CMSC 27200 - Problem Set 4

Sohini Banerjee

February 2, 2024

0

Note: I am using the 3-day late pass for this assignment!
Collaborators: Parnika Saxena
Sources: Lecture Notes, Office Hours

1a

$T(n) = T(\frac{n}{2}) + 2n, T(1) = 1$
$T(n) = (T(\frac{n}{4}) + 2(\frac{n}{2})) + 2n = T(\frac{n}{4}) + 2n + n$
$T(n) = (T(\frac{n}{8}) + 2(\frac{n}{4})) + 2n + n = T(\frac{n}{8}) + 2n + n + \frac{1}{2}n$
$T(n) = T(\frac{n}{2^k}) + 2n \sum_{j=0}^{k-1} (\frac{1}{2})^j$
We can simplify: $\sum_{j=0}^{k-1} (\frac{1}{2})^j = \frac{1(1-(\frac{1}{2})^k)}{1-\frac{1}{2}} = 2(1 - (\frac{1}{2})^k) = 2 - 2(\frac{1}{2})^k$
$T(n) = T(\frac{n}{2^k}) + 2n(2 - 2(\frac{1}{2})^k)$
$T(n) = T(\frac{n}{2^k}) + 4n(1 - (\frac{1}{2})^k)$
Let $k = \log_2 n$. Then, $2^k = n$ and $\frac{n}{2^k} = 1$.
$T(n) = T(1) + 4n(1 - (\frac{1}{2})^{\log_2 n})$
$T(n) = T(1) + 4n(1 - \frac{1}{n})$
$T(n) = 1 + 4n - 4$
$\boxed{T(n) = 4n - 3}$
Therefore, $T(n)$ is $\boxed{O(n)}$.

1b

$T(n) = 2T(\frac{n}{2}) + 3n^2, T(1) = 8$
$T(n) = 2(2T(\frac{n}{4}) + 3(\frac{n}{2})^2) + 3n^2 = 4T(\frac{n}{4}) + 3n^2 + \frac{3}{2}n^2$
$T(n) = 4(2T(\frac{n}{8}) + 3(\frac{n}{4})^4) + 3n^2 + \frac{3}{2}n^2 = 8T(\frac{n}{8}) + 3n^2 + \frac{3}{2}n^2 + \frac{3}{4}n^2$
$T(n) = 2^k T(\frac{n}{2^k}) + 3n^2 \sum_{j=0}^{k-1} (\frac{1}{2})^j$
We can simplify: $\sum_{j=0}^{k-1} (\frac{1}{2})^j = \frac{1(1-(\frac{1}{2})^k)}{1-\frac{1}{2}} = 2(1 - (\frac{1}{2})^k) = 2 - 2(\frac{1}{2})^k$
$T(n) = 2^k T(\frac{n}{2^k}) + 3n^2(2 - 2(\frac{1}{2})^k)$
$T(n) = 2^k T(\frac{n}{2^k}) + 6n^2(1 - 1(\frac{1}{2})^k)$
Let $k = \log_2 n$. Then, $2^k = n$ and $\frac{n}{2^k} = 1$.
$T(n) = nT(1) + 6n^2(1 - (\frac{1}{2})^{\log_2 n})$
$T(n) = 8n + 6n^2(1 - \frac{1}{n})$
$T(n) = 8n + 6n^2 - 6n$
$\boxed{T(n) = 6n^2 + 2n}$
Therefore, $T(n)$ is $\boxed{O(n^2)}$.

1c

$T(n) = 4T(\frac{n}{2}) + n^2 + 3, T(1) = 1$
$T(n) = 4(4T(\frac{n}{4}) + (\frac{n}{2})^2 + 3) + n^2 + 3 = 16T(\frac{n}{4}) + 2n^2 + 12 + 3$
$T(n) = 16(4T(\frac{n}{8}) + (\frac{n}{4})^2 + 3) + 2n^2 + 12 + 3 = 64T(\frac{n}{8}) + 3n^2 + 48 + 12 + 3$
$T(n) = 4^kT(\frac{n}{2^k}) + kn^2 + \sum_{j=0}^{k-1} 3 \cdot 4^j$
We can simplify: $\sum_{j=0}^{k-1} 3 \cdot 4^j = \frac{3(1-4^k)}{1-4} = 4^k - 1$
$T(n) = 4^kT(\frac{n}{2^k}) + kn^2 + (4^k - 1)$
Let $k = \log_2 n$. Then, $2^k = n$ and $\frac{n}{2^k} = 1$. Also, $n^2 = (2^k)^2 = (2^2)^k = 4^k$.
$T(n) = n^2T(1) + n^2 \log_2 n + (4^{\log_2 n} - 1)$
$T(n) = n^2 + n^2 \log_2 n + (n^2 - 1)$
$\boxed{T(n) = n^2 \log_2 n + 2n^2 - 1}$
Therefore, $T(n)$ is $\boxed{O(n^2 \log_2 n)}$.

