COMP 6651 Algorithm Design Techniques

Assignment 1 Answers

Name : Parsa Kamalipour , *StudentID* : 40310734

Exercise 1: (20 marks)

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

b) $f(n) = O((f(n))^2)$.

c) $f(n) = \Theta(f(n/2))$.

d) $f(n) + o(f(n)) = (f(n))$.

Answers to Exercise 1

a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.
This statement is false. The fact to say that $f(n) = O(g(n))$ means that $f(n)$ does not grows asymptotically faster than $g(n)$, but this does not implies that $g(n)$ also grows no faster than $f(n)$.

Counterexample:
Let $f(n) = n$ and $g(n) = n^2$. Then we have:

- $F(n) = O(g(n))$, since $n = O(n^2)$ is basically $n \leq C.n^2$ for some constant $C > 0$ and $n > n_0$.
- However, $g(n) \neq O(f(n))$, because n^2 grows faster than n, and there is no constant C such that $n^2 \leq C.n$ for some $n > n_0$.

Thus, the implication does not hold in general, and the conjection is **FALSE!**

b) $f(n) = O((f(n))^2)$.
Prove: This statement is true for non-descending, asymptotically positive functions.

Proof:
Lets take any asymptotically positive function $f(n)$ with the key idea that if $f(n)$ is a non-decreasing function, then $f(n) \leq f(n)^2$ for $n > n_0$:

- For example, consider $f(n) = n$, we have $n = O(n^2)$, because $n \leq C.n^2$ for some constant C when $n > n_0$.

This holds for any $n > n_0$ with asymptotically positive function, that $f(n) = O((f(n))^2)$ is **TRUE!**

c) $f(n) = \Theta(f(n/2))$.
Disprove:
This statement is false in general. While some functions might satisfy this property, not all functions do.

Counterexample:
Let $f(n) = 4^n$, We will have:

- $f(n) = 4^n$
- $f(n/2) = 4^{n/2} \rightarrow f(n/2) = 2^n$

By the different results of these two we can see that the growth rates are different, which does not satisfy the $\Theta(f(n))$. And the reason for it is that 4^n and 2^n has two separate order of growth, so they are not asymptotically the same.
Thus, the conjecture is **FALSE!**.

d) $f(n) + o(f(n)) = \Theta(f(n))$.
Prove: This statement is true.

Proof:
Let's assume $g(n) = o(f(n))$.
Then, there exists positive constant C and n such that $n > n_0$:

$$0 \leq g(n) < C.f(n)$$

$$f(n) \leq f(n) + g(n) \leq f(n) + C.f(n) \quad (add\ f(n)\ to\ every\ sides)$$

$$f(n) \leq f(n) + o(f(n)) \leq (1 + C).f(n) \quad (replace\ g(n))$$

Therefore, $f(n) + o(f(n))$ is bounded by $f(n)$ on both sides with some constant which makes it equal to $\Theta(f(n))$.

For example, if $f(n) = n$ and $o(f(n)) = \log n$, then $n + \log n$ still behaves asymptotically like n , meaning it is $\Theta(n)$. Thus, the conjecture is **TRUE!**.

Exercise 2: (10 marks)

The solution to the recurrence $T(n) = 4T(n/2) + n$ turns out to be $T(n) = \Theta(n^2)$.

a) Show that a substitution proof with the assumption $T(n) \leq cn^2$ fails.

b) Then show how to subtract a lower-order term to make a substitution proof work.

Answers to Exercise 2

a) Show that a substitution proof with the assumption $T(n) \leq cn^2$ fails.
Let's assume $T(n) \leq Cn^2$ where C and n_0 are positive constants and $n_0 \leq n$.
We have $T(n) = 4T(n/2) + n$ and based on this formula lets plug-in the $n/2$ to our assumption:

$$T(n/2) \leq C.(n/2)^2$$

$$T(n/2) \leq Cn^2/4$$

Now lets plug-in this T(n/2) to $T(n) = 4T(n/2) + n$:

$$T(n) \leq 4 * (C * n^2/4) + n$$

$$T(n) \leq Cn^2 + n$$

But this inequality includes an additional term $+n$, which is not accounted for in the assumption $T(n) \leq cn^2$
This suggests that the assumption $T(n) \leq cn^2$ does not hold because the lower-order term n will eventually dominate, making the assumption $T(n) \leq cn^2$ insufficient.
Therefore, **the proof fails due to the additional +n term.**

