CSCI-SHU 220: Algorithms Posted: February 11, 2025
Homework 1 Due: 11:55pm (Shanghai time), February 25, 2025

This assignment has a total of 90 base points and 10 extra points, and the cap is 90. Bonus questions are indicated using the $\star$ mark.

Submission Instructions: Please submit to Gradescope (we will post the link later in the announcements). During submission, you need to **mark/map the solution to each question**; otherwise, we may apply a penalty.

Another notice: you only get full credits for the algorithm design questions if your algorithm matches the desirable complexity. If the desirable complexity is not stated, you need to design an algorithm to be as fast as possible.

Please specify the following information before submission:

- Your Name: Yixia Yu
- Your NetID: yy5091

Problem 1: Asymptotic analysis [5×4 pts]

(1) What's the asymptotic relationship between $f(n) = (\log\log n)^{\log n}$ and $g(n) = \log^{\log\log n} n$ and prove it?

(2) What's the asymptotic relationship between $f(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{n}$ and $g(n) = \sum_{i=1}^{n} 2^{-\lceil \log i \rceil}$ and prove it?

(3) Prove that $n! = \omega(n^{0.99n})$.

(4) In the practice table from the lecture, we have examples of f and g satisfying the following combinations of asymptotic relationships: $\{O, o\}$, $\{\Omega, \omega\}$, and $\{\Theta, O, \Omega\}$. Are there any other possible combinations? For every combination not in the listed three (combination also includes the empty set, or any singleton), construct an example of f and g and prove it.

(5) What's the asymptotic relationship between $f(n) = n^{0.6}$ and $g(n) = 2^{2^{\lfloor \log\log n \rfloor}}$ and prove it?

Solution. (1)

$$f(n) = \omega g(n)$$

Proof:

$$\begin{aligned}\log f(n) &= \log\,(\log\log n)^{\log n} \\ &= \log n \log(\log\log n)\end{aligned}$$

$$\log g(n) = \log\left(\log^{\log\log n} n\right)$$
$$= \log\log n \log(\log n)$$
$$\text{Since } \frac{\log n}{(\log(\log n))^2} \to +\infty \text{ as } n \to +\infty$$
$$\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \frac{\log n \log(\log\log n)}{(\log(\log n))^2} = +\infty$$
$$\text{So, } f(n) = \omega g(n)$$

(2):

$$\text{Since } \log i \le \lceil \log i \rceil \le \log i + 1$$
$$\text{Then, } 2^{-\lceil \log i \rceil} \ge 2^{-\log i - 1} = \frac{1}{2i}$$
$$\text{And } 2^{-\lceil \log i \rceil} \le 2^{-\log i} = \frac{1}{i}$$
$$\text{So, } \frac{1}{2}\sum_{i=1}^{n} \frac{1}{i} \le \sum_{i=1}^{n} 2^{-\lceil \log i \rceil} \le \sum_{i=1}^{n} \frac{1}{i}$$
$$\text{Thus, } 1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n} = \Theta \sum_{i=1}^{n} 2^{-\lceil \log i \rceil}$$
$$\text{i.e. } f(n) = \Theta g(n)$$

(3):

$$\text{We first prove that } \log n! = \Theta(n \log n)$$
$$\text{Since } log(n!) = log(1) + log(2) + \ldots + log(n)$$
$$\text{We have by monotonicity: } log(n!) \le \sum_{k=1}^{n} log(n) = nlog(n)$$
$$\text{From the bound, we can use the fact that } log(k) \ge log(\frac{n}{2}), \forall k \ge 2$$
$$\text{Thus, we can get:}$$
$$log(n!) \ge \sum_{k=\frac{n}{2}}^{n} log(k) \ge \sum_{k=\frac{n}{2}}^{n} log(\frac{n}{2}) = \frac{n}{2} log(\frac{n}{2})$$
$$\text{Since } \log n! = \Theta(n \log n)$$
$$\text{And } \log(n^{0.99n}) = 0.99n \log n = o(n \log n)$$
$$\text{So, } n! = \omega(n^{0.99n})$$

