CS 344 **Problem Set 1 Solutions**
Spring 2024 **Due:** February 16, 2024, 11:59 p.m

1. **Big-O notation.** We have learnt big-O notation to compare the growth rates of functions, this exercise helps you to better understand its definition and properties.

   (a) (10 points) Suppose n is the input size, we have the following commonly seen functions in complexity analysis: $f_1(n) = 1, f_2(n) = \log n, f_3(n) = n, f_4(n) = n \log n, f_5(n) = n^2, f_6(n) = 2^n, f_7(n) = n!$. Intuitively, the growth rate of the functions satisfy $1 < \log n < n < n \log n < n^2 < 2^n < n!$. Prove this is true. [**Hint**: You are expected to prove the following asymptotics by using the definition of big-O notation: $1 = O(\log n), \log n = O(n), n = O(n \log n), n \log n = O(n^2), n^2 = O(2^n), 2^n = O(n!)$. **Note**: Chap 3.2 of our textbook provides some math facts in case you need.]

   To prove the growth rate of the functions satisfy $1 < \log n < n < n \log n < n^2 < 2^n < n!$, we need to prove the following five asymptotics: $1 = O(\log n), \log n = O(n), n = O(n \log n), n \log n = O(n^2), n^2 = O(2^n), 2^n = O(n!)$. For each of the asymptotics (generally denoted as $f(n) = O(g(n))$ for illustrative purpose), we need to prove that there exist a constant c and an integer N, so that for all $n > N$, we have $f(n) < c \cdot g(n)$. The following are the proofs:

   - To prove $1 = O(\log n)$, we need to prove there exists a constant c and and an integer N, so that for all $n > N$, we have $1 < c \log n$, which requires $c > \frac{1}{\log n}$. We know when $n > 2$, $\log n > 1$, thus $\frac{1}{\log n} < 1$, so by choosing $c = 1$, we always have $1 < c \log n$ when $n > 2$, which proves $1 = O(\log n)$.
   - To prove $\log n = O(n)$, we need to prove there exist a constant c and and an integer N, so that for all $n > N$, we have $\log n < c \cdot n$, which requires $c > \frac{\log n}{n}$. We know when $n \geq 2$, $\frac{\log n}{n} \leq \frac{1}{2}$, so by choosing $c = 1$, we always have $\log n < c \cdot n$ when $n > 2$, which proves $\log n = O(n)$.
   - To prove $n = O(n \log n)$, we need to prove there exist a constant c and and an integer N, so that for all $n > N$, we have $n < cn \log n$, i.e., $1 < c \log n$, which requires $c > \frac{1}{\log n}$. We know when $n > 2$, $\frac{1}{\log n} < 1$, so by choosing $c = 1$, we always have $1 < c \log n$ when $n > 2$, which proves $n = O(n \log n)$.
   - To prove $n \log n = O(n^2)$, we need to prove there exist a constant c and and an integer N, so that for all $n > N$, we have $n \log n < c \cdot n^2$, i.e., $\log n < c \cdot n$, which requires $c > \frac{\log n}{n}$. We know when $n \geq 2$, $\frac{\log n}{n} \leq \frac{1}{2}$, so by choosing $c = 1$, we always have $\log n < c \cdot n$ when $n > 2$, which proves $n \log n = O(n^2)$.
   - To prove $n^2 = O(2^n)$, we need to prove there exist a constant c and and an integer N, so that for all $n > N$, we have $n^2 < c \cdot 2^n$, which requires $c > \frac{n^2}{2^n}$. We know when $n \geq 4$, $\frac{n^2}{2^n} \leq 1$, thus by choosing $c = 2$, we always have $n^2 < c \cdot 2^n$ when $n \geq 4$, which proves $n^2 = O(2^n)$.
   - To prove $2^n = O(n!)$, we need to prove there exist a constant c and and an integer N, so that for all $n > N$, we have $2^n < c \cdot n!$, which requires $c > \frac{2^n}{n!} = \frac{2 \cdot 2 \cdot 2 \cdot 2 \ldots \cdot 2}{1 \cdot 2 \cdot 3 \cdot 4 \ldots \cdot n}$.