1d

$T(n) = 10T(\frac{n}{2}) - 16T(\frac{n}{4}) + 6n + 7, T(1) = 1, T(2) = 9$
Let $10T(\frac{n}{2}) - 16T(\frac{n}{4})$ be the homogeneous part of the recurrence and $6n + 7$ be the inhomogeneous part of the recurrence.

First, we will solve the homogeneous part of the recurrence.
$T_0(n) = n^p = 10(\frac{n}{2})^p - 16(\frac{n}{4})^p$
$n^p = 10\frac{n^p}{2^p} - 16\frac{n^p}{4^p}$
$\frac{10}{2^p} - \frac{16}{(2^p)^2} = 1$
$(2^p)^2 - 10(2^p) + 16 = 0$
We can let $x = 2^p$.
$x^2 - 10x + 16 = 0$
$(x - 8)(x - 2) = 0$
$x = 8, 2$
This means $2^p = 8$ so $p = 3$ and $2^p = 2$ so $p = 1$.
Therefore, the homogeneous part of the recurrence is $T_0(n) = c_1n + c_2n^3$.

Now, we will solve the inhomogeneous part of the recurrence.
$T_1(n) = c_3n \log_2 n + c_4$
$c_3n \log_2 n + c_4 = 10(c_3(\frac{n}{2}) \log_2(\frac{n}{2}) + c_4) - 16(c_3(\frac{n}{4}) \log_2(\frac{n}{4}) + c_4) + 6n + 7$
$c_3n \log_2 n + c_4 = 5c_3n \log_2(\frac{n}{2}) + 10c_4 - 4c_3n \log_2(\frac{n}{4}) - 16c_4 + 6n + 7$
$c_3n \log_2 n + c_4 = 5c_3n(\log_2 n - 1) + 10c_4 - 4c_3n(\log_2 n - 2) - 16c_4 + 6n + 7$
$c_3n \log_2 n + c_4 = c_3n \log_2 n + 3c_3n - 6c_4 + 6n + 7$
This means $0 = (3c_3 + 6)n$ so $c_3 = -2$ and $-6c_4 + 7 = c_4$ so $c_4 = 1$.
Therefore, the inhomogeneous part of the recurrence is $T_1(n) = -2n \log_2 n + 1$.

Now, we will use the initial conditions to solve for c_1 and c_2.
$T(n) = c_1n + c_2n^3 - 2n \log_2 n + 1$
$T(1) = 1$, so $c_1 + c_2 + 1 = 1$, so $c_1 = -c_2$.
$T(2) = 9$, so $2c_1 + 8c^2 - 4 \log_2 2 + 1 = 2c_1 - 8c_1 - 4 + 1 = -6c_1 - 3 = 9$, so $c_1 = -2$ and $c_2 = 2$.
$\boxed{T(n) = -2n + 2n^3 - 2n \log_2 n + 1}$
Therefore, $T(n)$ is $\boxed{O(n^3)}$.

2

Note, we will use K to denote the index K in 0-based indexing and k to denote the kth element in the sorted merged array. For example, if $k = 5$, then $K = 4$, such that we find the 5th element, which appears at index 4 of the sorted

merged array. Thus, $K = k - 1$ for the rest of this problem. In general, we will apply a binary search algorithm on both arrays to narrow down where index K is located.

