O(n\log n)
\end{aligned}
$$

Therefore, the expected running time of the algorithm is: $O(n+E[X]) = O(n+n\log n) = O(n\log n)$.

Problem 7 **Points:**

Without loss of generality, we can assume that the numbers in the array $S[1..n]$ are distinct. The cases where numbers may repeat can be handled with simple tweaks.

We will generate the 2D points using the following routine.

Algorithm 7: generate-2Dpoints-from-numbers

Intput:
$S[1..n]$: an array of n numbers
a: the smallest number of the original array S
b: the largest number of the original array S
Output:
$P[1..n]$, an array of n 2D points, ready to be fed to Convex-Hull algorithm

$P \leftarrow [\,]$;
$k_P \leftarrow 0$;
for $x \in S$ **do**
  $\theta \leftarrow \left(\frac{x-a}{b-a}\right)\pi/2$;
  $P[k_P] \leftarrow (cos\ \theta, sin\ \theta)$;
  $k_P \leftarrow k_P + 1$;
end
return P;

After getting the Convex-Hull, which is also an array of 2D points, we generate the numbers using the following routine.

Algorithm 8: generate-numbers-from-2Dpoints

Intput:
$P[1..n]$: an array of n 2D points, which form a Convex-Hull
a: the smallest number of the original array S
b: the largest number of the original array S
Output: $S'[1..n]$, a sorted array of n numbers

$S \leftarrow [\,]$;
$k_S \leftarrow 0$;
for $(x,y) \in P$ **do**
  $\theta \leftarrow$calculate-angle(x,y);
  $S[k_S] \leftarrow a + (b-a) \times \frac{\theta}{\pi/2}$;
  $k_S \leftarrow k_S + 1$;
end
return P;

Combining the two above routines, we have the following algorithm to sort the array $S[1..n]$.

Algorithm 9: sorting-using-convex-hull

Intput:
$S[1..n]$: an array of n numbers
Output:
$S'[1..n]$, a sorted array of the same n numbers in S

$a \leftarrow \min(S)$;
$b \leftarrow \max(S)$;
$P \leftarrow$ **generate-2Dpoints-from-numbers**(S, a, b);
$C \leftarrow$ **generate-convex-hull**(P);
$C \leftarrow$ reorder C so that $(1,0)$ is the first point;
$S' \leftarrow$ **generate-numbers-from-2Dpoints**(C, a, b);
return S';

We define the input and output of the ***generate-convex-hull*** algorithm as follows.

Algorithm 10: generate-convex-hull

Intput:
$P[1..n]$: an array of n distinct 2D points in cartesian space, no three points are linear
Output:
$C[1..n]$, the convex hull of P in anti-clockwise order

Next, we will first prove the correctness of the complete algorithm (Algorithm 9). Later, we will prove that the two routines (Algorithms 8 and 7) run in linear time. Finally, we will argue that the best possible running time of the algorithm ***generate-convex-hull*** (Algorithm 10) is $\Theta(n \log n)$.

Correctness

We will prove the correctness of Algorithm 9 by proving a number of claims.

Claim 1: Given an array $S[1..n]$ of distinct numbers, the mapping from a real number $x \in S$ to a 2D point $(\cos\theta, \sin\theta)$ is one-to-one and onto if $\theta = \frac{x - \min(S)}{\max(S) - \min(S)} \times \frac{\pi}{2}$.

Proof: The angle θ calculated by this mapping is in the range $[0, \frac{\pi}{2}]$, where a larger number is mapped to a larger θ. Therefore, the point $(\cos\theta, \sin\theta)$ will reside on the first quadrant of a unit circle with origin as its center. Because the numbers in S are distinct, the angles will be unique for each of the numbers, and hence, no two numbers will map to the same 2D point. Therefore, the mapping is one-to-one.

For all the real numbers in S, we get a θ and we will calculate a point $(\cos\theta, \sin\theta)$. Therefore, there exists a point in P for each $x \in S$. Hence, the mapping is onto.