(4):

The possible combinations are: $\emptyset, \{\Omega\}, \{O\}$

$$\text{For } \emptyset, \text{we can have: } f(n) = n, g(n) = n^{1+\sin(n)}$$

$$\text{For } \{\Omega\}, \text{we can have: } f(n) = \begin{cases} n^2, & \text{if } n \text{ is even} \\ n, & \text{if } n \text{ is odd} \end{cases}, g(n) = n$$

$$\text{For } \{O\}, \text{we can have: } f(n) = n\sin^2(n), g(n) = n$$

(5): The asymptotic relationship is empty set $\emptyset$
Since we have

$$2^{2^{\log\log n-1}} \le 2^{2^{\lfloor \log\log n\rfloor}} \le 2^{2^{\log\log n}}$$
$$2^{0.5\log n} \le 2^{2^{\lfloor \log\log n\rfloor}} \le 2^{\log n}$$
$$n^{0.5} \le 2^{2^{\lfloor \log\log n\rfloor}} \le n$$

Since $n^{0.5} \le n^{0.6} \le n$,then

$$\liminf_{n\to+\infty} \frac{f(n)}{g(n)} = 0, \limsup_{n\to+\infty} \frac{f(n)}{g(n)} = +\infty$$

So, $f(n) = \emptyset g(n)$

Problem 2: Finding the maximum/minimum $[10 + 10^\star$ pts]

For an array A of n *different* numbers (not necessarily sorted), we want to find the *largest* number and the *smallest* number in A simultaneously. However, we have no access to A. Instead, we are given an oracle Compare that can be used to compare the numbers in A. For $i \neq j$, Compare(i, j) returns i if $A[i] > A[j]$ and returns j if $A[j] > A[i]$ (recall that the numbers in A are different by assumption, so we cannot have $A[i] = A[j]$ if $i \neq j$). For convenience, let us assume n is even.

(1) Design an algorithm which calls the Compare oracle at most $\frac{3}{2}n - 2$ times and returns a pair $(i_{\mathsf{max}}, i_{\mathsf{min}})$ such that $A[i_{\mathsf{max}}]$ (resp., $A[i_{\mathsf{min}}]$) is the largest (resp., smallest) number in A. Give the pseudocode and justify its correctness.

(b)* Show that any algorithm has to call the Compare oracle $\frac{3}{2}n - 2$ times in worst case in order to find the largest and smallest numbers in A.

Solution.

Algorithm 1 Find the maximum and minimum indices in array A using at most $\frac{3n}{2} - 2$ comparisons

```
1: Initialize empty lists: CandidatesMax and CandidatesMin
2: for $i = 1$ to $n - 1$ step 2 do
3:     if Compare($i, i + 1$) = $i$ then
4:         Append $i$ to CandidatesMax
5:         Append $i + 1$ to CandidatesMin
6:     else
7:         Append $i + 1$ to CandidatesMax
8:         Append $i$ to CandidatesMin
9:     end if
10: end for
11: $i_{\mathsf{max}} \leftarrow$ CandidatesMax[0]
12: for each index $j$ in CandidatesMax do
13:     if Compare($j, i_{\mathsf{max}}$) = $j$ then
14:         $i_{\mathsf{max}} \leftarrow j$
15:     end if
16: end for
17: $i_{\mathsf{min}} \leftarrow$ CandidatesMin[0]
18: for each index $j$ in CandidatesMin do
19:     if Compare($j, i_{\mathsf{min}}$) = $i_{\mathsf{min}}$ then
20:         $i_{\mathsf{min}} \leftarrow j$
21:     end if
22: end for
Ensure: ($i_{\mathsf{max}}, i_{\mathsf{min}}$)
```

Correctness:

The core idea is to reduce the number of candidates for both maximum and minimum in a single pass. By comparing elements pairwise, we guarantee that for each comparison, one element goes to the potential maximum list and the other to the potential minimum list. This effectively halves the number of elements we need to consider in subsequent steps. The separate loops for finding the maximum and minimum among the candidates then efficiently determine the true maximum and minimum indices. The algorithm's correctness stems from the transitivity of the comparison: if a > b and b > c, then a > c. The pairwise comparisons, combined with the subsequent candidate refinement, ensure that the true maximum and minimum are identified.