Algorithm

- Stored Data: Keep track of the left and right bounds of the A and B subarray.
- Initialization: Let $leftA = 0$ and $rightA = n - 1$ be the bounds of the A subarray. Similarly, let $leftB = 0$ and $rightB = m - 1$ be the bounds of the B subarray.
- Recursive Step:
  - Base Case: If $rightA < leftA$, return the $(K - leftA)$th element in the B subarray. Similarly, if $rightB < leftB$, return the $(K - leftB)$th element in the A subarray.
  - Otherwise, calculate the midpoint of the A and B subarray, $midA$ and $midB$.
    - If $A[midA] < B[midB]$:
      - If $K \leq midA + midB$: set $rightB = midB - 1$ (removes upper half of B subarray)
      - $K > midA + midB$: set $leftA = midA + 1$ (removes lower half of A subarray)
    - If $A[midA] > B[midB]$:
      - $K \leq midA + midB$: set $rightA = midA - 1$ (removes upper half of A subarray)
      - $K > midA + midB$: set $leftB = midB + 1$ (removes lower half of B subarray)
  - Call the recursive function with the updated $leftA$, $rightA$, $leftB$, and $rightB$ subarray bounds.

Pseudocode

```
def Kth(leftA, rightA, leftB, rightB):
    if leftA > rightA, do:
        ret B[K - leftA]
    if leftB > rightB, do:
        ret A[K - leftB]

    set midA = (leftA + rightA) // 2
    set midB = (leftB + rightB) // 2
    if A[midA] < B[midB], do:
        if K <= midA + midB, do:
            ret Kth(leftA, rightA, leftB, midB - 1)
        else, do:
            ret Kth(midA + 1, rightA, leftB, rightB)
    else, do:
        if K <= midA + midB, do:
            ret Kth(leftA, midA - 1, leftB, rightB)
        else, do:
            ret Kth(leftA, rightA, midB + 1, rightB)
```

Explanation

At each step, we aim to decrease the size of one subarray by half. We determine the midpoint of each subarray, $midA$ and $midB$. We make the following observations:

- $midA$ is the number of elements that lie to the left of $midA$ in subarray A
- $midB$ is the number of elements that lie to the left of $midB$ in subarray B
- $T = midA + midB$ is the number of elements that lie to the left of $midA$ in subarray A and $midB$ in subarray B

We know that either $K \leq T$ or $K > T$, and either $A[midA] < B[midB]$ or $A[midA] > B[midB]$, so we can outline 4 cases below:

- If $A[midA] < B[midB]$: This means $A[midA]$ comes before $B[midB]$ in the sorted merged array.
  - There are at least $T+1$ elements before $B[midB]$ in the sorted merged array. This includes $A[midA]$ since $A[midA] < B[midB]$, $midA$ number of elements before $A[midA]$, and $midB$ number of elements before $B[midB]$, for a total of at least $T+1$ elements. This means there is index 0 to at least T before $B[midB]$ in the sorted merged array.
  - There are at most $T+1$ elements before and including $A[midA]$ in the sorted merged array. This includes $A[midA]$, $midA$ number of elements before $A[midA]$, and $midB$ number of elements before $B[midB]$, for a total of at most $T+1$ elements. This means there is index 0 to at most T before and including $A[midA]$ in the sorted merged array.
  - Based on these observations, we narrow our search for the following cases:
    - $K \leq T$: Since there is index 0 to at least T before $B[midB]$ in the sorted merged array, we eliminate $midB$ and everything after in subarray B by setting $rightB = midB - 1$.
    - $K > T$: Since there is index 0 to at most T before and including $A[midA]$ in the sorted merged array, we eliminate $midA$ and everything before in subarray A by setting $leftA = midA + 1$.
- If $A[midA] > B[midB]$: This means $B[midB]$ comes before $A[midA]$ in the sorted merged array.
  - There are at least $T+1$ elements before $A[midA]$ in the sorted merged array. This includes $B[midB]$ since $B[midB] < A[midA]$, $midA$ number of elements before $A[midA]$, and $midB$ number of elements before $B[midB]$, for a total of at least $T+1$ elements. This means there is index 0 to at least T before $A[midA]$ in the sorted merged array.
  - There are at most $T+1$ elements before and including $B[midB]$ in the sorted merged array. This includes $B[midB]$, $midA$ number of elements before $A[midA]$, and $midB$ number of elements before $B[midB]$, for a total of at most $T+1$ elements. This means there is index 0 to at most T before and including $B[midB]$ in the sorted merged array.
  - Based on these observations, we narrow our search for the following cases:
    - $K \leq T$: Since there is index 0 to at least T before $A[midA]$ in the sorted merged array, we eliminate $midA$ and everything after in subarray A by setting $rightA = midA - 1$.
    - $K > T$: Since there is index 0 to at most T before and including $B[midB]$ in the sorted merged array, we eliminate $midB$ and everything before in subarray B by setting $leftB = midB + 1$.