We know when $n \geq 3$, $\frac{2^n}{n!} = \frac{2 \cdot 2 \cdot 2 \cdot 2 \ldots \cdot 2}{1 \cdot 2 \cdot 3 \cdot 4 \ldots \cdot n} < \frac{2 \cdot 2}{1 \cdot 2} = 2$ (because $\frac{2}{3} < 1, \frac{2}{4} < 1, \ldots, \frac{2}{n} < 1$), thus by choosing $c = 2$, we always have $2^n < c \cdot n!$ when $n \geq 3$, which proves $2^n = O(n!)$.

(b) (10 points) Let $f, g : N \to R^+$, prove that $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$. [**Hint**: The key is $\max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 \cdot \max\{f(n), g(n)\}$. **Note**: Proving this will help you to understand why we can leave out the insignificant parts in big-O notation and only keep the dominate part, e.g., $O(n^2 + n \log n + n) = O(n^2)$.]

To prove $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$, we need to prove that if a function $h(n)$ is upper-bounded by $f(n)+g(n)$, then it is also upper-bounded by $\max\{f(n), g(n)\}$, and vise versa.

- Proving $\Rightarrow$: Suppose $h(n) = O(f(n)+g(n))$, by definition, this means there exist a constant c and an integer N, such as $\forall n > N$, we have $h(n) < c(f(n) + g(n))$. Because $f(n)+g(n) \leq 2 \cdot \max\{f(n), g(n)\}$, we thus have $h(n) < c(f(n)+g(n)) \leq 2c \cdot \max\{f(n), g(n)\}$. This means that we can find a constant (i.e., $2c$) and an integer (still N), s.t., $\forall n > N$, we will have $h(n) < 2c \cdot \max\{f(n), g(n)\}$, which means that $h(n) = O(\max\{f(n), g(n)\})$.
- Proving $\Leftarrow$: Suppose $h(n) = O(\max\{f(n), g(n)\})$, by definition, this means that there exist a constant c and integer N, such that $\forall n > N$, we have $h(n) < c \cdot (\max\{f(n), g(n)\})$. Because $\max\{f(n), g(n)\} \leq f(n) + g(n)$, we thus have $h(n) < c \cdot (\max\{f(n), g(n)\}) \leq c \cdot (f(n)+g(n))$. This means that by selecting the same constant c and integer N, we will have $\forall n > N$, $h(n) < c \cdot (f(n) + g(n))$, which means $h(n) = O(f(n) + g(n))$.

Summarizing the above two directions, we have $O(f(n)+g(n)) = O(\max\{f(n), g(n)\})$.

2. **Proof of correctness.** (10 points) We have the following algorithm that sorts a list of integers to ascending order. Prove that this algorithm is correct. [**Hint:** You are expected to use mathematical induction to provide a rigorous proof.]

Algorithm 1: `Sort a list`

```
Input: Unsorted list A = [a_1, ..., a_n] of n items
Output: Sorted list A' = [a'_1, ..., a'_n] of n items in ascending order
for i = 0, i < n − 1, i++ do
    // Find the minimum element
    min_index = i
    for j = i + 1, j < n, j++ do
        if A[j] < A[min_index] then
            min_index = j
    // Swap the minimum element with the first element
    swap(A, i, min_index)
return A
```

(1) A comprehensive version of proof:

We use two loop invariants:

Outer loop: At the beginning of the ith iteration, elements 1 through $i-1$ are sorted in increasing order, and all of the elements i through $A.length$ are at least as large as all of the elements from 1 to $i-1$.

Inner loop: At the beginning of the jth iteration, minIndex is the index of the smallest element in the range $[i, j-1]$.

Outer loop:

Initialization: At the beginning of the first iteration, there are no elements in the range $[1, 0]$, so they are vacuously sorted in increasing order. The second clause of the loop invariant is also vacuously true.