(b)

Proof. Let us define two sets:

- MAX: set of elements that could still be the maximum
- MIN: set of elements that could still be the minimum

Initially, all n elements belong to both MAX and MIN. To determine the maximum and minimum, the algorithm must eliminate $n-1$ elements from MAX and $n-1$ elements from MIN, for a total of $2n-2$ eliminations.

The crucial insight is that a single comparison can eliminate at most 2 elements, one from each set. Specifically:

- When $\textsc{Compare}(i, j)$ returns i, we can eliminate j from MAX and i from MIN.
- When $\textsc{Compare}(i, j)$ returns j, we can eliminate i from MAX and j from MIN.

However, this maximum efficiency (eliminating 2 elements per comparison) is only possible when both elements being compared are in $MAX \cap MIN$. Once an element has been eliminated from one set but remains in the other, comparing it can only eliminate at most one element.

The optimal strategy involves first pairing up the elements and comparing each pair. After $\frac{n}{2}$ such comparisons:

- $\frac{n}{2}$ elements (the winners) remain in MAX but are eliminated from MIN
- $\frac{n}{2}$ elements (the losers) remain in MIN but are eliminated from MAX

At this point, we've eliminated $\frac{n}{2}$ elements from each set, with $\frac{n}{2}$ elements remaining in each. To determine the maximum, we need to eliminate $\frac{n}{2}-1$ more elements from MAX, requiring at least $\frac{n}{2}-1$ additional comparisons. Similarly, determining the minimum requires eliminating $\frac{n}{2}-1$ more elements from MIN, needing at least $\frac{n}{2}-1$ more comparisons.

Therefore, the total number of comparisons is at least:

$$\frac{n}{2} + \left(\frac{n}{2} - 1\right) + \left(\frac{n}{2} - 1\right) = \frac{3n}{2} - 2$$

□

Problem 3: Solving recurrences [$4 \times 5 + 5$ pts]

(1) Find big-Θ bounds for the following recurrences (and show your bounds are correct). For the base case, simply assume $T(n) = 1$ for all $n \leq 2$.

(i) $T(n) = 8T(n/3) + n^{1.5} \log^4 n + 9n$

(ii) $T(n) = T(n - \sqrt{n}) + 6 \log n$

(iii) $T(n) = 2T(n/2) + \frac{n}{\log n}$

(iv) $T(n) = 5\sqrt{n} \cdot T(\sqrt{n}) + 2n$

(2) Recall the Fibonacci sequence $F_0, F_1, F_2, \ldots$ defined using the recurrence $F_n = F_{n-1} + F_{n-2}$ with the base case $F_0 = 0$ and $F_1 = 1$. Prove by induction that $\phi^{n-2} \leq F_n < \phi^n$ for all $n \geq 1$, where $\phi = (1 + \sqrt{5})/2$. Based on this, further show that $F_n = \Theta(\phi^n)$.

Solution.

(1) (i) By master theorem, we have $a = 8, b = 3, f(n) = n^{1.5} \log^4 n + 9n$. Since $f(n) = O(n^{\log_3 8 - \epsilon})$ for $\epsilon = 0.5$, we have $T(n) = \Theta(n^{\log_3 8})$.