Finally, we consider the termination case. If $leftA > rightA$, we it means we have exhausted all possibilities for the Kth element in array A. Therefore, it must lie in array B. In particular, we return the $(K - leftA)$th element in B since $leftA$ represents the number of elements smaller than the overall Kth element. The same argument applies for $leftB > rightB$.

Runtime

At each step, we are reducing our search space by half. So, for array A, there can be at most $\log n$ recursive calls and similarly, for array B, there can be at most $\log m$ recursive calls. This means that the total runtime is $O(\log n + \log m)$.

3

Suppose we have an array A of size n that stores $x_1, ..., x_n$, such that $A[0] = x_1$, ..., $A[n-1] = x_n$. Our algorithm is to calculate all possible pair sums and use a two pointer strategy to find 2 pair sums that add to our expected sum S.

Algorithm

Step 1 : Calculate all possible sums. Store this in *sums*.

- Stored Data: Store all possible sums in a list *sums*.
- Initialization: Let *sums* be an empty list.

- Iterative Step: For each $0 \leq i < n$ and $0 \leq j < n$ such that $i < j$, add the object $(s = x_i + x_j, i, j)$ to *sums*. After this, sort *sums* by s.

Step 2 : Find all unique pair sums and maximum i and minimum j for each.

- Stored Data: Store the new sums in a list *new_sums*.
- Initialization: Let *new_sums* be an empty list.
- Iterative Step: For each sum in *sums*, find the maximum index i such that there exists a j where $i < j$ and $x_i + x_j = s$ and find the minimum index j such that there exists a i where $i < j$ and $x_i + x_j = s$. Store the sum, maximum index i, and minimum index j as an object in *new_sums*. This can be done by iterating through *sums* since duplicate pair sums are adjacent since *sums* is in sorted order.

Step 3 : Use two pointers to find sums in *new_sums* that add up to S.

- Stored Data: Keep track of $left$ and *right* pointers of the *new_sums* array.
- Initialization: Let $left = 0$ and $right = len(new\_sums) - 1$.
- Iterative Step: If $left > right$, return $False$. Otherwise, calculate the current total sum *current_S*.
  - If $current\_S < S$, increment $left$.
  - If $current\_S > S$, decrement *right*.
  - If $current\_S = S$:
    - If minimum j of $left$ sum is less than maximum i of *right* sum OR if minimum j of *right* sum is less than maximum i of the $left$ sum, return $True$.
    - Otherwise, increment $left$ and decrement *right*.

Pseudocode

```
def does_sum_exist(A, S):
    n = len(A)

    sums = []
    for i in range(n):
        for j in range(i + 1, n):
            s = A[i] + A[j]
            sums.append((s, i, j))
    sums.sort()

    new_sums = []
    index = 0
    while index < len(sums):
        next_index = index + 1
        while next_index < len(sums) and sums[index][0] == sums[next_index][0]:
            new_s = sums[index][0]
            new_max_i = max(sums[index][1], sums[next_index][1])
            new_min_j = min(sums[index][2], sums[next_index][2])
            sums[index] = (new_s, new_max_i, new_min_j)

            next_index += 1
        new_sums.append(sums[index])
        index = next_index

    left, right = 0, len(new_sums) - 1
    while left <= right:
        current_S = new_sums[left][0] + new_sums[right][0]
        if current_S < S:
```

```
            left += 1
        if current_S > S:
            right -= 1
        if current_S == S:
            if new_sums[left][2] < new_sums[right][1] or new_sums[right][2] < new_sums[left][1]:
                return True
            else:
                left += 1
                right -= 1

    return False
```

Explanation

The correctness of the algorithm is determined at step 3. At this point, we have all unique sums of 2 numbers and stored the maximum i and minimum j for which there exists another index to produce the sum. Given this, we reduce our problem to finding 2 sums of 2 numbers that add up to S.