Maintenance: Suppose the invariant holds before iteration i. We show that it holds before iteration $i+1$. If the inner loop invariant holds, then at the end of the inner loop, minIndex is the index of the smallest element in the array range $[i, A.length]$. Then the Swap command ensures that the smallest element in the range $[i, A.length]$ gets placed in $A[i]$. By the previous invocation of the loop invariant, we know that $A[1] \leq A[2] \leq \cdots \leq A[i-1]$, and $A[i]$ is at least as large as $A[i-1]$. Furthermore, from what we are arguing now, we know $A[i]$ is at least as small as all the elements in $A[i+1], \cdots, A[A.length]$. This maintains the invariant for the next iteration.

Termination: At the beginning of the $(A.length + 1)$th iteration, elements 1 through $A.length$ are sorted in increasing order, so the array is sorted.

Inner loop:

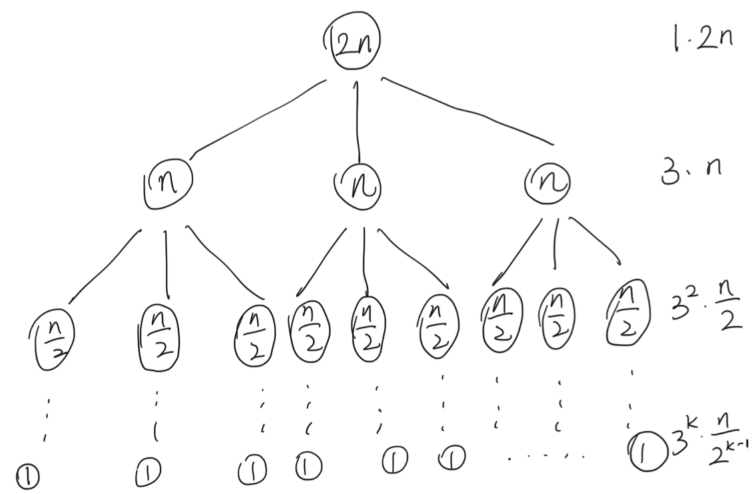
Initialization: At the beginning of iteration $i+1$, minIndex is i, which is the index of the smallest element in the range $[i, i]$.

Maintenance: Suppose the invariant holds prior to iteration j. We show that it holds prior to iteration $j+1$. At the beginning of the jth iteration, minIndex is the index of the smallest element in the range $[i, j-1]$. If $A[j] < A[minIndex]$, then j is the index of the smallest element in the range $[i, j]$, so setting $minIndex = j$ is correct. If $A[j]$ is not less than $A[minIndex]$, then minIndex is the index of the smallest element in the range $[i, j]$, so it is correct to leave minIndex alone. This maintains the invariant for the next iteration.

Termination: At the beginning of the $(A.length+1)$th iteration, minIndex is the index of the smallest element in the range $[i, A.length]$, which is the condition we needed for our proof of the outer loop invariant.

3. **Practice the recursion tree.** (10 points) We have already had a recurrence relation of an algorithm, which is $T(n) = 3T(n/2) + 2n$. Solve this recurrence relation, i.e. express it as $T(n) = O(f(n))$, by using the recursion tree method. [**Note**: If you find it difficult to draw a picture using Word or Latex directly, you can draw it on a piece of paper by hand and take a picture of it using your phone, then insert the picture into your Word or Latex submission.]

By this recursion tree, we know:



In Layer 0: 1 problem, each taking $2n$ work, total work is $1 \cdot 2n = 3^0 \cdot \frac{2n}{2^0}$.

In Layer 1: 3 problems, each taking n work, total work is $3 \cdot n = 3^1 \cdot \frac{2n}{2^1}$.

In Layer 2: 3^2 problems, each taking $\frac{n}{2}$ work, total work is $3^2 \cdot \frac{n}{2} = 3^2 \cdot \frac{2n}{2^2}$.

...

In Layer k: 3^k problems, each taking $\frac{n}{2^{k-1}}$ work, total work is $3^k \cdot \frac{2n}{2^k}$.