(ii) Let $m = \sqrt{n}$, then we have $T(m^2) = T(m^2 - m) + 12 \log m$.
So we have $T((m^2) = T(m - 0.5)^2) + 12 \log m - 0.25$
Let $S(m) = T(m^2)$
So we have $S(m) = S(m - 0.5) + 12 \log m - 0.25$
Since the step is 0.5, we have $S(m) = \Theta(m \log m)$
So we have $T(n) = \Theta(\sqrt{n} \log \sqrt{n}) = \Theta(\sqrt{n} \log n)$

(iii) By recursion tree, the i-th layer has the sum $\frac{n}{\log \frac{n}{i}}$ and we have $\log n - 1$ layers
So we have $T(n) = \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i}$
Let $j = \log n - i$. We have

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i} \\ &= \sum_{j=\log n}^{1} \frac{n}{j} \cdot (-1) \\ &= n \sum_{j=1}^{\log n} \frac{1}{j} \end{aligned}$$

we observe that:

$$\int_1^{\log n} \frac{1}{x}\, dx < \sum_{j=1}^{\log n} \frac{1}{j} < 1 + \int_1^{\log n} \frac{1}{x}\, dx$$

This gives us:

$$\ln(\log n) < \sum_{j=1}^{\log n} \frac{1}{j} < 1 + \ln(\log n)$$

In fact, it can be shown that as n grows large:

$$\sum_{j=1}^{\log n} \frac{1}{j} = \ln(\log n) + \gamma + O\left(\frac{1}{\log n}\right)$$

where $\gamma \approx 0.57721$ is the Euler-Mascheroni constant.

Therefore, the original sum has the asymptotic behavior:

$$\begin{aligned} T(n) &= n \sum_{j=1}^{\log n} \frac{1}{j} \\ &= n \cdot (\ln(\log n) + \gamma + o(1)) \\ &= n \ln(\log n) + \gamma n + o(n) \\ &= \Theta(n \log\log n) \end{aligned}$$

Thus, $T(n) = \Theta(n \log\log n)$.

(iv) Let $n = 2^m, S(m) = T(2^m) = T(n)$, then we have

$$S(m) = 5 \cdot 2^{m/2} \cdot S(m/2) + 2^{m+1}$$

Divide both sides by 2^m, we have

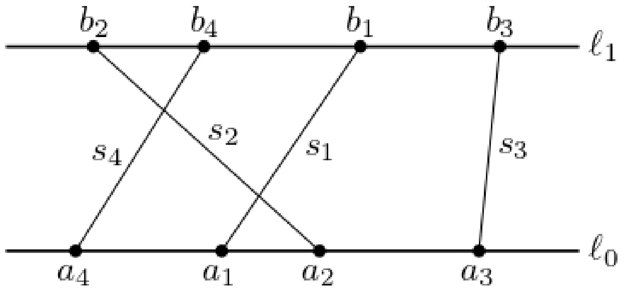
$$\begin{aligned} \frac{S(m)}{2^m} &= 5 \cdot \frac{S(m/2)}{2^{-m/2}} + 2 \\ \text{Let } P(m) &= \frac{S(m)}{2^m} \\ P(m) &= 5 \cdot P(m/2) + 2 \\ \text{By master theorem, we have } a &= 5, b = 2, f(m) = 2 \\ \text{Since } f(m) &= O(m^{\log_2 5 - \epsilon}) \text{ for } \epsilon = 1 \\ \text{we have } P(m) &= \Theta(m^{\log_2 5}) \\ \text{So, } S(m) &= \Theta(2^m \cdot m^{\log_2 5}) \\ \text{Thus, } T(n) &= \Theta(n \cdot (\log n)^{\log_2 5}) \end{aligned}$$

(2) When $n = 1$, we have $F_n = 1$ and $\phi^{n-2} = \frac{2}{1+\sqrt{5}}$.
So we have $\phi^{n-2} \le F_n < \phi^n$ for $n = 1$.
Assume that $\phi^{n-2} \le F_n < \phi^n$ for $n = k$,
then we have $F_{k+1} = F_k + F_{k-1} < \phi^k + \phi^{k-1} = \phi^{k-1}(\phi + 1) = \phi^{k-1}\phi^2 = \phi^{k+1}$
And $F_{k+1} = F_k + F_{k-1} > \phi^{k-2} + \phi^{k-1} = \phi^{k-2}(\phi + 1) = \phi^{k-2}\phi^2 = \phi^k$ So we have $\phi^{n-2} \le F_n < \phi^n$ for all $n \ge 1$
Since $\phi^n \cdot \underbrace{\phi^{-2}}_{constant} \le F_n < \phi^n$ for all $n \ge 1$, we have $F_n = \Theta(\phi^n)$