Claim 2: Let $P[1..n]$ be an array of 2D points constructed from $S[1..n]$ using the mapping mentioned in Claim 1. Let $S[i]$ be mapped to $P[i] = (\cos\theta_i, \sin\theta_i)$. Then, $x_i = \min(S) + [\max(S) - \min(S)] \times \frac{\theta_i}{\pi/2}$ will calculate the original $S[i]$.

Proof: We have already proved that $S[i] \rightarrow P[i]$ mapping is onto and one-to-one. Therefore, the inverse mapping, which is $x_i = \min(S) + [\max(S) - \min(S)] \times \frac{\theta_i}{\pi/2}$ will calculate the original number $S[i]$.

Claim 3: Let $P[1..n]$ be an array of 2D points constructed from $S[1..n]$ using the mapping mentioned in

Claim 1. Then, all 2D points in P will be included in the convex hull of P.

Proof: Let P_1 and P_2 be two adjacent points on the quarter-circle. Then, all other points are on the same side of the line P_1P_2. Therefore, both P_1 and P_2 should be in the convex hull. In general, all two adjacent points on the circle will be in the convex hull. Therefore, the construction of P dictates that all points in P will be in its convex hull.

Claim 4: If $P[1..n]$ is an array of 2D points constructed from $S[1..n]$ using the mapping mentioned in Claim 1, then the convex hull of P will always contain the edge $(0,1) \rightarrow (1,0)$.

Proof: The largest number in S will be mapped to $\theta = \pi/2$, and hence, to the 2D point $(0,1)$. Similarly, the smallest number in S will be mapped to $(1,0)$. All other points in P will reside on the arc of the quarter-circle. Therefore, $(0,1) \rightarrow (1,0)$ will be in the convex-hull. The edge $(1,0) \rightarrow (0,1)$ does not work in the anti-clockwise direction.

Claim 5: If $P[i] \rightarrow P[j]$ is an edge on the hull in counter-clockwise direction and if the edge is not $(0,1) \rightarrow (1,0)$, then $x_i < x_j$ where x_i and x_j are the numbers calculated from $P[i]$ and $P[j]$ respectively.

Proof: Because $P[i] \rightarrow P[j]$ is an edge, and because we are strictly considering the anti-clockwise direction, $P[j]$ must come after $P[i]$. This effectively means that if $P[j] = (\cos\theta_j, \sin\theta_j)$ and $P[i] = (\cos\theta_i, \sin\theta_i)$, then $\theta_j > \theta_i$. Because the mapping from θ to x has a positive coefficient in θ, $x_j > x_i$.

Therefore, the numbers are mapped to different distinct 2D points in a way that all points are in the convex-hull, and that instead of just the one edge, we always have that the 2D point corresponding to the smaller number comes first. Therefore, by starting from $(1,0)$, by simply replacing the points with their corresponding numbers, we will have a sorted array of the original numbers.

Optimal running time of *generate-convex-hull*

In Algorithms 7 and 8, we perform constant amount of work for each of the points in $S[1..n]$ and $P[1..n]$, respectively. By Claim 4, we know that the problem size for both these routines is n. Therefore, Algorithms 7 and 8 both require $\Theta(n)$ time.

In Algorithm 9, we perform some constant work, and we have a step where we re-order the convex-hull so that $(1,0)$ comes first. This will take $\Theta(n)$ time as well, as we simply need to look for $(1,0)$ in the convex-hull. Therefore, the running-time of the complete Algorithm 9 is:

$$T(n) = O(1) + \Theta(n) + T(generateCH) + \Theta(n) + \Theta(n) = \Theta(n) + T(generateCH)$$

Because the optimal running time for sorting can be $\Theta(n \log n)$, we conclude that $T(generateCH)$ must at least be $\Theta(n \log n)$, which is the best asymptotic running time we can expect.

Handling duplicates

For simplicity of analysis, we assumed that no two numbers are the same in S. However, if duplicates appear, those can be handled with simple tweaks that require no more than $\Theta(n)$ work. Therefore, the analysis that *generate-convex-hull* will require at least $\Theta(n \log n)$ time is still correct.