We then need to determine the total number of layers in the tree. We find that in the k-th layer, n will be divided by 2^{k-1}, i.e., $\frac{n}{2^{k-1}}$. The final layer would divide n to 1, let $\frac{n}{2^{k-1}} = 1$, then we have $k = 1 + \log_2 n$, which is the total number of layers. As a result, the total computational cost is $3^0 \cdot \frac{2n}{2^0} + 3^1 \cdot \frac{2n}{2^1} + 3^2 \cdot \frac{2n}{2^2} + \cdots + 3^k \cdot \frac{2n}{2^k} = 2n \cdot \left((\frac{3}{2})^0 + (\frac{3}{2})^1 + (\frac{3}{2})^2 + \cdots + (\frac{3}{2})^k\right) = 2n \cdot \frac{1-(\frac{3}{2})^{k+1}}{1-\frac{3}{2}} = 4n \cdot \left((\frac{3}{2})^{k+1} - 1\right) = 4n \cdot \left((\frac{3}{2})^{2+\log_2 n} - 1\right) = 4n \cdot (\frac{3}{2})^2 \cdot (\frac{3}{2})^{\log_2 n} - 4n = 9n \cdot n^{\log_2 1.5} - 4n = 9 \cdot n^{1+\log_2 1.5} - 4n = 9 \cdot n^{\log_2 3} - 4n = O(n^{\log_2 3}) \approx O(n^{1.58})$.

4. **Practice with the iteration method.** We have already had a recurrence relation of an algorithm, which is $T(n) = 4T(n/2) + n^2 \log n$.

   (a) (5 points) Solve this recurrence relation, i.e., express it as $T(n) = O(f(n))$, by using the iteration method.

$$T(n) = 4T(n/2) + n^2 \log n \tag{1}$$

Solve for $T(n/2)$ with (1) we have:

$$T(n/2) = 4T((n/2)/2) + (n/2)^2 \log(n/2) = 4T(n/4) + n^2/4 \log(n/2) \tag{2}$$

By substituting (2) into (1), we have:

$$T(n) = 4[4T(n/4) + n^2/4 \log(n/2)] + n^2 \log n = 16T(n/4) + n^2 \log(n/2) + n^2 \log n \tag{3}$$

Solve for $T(n/4)$ with (3) we have:

$$\begin{aligned} T(n/4) &= 16T((n/4)/4) + (n/4)^2 \log((n/4)/2) + (n/4)^2 log(n/4) \\ &= 16T(n/16) + (n^2/16) \log(n/8) + (n^2/16) \log(n/4) \end{aligned} \tag{4}$$

By substituting (4) into (3) we have:

$$\begin{aligned} T(n) &= 16[16T(n/16) + (n^2/16)\log(n/8) + (n^2/16)\log(n/4)] + n^2\log(n/2) + n^2\log n \\ &= 256T(n/16) + n^2\log(n/8) + n^2\log(n/4) + n^2\log(n/2) + n^2\log n \end{aligned} \tag{5}$$

This gives the general form:

$$\begin{aligned} T(n) &= 4^kT(n/2^k) + n^2[\log n + \log(n/2) + \log(n/4) + \cdots + \log(n/2^{k-1})] \\ &= 4^kT(n/2^k) + n^2\log(n^k/(2\cdot 4\cdot 8\cdots 2^{k-1})) \\ &= 4^kT(n/2^k) + n^2\log\left(\frac{n^k}{2^{k(k-1)/2}}\right) \end{aligned} \tag{6}$$

In order for $T(n/2^k)$ to decrease to $T(1)$, we should have $2^k = n$, i.e., $k = \log n$, as a result, we have:

$$\begin{aligned} T(n) &= 4^kT(n/2^k) + n^2\log\left(\frac{n^k}{2^{k(k-1)/2}}\right) \\ &\le 4^kT(n/2^k) + n^2\log(n^k) \\ &= 4^{\log n}T(1) + n^2\log(n^{\log n}) \\ &= n^2T(1) + n^2\log^2 n \end{aligned} \tag{7}$$

Because by default $T(1) = 1$, we thus have $T(n) \le n^2 + n^2\log^2 n$, i.e., $T(n) = O(n^2 + n^2\log^2 n) = O(n^2\log^2 n)$.