Problem 4: Counting intersection points [10 pts]

Consider two horizontal lines $\ell_0 : y = 0$ and $\ell_1 : y = 1$ in the plane. We have n distinct points $a_1, \ldots, a_n$ on ℓ_0 and n distinct points $b_1, \ldots, b_n$ on ℓ_1. Note that the points $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ are *not* necessarily sorted by their x-coordinates. Now for each $i \in \{1, \ldots, n\}$, we draw a segment s_i connecting a_i and b_i. See the figure below for an example of $n = 4$. These segments $s_1, \ldots, s_n$ may intersect with each other, and for simplicity, we assume no three segments intersect at the same point. We aim to count the number of intersection points of $s_1, \ldots, s_n$.

Formally, design an algorithm CountInt(n, A, B) where $A[1 \ldots n]$ stores the x-coordinates of $a_1, \ldots, a_n$ and $B[1 \ldots n]$ stores the x-coordinates of $b_1, \ldots, b_n$. The algorithm should return the number of intersection points of the segments $s_1, \ldots, s_n$, and should have time complexity $O(n \log n)$. Describe the basic idea and give the pseudocode. Then, justify its correctness and show it runs in $O(n \log n)$ time.



Solution. Observing that for every points on l_1 and l_0 , we will have one intersection point when comparing k-th points (from left to right) on l_1 and l_0 and have i>j for a_i and b_j because under this condition, $b_i > b_j$ and $a_i < a_j$
So the idea is to merge sort on $a_1, a_2, \cdots, a_n$ and $b_1, b_2, \cdots, b_n$ to get the sorted position from them, then compare each index with corresponding postion's index of $b_1, b_2, \cdots, b_n$ to get the number of intersection points.

Algorithm 2 Count Intersection Points

```
1: **Function** CountInt($n, A, B$)
2: Create pairs $P_A[1...n]$ where $P_A[i] = (A[i], i)$ for $i = 1$ to $n$
3: Create pairs $P_B[1...n]$ where $P_B[i] = (B[i], i)$ for $i = 1$ to $n$
4: MergeSort($P_A, 1, n$) {Sort by first element}
5: MergeSort($P_B, 1, n$) {Sort by first element}
6: Create arrays $I[1...n]$ and $J[1...n]$
7: for $i = 1$ to $n$ do
8:     $I[i] \leftarrow$ second element of $P_A[i]$
9:     $J[i] \leftarrow$ second element of $P_B[i]$
10: end for
11: $count \leftarrow 0$
12: for $i = 1$ to $n$ do
13:     if $J[i] > I[i]$ then
14:         $count \leftarrow count + 1$
15:     end if
16: end for
17: return $count$
```

Algorithm 3 Merge Sort

```
1: Function MergeSort($P, left, right$)
2: if $left < right$ then
3:     $mid \leftarrow \lfloor (left + right)/2 \rfloor$
4:     MergeSort($P, left, mid$)
5:     MergeSort($P, mid + 1, right$)
6:     Merge($P, left, mid, right$)
7: end if
```

Algorithm 4 Merge

```
1: Function MERGE($P, left, mid, right$)
2: Create temporary arrays $L[1...(mid - left + 1)]$ and $R[1...(right - mid)]$
3: Copy $P[left...mid]$ to $L$ and $P[mid + 1...right]$ to $R$
4: $i \leftarrow 1, j \leftarrow 1, k \leftarrow left$
5: while $i \leq (mid - left + 1)$ AND $j \leq (right - mid)$ do
6:     if $L[i].first \leq R[j].first$ then
7:         $P[k] \leftarrow L[i]$
8:         $i \leftarrow i + 1$
9:     else
10:        $P[k] \leftarrow R[j]$
11:        $j \leftarrow j + 1$
12:    end if
13:    $k \leftarrow k + 1$
14: end while
15: while $i \leq (mid - left + 1)$ do
16:    $P[k] \leftarrow L[i]$
17:    $i \leftarrow i + 1$
18:    $k \leftarrow k + 1$
19: end while
20: while $j \leq (right - mid)$ do
21:    $P[k] \leftarrow R[j]$
22:    $j \leftarrow j + 1$
23:    $k \leftarrow k + 1$
24: end while
```