Since new_sums is in sorted order of sums, we can initialize $left$ and $right$ pointers at opposite ends of the array. From here, we can get the following conditions:

- $new\_sums[left] + new\_sums[right] < S$: In this case, we have traversed all elements of new_sums that lie after $right$, meaning to increase our current total sum, we must increment $left$ because $new\_sums[left + 1] > new\_sums[left]$.
- $new\_sums[left]+new\_sums[right] > S$: In this case, we have traversed all elements of new_sums that lie before $left$, meaning to decrease our current total sum, we must decrement $right$ because $new\_sums[right - 1] < new\_sums[right]$.
- $new\_sums[left] + new\_sums[right] = S$: In this case, we know there exists 2 pairs of sums that add up to S. However, we have to verify if they are legitimate. Define the following:
  - Let s_1 denote the first sum and s_2 denote the second sum.
    - Let i_1 be the maximum index i such that there exists some j where $i_1 < j$ and $x_{i_1} + x_j = s_1$.
    - Let i_2 be the maximum index i such that there exists some j where $i_2 < j$ and $x_{i_2} + x_j = s_2$.
    - Let j_1 be the minimum index j such that there exists some i where $i < j_1$ and $x_i + x_{j_1} = s_1$.
    - Let j_2 be the minimum index j such that there exists some i where $i < j_2$ and $x_i + x_{j_2} = s_2$.
  - For this current solution to be legitimate, one of the following has to hold:
    - $j_1 < i_2$: In this case, there is some $a < j_1$ where $x_a + x_{j_1} = s_1$ and $i_2 < b$ where $x_{i_2} + x_b = s_2$. Thus, $a < j_1 < i_2 < b$ and $x_a + x_{j_1} + x_{i_2} + x_b = s_1 + s_2 = S$, so a valid solution exists.
    - $j_2 < i_1$: In this case, there is some $a < j_2$ where $x_a + x_{j_2} = s_2$ and $i_1 < b$ where $x_{i_1} + x_b = s_1$. Thus, $a < j_2 < i_1 < b$ where $x_a + x_{j_2} + x_{i_1} + b = s_2 + s_1$, so a valid solution exists.
  - In particular, we must check both of these conditions because whether each pair sum is produced before or after the other is arbitrary, as new_sums is sorted by sum, not index. If none of these conditions hold, we increment $left$ and decrement $right$. The reason we have to modify both $left$ and $right$ is because each pair sum in new_sums is unique and $new\_sums[left] + new\_sums[right] = S$, so incrementing or decrementing only one of $left$ or $right$ cannot ever produce S since only one number is changing.

Finally, note that we terminate once $left$ has passed $right$. The reason we can allow $left$ and $right$ to be equal is that can represent two sets of identical sums at four different indices. For example, suppose that $left = right$ and $S[left][1] = i$ and $S[left][2] = j$. This means there exists an a such that $i < a$ and $x_i + x_a = s$ and there exists a b such that $b < j$ and $x_b + x_j = s$. If $S[left][2] < S[left][1]$, this means that $j < i$, so $b < j < i < a$, so $x_b + x_j + x_i + x_a = s + s = S$.

Runtime

- **Step 1** : We iterate through at most n elements in each loop since there a total of n^2 possible pair sums. Then, we sort the n^2 sums, which takes $n^2 \log n$ time. Therefore, this step is $O(n^2 \log n)$.
- **Step 2** : We iterate through n^2 sums in one loop. Although there is an internal loop, we update *index* to match the end of the internal loop, so each element in *sums* is explored once. Therefore, this step is $O(n^2)$.
- **Step 3** : Using two pointers for *new_sums*, of which there are at most n^2, we hit each sum exactly once. Therefore, this step is $O(n)$.

Combining all the steps, this algorithm takes $O(n^2 \log n)$ time.

4

We will apply a divide and conquer algorithm to find all the visible lines. The idea is that we find the visible lines among lower sloped lines and visible lines among higher sloped lines. Them, we combine these solutions appropriately to get the visible lines among both these sets of lines.