(b) (5 points) Prove, by using mathematical induction, that the iteration rule you have observed in 4(a) is correct and you have solved the recurrence relation correctly. [**Hint**: You can write out the general form of $T(n)$ at the iteration step t, and prove that this form is correct for any iteration step t by using mathematical induction. Then by finding out the eventual number of t and substituting it into your general form of $T(n)$, you get the $O(\cdot)$ notation of $T(n)$.]

The iteration rule we summarized in 4(a) is:

$$T(n) = 4^kT(n/2^k) + n^2\log\left(\frac{n^k}{2^{k(k-1)/2}}\right) \tag{8}$$

Initial Sate: When $k = 1$, (8) is actually $T(n) = 4T(n/2) + n^2\log n$, which is true because of the recursion relation itself.

Induction: Suppose the iteration rule is true for any integer $\le k$, we prove the iteration rule holds for $k+1$. By applying the recursion relation $T(n) = 4T(n/2) + n^2\log n$ to $T(n/2^k)$ in (8) we have:

$$T(n/2^k) = 4T(n/2^{k+1}) + (n/2^k)^2\log(n/2^k) \tag{9}$$

By substituting (9) to (8) we have:

$$
\begin{aligned}
T(n) &= 4^k[4T(n/2^{k+1}) + (n/2^k)^2\log(n/2^k)] + n^2\log\left(\frac{n^k}{2^{k(k-1)/2}}\right) \\
&= 4^{k+1}T(n/2^{k+1}) + n^2\log\left(\frac{n}{2^k}\right) + n^2\log\left(\frac{n^k}{2^{k(k-1)/2}}\right) \\
&= 4^{k+1}T(n/2^{k+1}) + n^2\log\left(\frac{n^{k+1}}{2^{k+k(k-1)/2}}\right) \\
&= 4^{k+1}T(n/2^{k+1}) + n^2\log\left(\frac{n^{k+1}}{2^{(k+1)k/2}}\right)
\end{aligned}
\tag{10}
$$

This is exactly our iteration rule (9) with $k+1$ as the parameter, which means that the iteration rule still holds for $k+1$.

Based on the initial rule and induction step, we proved that the iteration rule we found out is correct.

5. **Practice with the Master Theorem**. Solve the following recurrence relations; i.e. express each one as $T(n) = O(f(n))$ for the tightest possible function $f(n)$ using the Master Theorem, and give a short justification. Unless otherwise stated, assume $T(1) = 1$. **[To see the level of detail expected, we have worked out the first one for you.]**

   (z) $T(n) = 6T(n/6) + 1$. We apply the master theorem with $a = b = 6$ and with $d = 0$. We have $a > b^d$, and so the running time is $O(n^{\log_6(6)}) = O(n)$.

   (a) (5 points) $T(n) = 3T(n/4) + \sqrt{n}$
   $T(n) = O(n^{\log_4 3})$, using the Master Theorem with $a = 3, b = 4, d = 1/2$, we have $a > b^d$, so the answer is $O(n^{\log_b a})$.

   (b) (5 points) $T(n) = 7T(n/3) + \Theta(n^3)$
   $T(n) = O(n^3)$, using the Master Theorem with $a = 7, b = 3, d = 3$. (Notice that the $\Theta(n^3)$ expression is $O(n^3)$ as well, but not the other direction, according to their definitions.) Then $a < b^d$, so the running time is $O(n^d) = O(n^3)$.

   (c) (5 points) $T(n) = 2T(n/3) + n^c$, where $c \geq 1$ is a constant that doesn't depend on n.
   $T(n) = O(n^c)$. We have $a = 2, b = 3, d = c$, because $c \geq 1$, we always have $2 < 3^c$, i.e., $a < b^d$, thus the running time is $O(n^c)$.