Correctness: The algorithm sorts the points on l_0 and l_1 by their x-coordinates. For every points on l_1 and l_0 , we will have one intersection point when comparing k-th points (from left to right) on l_1 and l_0 and have i>j for a_i and b_j because under this condition, $b_i > b_j$ and $a_i < a_j$

Complexity: The algorithm sorts the points on l_0 and l_1 by their x-coordinates in $O(n \log n)$ time. Then, it compares the indices of the sorted points in $O(n)$ time. Thus, the total time complexity is $O(n \log n)$.

Problem 5: 3SUM from 3 arrays [8 pts]

Given 3 non-empty and sorted arrays A, B, C of integers, each array is of size N_A, N_B and N_C respectively. Given an input a, output one value from each array such that the sum is equal to a. If not found, return 0. Design the algorithm, prove its correctness and complexity.

Solution.

Algorithm 5 3SUM from 3 arrays

```
1:  Function FindTriplet$A, B, C, a$
2:  $NA \leftarrow$ length$(A)$
3:  $NB \leftarrow$ length$(B)$
4:  $NC \leftarrow$ length$(C)$
5:  for $i = 1$ to $NA$ do
6:      $target \leftarrow a - A[i]$
7:      $j \leftarrow 1$
8:      $k \leftarrow NC$
9:      while $j \leq NB$ and $k \geq 1$ do
10:         $s \leftarrow B[j] + C[k]$
11:         if $s = target$ then
12:             return $(A[i], B[j], C[k])$
13:         else if $s < target$ then
14:             $j \leftarrow j + 1$
15:         else
16:             $k \leftarrow k - 1$
17:         end if
18:     end while
19: end for
20: return 0
```

Correctness:

The algorithm is correct because it systematically fixes each candidate from array A and uses a correctly implemented two-pointer method on two sorted arrays B and C to find complementary pairs that sum to the required value. This ensures that, if a valid triplet exists, it will be found.

Complexity:

The algorithm iterates through the elements of array A, and for each element, it uses a two-pointer method to find a pair in arrays B and C that sum to the target value. The two-pointer method takes $O(N_B + N_C)$ time, and since the algorithm iterates through all elements of array A, the total time complexity is $O(N_A(N_B + N_C))$.

Problem 6: Counting context-free grammar words [7 pts]

Context-free grammar (CFG) is an important concept in CS, and it recursively defines a set of strings/words. You may want to search for CFG and learn about the relevant context. In this problem, we omit many formal definitions and focus on the following grammar:

(1) $S \to D|D + S$

(2) $D \to 1D|2D|3D|4D|\dots|9D|0|1|2|3|\dots|9$

The above two rules read as follows: we always start with S, and S can either get turned into D or $D + S$. Here, + is a terminal symbol and cannot be transformed into other symbols. For the second rule, D can get transformed into $1D$ or $2D$ or $\dots$, and all the numbers are terminal symbols. The "|" essentially means "or" for the transformation. A word satisfying the above CFG is a string of terminal symbols that can be generated using the above rules.

Think about what the above grammar represents. Count the number of words satisfying the above CFG of a given length n using recursion.

Solution.

Algorithm 6 f(n):Counting context-free grammar words

```
1: Function: g(n)
2: if $n = 1$ then
3:     return 10
4: else
5:     return $9 \times g(n-1)$
6: end if
7: Function: f(n)
8: if $n = 1$ then
9:     return g(1)
10: end if
11: result ← g(n) {Case 1: $S \to D$.}
12: for $i \leftarrow 1$ to $n-2$ do
13:     result ← result + g(i) × f(n-i-1) {Case 2: $S \to D + S$ }
14: end for
15: return result
```

Problem 7: Search for a root [10 pts]

Given a continuous function $f : [1, 2] \times [1, 2] \to [0, 1]$ with the "bi-monotone" property: $f(x, y) > f(x', y)$ if $x > x'$ and $f(x, y) > f(x, y')$ if $y > y'$, we want to solve for the root of $f(x, y) = a$ where a is some given value. The only access to f is an oracle such that given (x, y), the oracle returns the value of $f(x, y)$.