Algorithm

- Stored Data: Keep track of the visible lines with lower slopes *lower_visible_lines* and higher slopes *higher_visible_lines*. Note that due to the implementation of the recursive step, every line in *lower_visible_lines* will be visible with each other and in sorted order by slope. Similarly, every line in *higher_visible_lines* will be visible with each other and in sorted order by slope.
- Initialization: Sort all the lines by slope. For all lines with the same slope, remove the line with the smaller y-intercept until every line has a distinct slope. Let these set of lines be $L = [L_1, ..., L_m]$.
- Recursive Step:
  - If there are fewer than 3 lines, all must be visible. Thus, we return all the lines.
  - If there are exactly 3 lines, either 2 or 3 are visible. If the first and second line intersect before the first and third, then all three lines are visible and we return all the lines. Otherwise, we return the first and third lines.
  - If there are more than 3 lines, we split the lines into *lower_lines* and *higher_lines* based on slope. Then, we call *visible_lines* for *lower_lines*, which gives *lower_visible_lines*, and for *higher_lines*, which gives *higher_visible_lines*. *lower_visible_lines* and *higher_visible_lines* are visible among each other and sorted by slope. We call *combine_visible_lines* with *lower_visible_lines* and *higher_visible_lines*, which returns the lines visible among all lines in *lower_visible_lines* and *higher_visible_lines*. It works as follows:
    - If there are fewer than 3 lines total in *lower_visible_lines* and *higher_visible_lines*, return the concatenation of *lower_visible_lines* and *higher_visible_lines*.
    - Otherwise, initialize *all_visible_lines* as *lower_visible_lines*. As we consider each line in *higher_visible_lines*, let the line under consideration be called *next_highest_line*. Consider the following for *next_highest_line*:
      - If there are fewer than 2 lines in *all_visible_lines*, append *next_highest_line* to *all_visible_lines*.
      - Otherwise, let the second most recent line added to *all_visible_lines* be called *penult* and most recent line added to *all_visible_lines* be called *ult*. Until *penult* and *ult* intersect before *penult* and *next_highest_line*, remove *ult* (and update *penult* and *ult* with second most recent and recent line added to *all_visible_lines*). Then, add *next_highest_line* to *all_visible_lines*.
    - Finally, return *all_visible_lines*.

Pseudocode

```
def delete_middle(first_line, second_line, third_line):
    if intersectX(first_line, second_line) > intersectX(first_line, third_line):
        return True
```

```
def visible_lines(L):
    if size(L) < 3, do:
        return L
    else if size(L) == 3, do:
        set first_line = L[0]
        set second_line = L[1]
        set third_line = L[2]

        if delete_middle(first_line, second_line, third_line), do:
            L.removes(second_line)
        return L
    else:
        set lower_lines, higher_lines = split_by_slope(L)
        set lower_visible_lines = visible_lines(lower_lines)
        set higher_visible_lines = visible_lines(higher_lines)
        return combine_visible_lines(lower_visible_lines, higher_visible_lines)

def combine_visible_lines(lower_visible_lines, higher_visible_lines):
    if size(lower_visible_lines) + size(higher_visible_lines) < 3, do:
        return concatenate(lower_visible_lines, higher_visible_lines)

    set all_visible_lines = lower_visible_lines

    for next_highest_line in higher_visible_line:
        if size(all_visible_lines) < 2, do:
            all_visible_lines.append(next_highest_line)
        else, do:
            while size(all_visible_lines) >= 2 and delete_middle(all_visible_lines
                [-2], all_visible_lines[-1], next_highest_line), do:
                 all_visible_lines.pop()
            all_visible_lines.append(next_highest_line)

    return all_visible_lines
```

Explanation

Eliminate duplicate slopes
First, we eliminate all lines of duplicate slopes because among multiple parallel lines, keeping only the one with the highest y-intercept since that can be seen over others. Assume from now on, all our lines have different slopes. Now, sort all the lines in order of slope.

Case: less than 3 lines
Consider the case of fewer than 3 lines, showing all lines are visible.

- 0 lines: trivial
- 1 line: visible throughout domain
- 2 lines: one line visible before intersection (single point because there are no parallel lines) and other line visible after intersection

Case: exactly 3 lines
Now, consider the base case of 3 lines, L_1, L_2, and L_3. Assume these lines are sorted in order of slope. L_1 (minimum slope) is always visible and L_3 (maximum slope) is always visible, since their range can extend from a point to $-\infty$ (minimum slope) or a point to ∞ (maximum slope). L_2 visible only if L_1 and L_2 intersect before L_1 and L_3. This is because if L_1 and L_2 intersect after L_1 and L_3 intersect, then L_3 will have taken over L_1 after they intersect, and since L_3 has a higher slope than L_2, L_2 will never get a chance to take over L_1 since L_3 has already taken over.