b) Then show how to subtract a lower-order term to make a substitution proof work.
To make the substitution work, we need to introduce a slightly tighter bound by subtracting a lower-order term, say ϵn^2 , to account for the additional $+n$.
Now lets assume that $T(n) \leq Cn^2 - \epsilon n$ for all $n_0 \leq n$, where C, ϵ, *and* n_0 are positive. Lets plug-in $n/2$ to our assumption:

$$T(n) = 4T(n/2) + n$$

$$T(n/2) \leq C(n/2)^2 - (\epsilon n)/2$$

Now lets plug this $T(n/2)$ to the main $T(n)$:

$$T(n) \leq 4(C\frac{n^2}{4} - \frac{\epsilon n}{2}) + n$$

$$T(n) \leq cn^2 - 2\epsilon n + n$$

$$T(n) \leq cn^2 - \epsilon n - (\epsilon - 1)n$$

As long as $(\epsilon - 1)n$ is positive and we set our base case as $n_0 = 1$:
(because our hypothesis works for any $\epsilon \geq 1$)

$$T(n) \leq cn^2 - \epsilon n$$

Exercise 3: (10 marks)

For each of the following recurrences, sketch its recursion tree, and guess a good asymptotic upper bound on its solution. Then use the substitution method to verify your answer.

a) $T(n) = T(n/2) + n^3$

b) $T(n) = 3T(n-1) + 1$

Answers to Exercise 3

Problem (a): $T(n) = T(n/2) + n^3$

The recursion tree for $T(n) = T(n/2) + n^3$ is as follows:

$$\begin{array}{c} n^3 \\ \downarrow \\ T(n/2) + (n/2)^3 \\ \downarrow \\ T(n/4) + (n/4)^3 + (n/2)^3 \\ \downarrow \\ T(n/8) + (n/8)^3 + (n/4)^3 + (n/2)^3 \\ \vdots \end{array}$$

The depth of the tree is $\log n$, and at level i, the total cost is:

$$2^i \times \left(\frac{n}{2^i}\right)^3 = \frac{n^3}{2^{2i}}.$$

Summing the cost over all levels gives:

$$T(n) = n^3 \left(1 + \frac{1}{2^2} + \frac{1}{4^2} + \cdots\right),$$

which converges to $O(n^3)$.
Thus, the solution is:

$$T(n) = O(n^3).$$

Substitution Method:
We guess that $T(n) = O(n^3)$, and use induction to prove it.
Base Case: For small values of n, such as $n = 1$, the recurrence becomes:

$$T(1) = O(1^3) = O(1).$$

Clearly, this holds.
Inductive Hypothesis: Assume that for some $k \leq n$, the recurrence holds:

$$T(k) \leq ck^3,$$

for some constant $c > 0$.
Inductive Step: We need to prove that $T(n) \leq cn^3$ holds for n. Using the recurrence relation:

$$T(n) = T(n/2) + n^3.$$

By the inductive hypothesis, we can substitute $T(n/2)$:

$$T(n) \leq c(n/2)^3 + n^3.$$

Simplifying:

$$T(n) \leq c\frac{n^3}{8} + n^3.$$

Factoring out n^3:

$$T(n) \leq n^3 \left(\frac{c}{8} + 1\right).$$

For sufficiently large c, the inequality holds, confirming that:

$$T(n) = O(n^3).$$

Problem (b): $T(n) = 3T(n-1) + 1$:
The recursion tree for $T(n) = 3T(n-1) + 1$ is as follows:

$$\begin{array}{c} 1 \\ \downarrow \\ 3 \times (1+1+1) \\ \downarrow \\ 3^2 \times (1+1+1+1+1+1+1+1+1) \\ \downarrow \\ 3^3 \times \dots \\ \vdots \end{array}$$