Let R be the set of roots, i.e., $f(x^*, y^*) = a$ for any $(x^*, y^*) \in R$. We want to find an approximate solution $(\hat{x}, \hat{y})$ such that there exists $(x^*, y^*) \in R$ with $\hat{x} \le x^* \le (1 + \epsilon)\hat{x}$ and $\hat{y} \le y^* \le (1 + \epsilon)\hat{y}$ with as few oracle calls as possible. Here, $\epsilon > 0$ is another given parameter indicating the desirable level of accuracy. If there is no root, return 0.

Design an algorithm FINDROOT(f, a, ϵ), and prove its correctness and complexity (in terms of number of oracle calls). *This problem is a little bit more involved, but it does not require anything beyond the lectures. Suggest working on this after you finish everything else.*

Solution.

Algorithm 7 FindRoot: Approximate a solution (x, y) for $f(x, y) = a$

```
Function: FindRoot(f, a, $\epsilon$)
if $f(1,1) > a$ or $f(2,2) < a$ then
    return 0 {No solution exists in $[1,2] \times [1,2]$}
end if
$x_{\text{low}} \leftarrow 1,\ x_{\text{high}} \leftarrow 2$
while $x_{\text{high}}/x_{\text{low}} > 1 + \epsilon$ do
    $x_{\text{mid}} \leftarrow \sqrt{x_{\text{low}} \times x_{\text{high}}}$
    if $a < f(x_{\text{mid}}, 1)$ then
        $x_{\text{high}} \leftarrow x_{\text{mid}}$
    else if $a > f(x_{\text{mid}}, 2)$ then
        $x_{\text{low}} \leftarrow x_{\text{mid}}$
    else
        $x_{\text{high}} \leftarrow x_{\text{mid}}$
    end if
end while
$\hat{x} \leftarrow x_{\text{low}}$
$y_{\text{low}} \leftarrow 1,\ y_{\text{high}} \leftarrow 2$
while $y_{\text{high}}/y_{\text{low}} > 1 + \epsilon$ do
    $y_{\text{mid}} \leftarrow \sqrt{y_{\text{low}} \times y_{\text{high}}}$
    if $f(\hat{x}, y_{\text{mid}}) < a$ then
        $y_{\text{low}} \leftarrow y_{\text{mid}}$
    else
        $y_{\text{high}} \leftarrow y_{\text{mid}}$
    end if
end while
$\hat{y} \leftarrow y_{\text{low}}$
return $(\hat{x}, \hat{y})$
```

Correctness:

The algorithm is correct because it uses the strict monotonicity of $f(x, y)$ in both variables to maintain intervals that always contain a true root. By performing a multiplicative binary search, the algorithm narrows the intervals until the relative error for both the x and y coordinates is at most $1+\epsilon$. This guarantees that there exists a true solution (x^*, y^*) with

$$X_{\text{approx}} \leq x^* \leq (1+\epsilon)X_{\text{approx}} \quad \text{and} \quad Y_{\text{approx}} \leq y^* \leq (1+\epsilon)Y_{\text{approx}},$$

meeting the required approximation condition.

Complexity:

Each binary search (for both x and y) reduces the interval multiplicatively, starting from $[1, 2]$ (a factor of 2) and ending when the interval ratio is at most $1 + \epsilon$. This requires $O\big(\log_{1+\epsilon}(2)\big)$ iterations, which is approximately $O(1/\epsilon)$ for small ϵ. Since each iteration makes a constant number of oracle calls, the total number of oracle calls is $O(1/\epsilon)$.