Divide and conquer strategy
We use a divide and conquer algorithm. When we have at most 3 lines, we can find the visible lines using the procedure above. However, if there are more than 3 lines, we split them into two groups *lower_lines* and *higher_lines*, where *lower_lines* is the set of lines with the lower half of slopes and *higher_lines* is the set of lines with the upper half of slopes. Since the lines are sorted by slope, every line in *lower_lines* has a smaller slope than every line in *higher_lines*. Then, aim to find the visible lines for *lower_lines* and *higher_lines*, calling the results *lower_visible_lines* and *higher_visible_lines*, respectively.

At this point, we must combine the results *lower_visible_lines* and *higher_visible_lines*. Since *lower_visible_lines* are visible among each other, and *higher_visible_lines* are visible among each other, the correct solution to this problem involves finding all the lines among *lower_visible_lines* and *higher_visible_lines* that are visible altogether.

Combining solutions to subproblems
We must combine subproblem solutions, *lower_visible_lines* and *higher_visible_lines*, to get the solution to the overall problem. Due to our base case, we return visible lines such that they are in order of slope. So, we know that *lower_visible_lines* and *higher_visible_lines* are in order of slope.

From here, we must find the final combination of lines between *lower_visible_lines* and *higher_visible_lines*. First, we can let *all_visible_lines* be *lower_visible_lines*, so the initial set of lines in *all_visible_lines* are all visible and in sorted order by slope. Now, as we consider each new line in *higher_visible_lines*, we need to make sure it is visible among all lines in *all_visible_lines*.

Call each new line in *higher_visible_lines* being considered *next_highest_line*, since at this point, it will have the highest slopes among all of *all_visible_lines*. We consider *next_highest_line* against the second most recent and most recent line added to *all_visible_lines*. Call the second most recent line *penult* and most recent line *ult*. The idea is that when adding *next_highest_line*, it will always be visible since it is the maximum sloped line, so we must remove other lines if necessary in *all_visible_lines* such that they can all be seen with *next_highest_line*. Furthermore, we know that if *next_highest_line* can be added to *all_visible_lines* with comparing to *penult* and *ult*, the updated *all_visible_lines* must be visible because the intersections of previous lines occurred before *next_highest_line* intersected with *ult*.

- Suppose these 3 lines can be visible together (checked via base case of 3 lines). This means *next_highest_line* takes over *ult* as the latest visible line (based on x coordinate intersection takeover) after *ult* takes over *penult*. Since all previous intersections occur before that of *ult* and *next_highest_line*, it means *next_highest_line* can be added to *all_visible_lines*.
- Suppose these 3 lines cannot be visible together (checked via base case of 3 lines). Since *next_highest_line* currently has the highest slope of lines considered, it must be visible. Therefore, *ult* must be removed because *penult* and *ult* intersect after *penult* and *next_highest_line*, so *ult* cannot be seen. We continuously remove the most recent line in *all_visible_lines* until we can safely add *next_highest_line* (meaning the second most recent and recent are visible with it, or there is less than 2 lines left in *all_visible_lines*).

Runtime

- Sorting the lines takes $O(n \log n)$ time using mergesort.
- Removing the lines with duplicate slopes and preserving the one with highest y-intercept takes $O(n)$ time since we can simply iterate through the sorted lines and only consider lines without a duplicate slope (as the duplicate is always present right after in the sorted lines).
- Our algorithm replicates mergesort. Since we are splitting the lines in half and combining the solutions, the recursion depth is $\log_2 n$. At each stage, we have to merge two sets of lines together. The time complexity of merging a set of lines A and B is $O(len(A) + len(B))$. Furthermore, we know that each stage, each line can only be considered in a single merge, meaning each line is considered once per stage, so there are $O(n)$ considerations total for merging (across different merges in the same stage). Therefore, the recurrence relation is $T(n) = 2T(\frac{n}{2}) + n$, which has a time complexity of $n \log n$.

The overall time complexity of this algorithm is the number of recursions, $\log n$, by merging time complexity at each stage of recursion, n, so we get an $O(n \log n)$ algorithm.