6. **Proof of the Master Theorem.** (15 points) Now that we have practiced with the recursion tree method, the iteration method, and the Master method. The Master Theorem states that, suppose $T(n) = a \cdot T(n/b) + O(n^d)$, we have:

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

   Prove that the Master Theorem is true by using either the recursion tree method or the iteration method.

$$T(n) = a \cdot T(n/b) + O(n^d) \tag{11}$$

Solve for $T(n/b)$ we have:

$$T(n/b) = a \cdot T((n/b)/b) + O((n/b)^d) = a \cdot T(n/b^2) + O((n/b)^d) \tag{12}$$

By substituting (12) to (11) we have:

$$T(n) = a[a \cdot T(n/b^2) + O((n/b)^d)] + O(n^d) = a^2 \cdot T(n/b^2) + a \cdot O((n/b)^d) + O(n^d) \tag{13}$$

Solve for $T(n/b^2)$ with (13) we have:

$$\begin{aligned} T(n/b^2) &= a^2 \cdot T((n/b^2)/b^2) + a \cdot O(((n/b^2)/b)^d) + O((n/b^2)^d) \\ &= a^2 \cdot T(n/b^4) + a \cdot O((n/b^3)^d) + O((n/b^2)^d) \end{aligned} \tag{14}$$

By substituting (14) to (13) we have:

$$\begin{aligned} T(n) &= a^2[a^2 \cdot T(n/b^4) + a \cdot O((n/b^3)d) + O((n/b^2)^d)] + a \cdot O((n/b)^d) + O(n^d) \\ &= a^4 \cdot T(n/b^4) + a^3 \Delta O((n/b^3)^d) + a^2 \Delta O((n/b^2)^d) + a \cdot O((n/b)^d) + O(n^d) \end{aligned} \tag{15}$$

So the general iteration rule is:

$$\begin{aligned} T(n) &= a^k \cdot T(n/b^k) + a^{k-1} \Delta O((n/b^{k-1})^d) + \cdots + a^3 \Delta O((n/b^3)^d) + a^2 \Delta O((n/b^2)^d) + a \cdot O((n/b)^d) + O \\ &= a^k \cdot T\left(\frac{n}{b^k}\right) + O\left(\frac{n^d a^{k-1}}{(b^d)^{k-1}} + \cdots + \frac{n^d a^3}{(b^d)^3} + \frac{n^d a^2}{(b^d)^2} + \frac{n^d a}{b^d} + n^d\right) \\ &= a^k \cdot T\left(\frac{n}{b^k}\right) + O\left(n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \left(\frac{a}{b^d}\right)^3 + \cdots + \left(\frac{a}{b^d}\right)^{k-1}\right)\right) \end{aligned} \tag{16}$$

To decrease $T\left(\frac{n}{b^k}\right)$ to $T(1)$, we have $b^k = n$, thus $k = \log_b n$, in this case, we have:

$$\begin{aligned} T(n) &= a^k \cdot T(1) + O\left(n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \left(\frac{a}{b^d}\right)^3 + \cdots + \left(\frac{a}{b^d}\right)^{k-1}\right)\right) \\ &= a^k + O\left(n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \left(\frac{a}{b^d}\right)^3 + \cdots + \left(\frac{a}{b^d}\right)^{k-1}\right)\right) \end{aligned} \tag{17}$$

Notice that the later part is the sum of a geometric series with common ratio $r = \frac{a}{b^d}$, so:

- Case 1: if $r = 1$, i.e., $a = b^d$ and $d = \log_b a$, then $T(n) = a^k + O(n^d \cdot k) = a^{\log_b n} + O(n^d \cdot \log_b n) = n^{\log_b a} + O(n^d \cdot \log_b n) = n^d + O(n^d \cdot \log_b n) = O(n^d \cdot \log_b n) = O(n^d \log n)$.
- Case 2: if $r < 1$, i.e., $a < b^d$ and $d > \log_b a$, then $T(n) = a^k + O(n^d \cdot \frac{1-r^k}{1-r}) = n^{\log_b a} + O(n^d \cdot \frac{1-r^k}{1-r})$, because $\frac{1-r^k}{1-r}$ is a positive constant, then $T(n) = n^{\log_b a} + O(n^d)$, and because $d > \log_b a$, we have $T(n) = O(n^d)$.
- Case 3: if $r > 1$, i.e., $a > b^d$ and $d < \log_b a$, we still have $T(n) = a^k + O(n^d \cdot \frac{1-r^k}{1-r}) = n^{\log_b a} + O(n^d \cdot \frac{1-r^k}{1-r})$, because $\frac{1-r^k}{1-r}$ is still a positive constant, then we have $T(n) = n^{\log_b a} + O(n^d)$, but because in this case $d < \log_b a$, thus $T(n) = O(n^{\log_b a})$.

7. **Algorithm design.** Each of n users spends some time on a social media site. For each $i = 1, \ldots, n$, user i enters the site at time a_i and leaves at time $b_i \geq a_i$. You are interested in the question: how many distinct pairs of users are ever on the site at the same time? (Here, the pair (i, j) is the same as the pair (j, i)).

   Example: Suppose there are 5 users with the following entering and leaving times:

User	Enter time	Leave time
1	1	4
2	2	5
3	7	8
4	9	10
5	6	10

   Then, the number of distinct pairs of users who are on the site at the same time is three: these pairs are $(1, 2)$, $(4, 5)$, $(3, 5)$. (Drawing the intervals on a number line may make this easier to see).

   (a) (10 points) Given input $(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)$ as above in no particular order (i.e., not sorted in any way), describe a straightforward algorithm that takes $\Theta(n^2)$-time to compute the number of pairs of users who are ever on the site at the same time, and explain why it takes $\Theta(n^2)$-time. **[We are expecting pseudocode and a brief justification for its runtime.]**

   Our algorithm will run as follows:

   - Initialize variable *count* to 0.
   - For every user i, we check every other user $j \neq i$:
     If $a_i \leq a_j \leq b_i$ or $a_j \leq a_i \leq b_j$,then user i and user j are on the site at the same time. Increment *count* by 1.
   - We return *count*/2, which counts the number of distinct pairs of users who are ever on the site at the same time. We divide *count* by 2 because we double counted each pair.

   Running time analysis: For each user $i = 1, \cdots, n$, we iterate over all other users ($\Theta(n)$ of them) to check for the above inequality which can be done in constant time. Therefore, the algorithm runs in $\Theta(n^2)$.

   (b) (10 points) Give an $\Theta(n \log(n))$-time algorithm to do the same task and analyze its running time. (**Hint:** consider sorting relevant events by time). **[We are expecting pseudocode and a brief justification for its runtime.]**

   The key here is to realize that we can decouple the start and exit times from the user. Here is our algorithm:

   - Initialize variable *count* and variable *usersOnsite* to 0. Note that at time t_i, *usersOnSite* is equal to the current number of users on the site.
   - We produce a combined list l of entry and exit times. l has $2n$ tuples. The first element in each tuple is the entry/exit time and the second element is a binary indicating "entry" or "exit". Thus, user i can be split into (a_i, "*enter*") and (b_i, "*exit*").

- Next, we sort list l by the first element in each tuple using $MergeSort$. For each tuple p in the sorted list l, we check:
  If p[1] is "enter":
  - If $usersOnSite \geq 1$, increment $count$ by the value of $usersOnSite$.
  - Increment $usersOnSite$ by 1.

  If p[1] is "exit", decrement $usersOnSite$ by 1.
- Return count.

Running time analysis: Initialize list l takes $\Theta(n)$ time, since we are iterating over all user entry and exit times, generating 2 tuples for each user. Sorting the list using $MergeSort$ takes $\Theta(n \log n)$ time. Iterating through the sorted list l takes $\Theta(n)$ time, since there are a total of $2n$ tuples and each iteration takes constant time to execute. Overall, the algorithm takes $\Theta(n \log n)$ time.