At level i, there are 3^i subproblems, and the total cost at each level is:

$$T(n) = 1 + 3 + 3^2 + \cdots + 3^{n-1}.$$

This is a geometric series that sums to $O(3^n)$.
Thus, the solution is:

$$T(n) = O(3^n).$$

Substitution Method:
We guess that $T(n) = O(3^n)$, and use induction to prove it.
Base Case: For small values of n, such as $n = 1$, the recurrence becomes:

$$T(1) = O(3^1) = O(3).$$

Clearly, this holds.
Inductive Hypothesis: Assume that for some $k \leq n$, the recurrence holds:

$$T(k) \leq c3^k,$$

for some constant $c > 0$.
Inductive Step: We need to prove that $T(n) \leq c3^n$ holds for n. Using the recurrence relation:

$$T(n) = 3T(n-1) + 1.$$

By the inductive hypothesis, we can substitute $T(n-1)$:

$$T(n) \leq 3 \times c3^{n-1} + 1.$$

Simplifying:

$$T(n) \leq c3^n + 1.$$

For sufficiently large n, the term $+1$ is negligible, so:

$$T(n) \leq c3^n.$$

Thus, by induction, the solution is:

$$T(n) = O(3^n).$$

Exercise 4: (10 marks)

Use the Akra-Bazzi method to solve the following recurrences.

a) $T(n) = T(n/2) + T(n/3) + T(n/6) + nlgn$

b) $T(n) = (1/3)T(n/3) + 1/n$

Answers to Exercise 4

To solve these recurrences using the Akra-Bazzi method, we will first identify the parameters a_i, b_i, and $f(n)$ that fit the standard form of the Akra-Bazzi theorem:

$$T(n) = \sum_{i=1}^{k} a_i\, T(b_i n) + f(n),$$

where $a_i > 0$, $0 < b_i < 1$, and $f(n)$ is asymptotically positive.

Part (a): $T(n) = T(n/2) + T(n/3) + T(n/6) + n \lg n$

Step 1: Identify Parameters

- Number of recursive calls: $k = 3$
- Coefficients: $a_1 = a_2 = a_3 = 1$
- Fractions of size: $b_1 = \dfrac{1}{2}, \quad b_2 = \dfrac{1}{3}, \quad b_3 = \dfrac{1}{6}$
- Non-recursive part: $f(n) = n \lg n$

Step 2: Solve for p_0

Find p_0 such that:

$$\sum_{i=1}^{k} a_i\, b_i^{p_0} = 1$$

Substitute the values:

$$\left(\frac{1}{2}\right)^{p_0} + \left(\frac{1}{3}\right)^{p_0} + \left(\frac{1}{6}\right)^{p_0} = 1$$

Simplify:

$$2^{-p_0} + 3^{-p_0} + 6^{-p_0} = 1$$

Test $p_0 = 1$:

$$2^{-1} + 3^{-1} + 6^{-1} = \frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1$$

Therefore, $p_0 = 1$.

Step 3: Compute the Integral

Calculate:

$$\int_1^n \frac{f(u)}{u^{p_0+1}}\,du = \int_1^n \frac{u \lg u}{u^2}\,du = \int_1^n \frac{\lg u}{u}\,du$$

This integral evaluates to:

$$\int_1^n \frac{\lg u}{u}\,du = \frac{1}{2}(\lg n)^2$$

Step 4: Determine $T(n)$

Using the Akra-Bazzi theorem:

$$T(n) = \Theta\left(n^{p_0}\left(1 + \int_1^n \frac{f(u)}{u^{p_0+1}}\,du\right)\right)$$

Substitute $p_0 = 1$:

$$T(n) = \Theta\left(n\left(1 + \frac{1}{2}(\lg n)^2\right)\right) = \Theta\left(n(\lg n)^2\right)$$

Final Answer for Part (a)

$$T(n) = \Theta\left(n\,(\log n)^2\right)$$

Part (b): $T(n) = \frac{1}{3}T\left(\frac{n}{3}\right) + \frac{1}{n}$

Step 1: Identify Parameters

- Number of recursive calls: $k = 1$
- Coefficient: $a_1 = \frac{1}{3}$
- Fraction of size: $b_1 = \frac{1}{3}$
- Non-recursive part: $f(n) = \frac{1}{n}$

Step 2: Solve for p_0

Find p_0 such that:

$$a_1\, b_1^{p_0} = 1$$

Substitute the values:

$$\frac{1}{3}\left(\frac{1}{3}\right)^{p_0} = 1 \implies \left(\frac{1}{3}\right)^{1+p_0} = 1$$

Solve for p_0:

$$1 + p_0 = 0 \implies p_0 = -1$$

Step 3: Compute the Integral

Calculate:

$$\int_1^n \frac{f(u)}{u^{p_0+1}}\,du = \int_1^n \frac{\frac{1}{u}}{u^0}\,du = \int_1^n \frac{1}{u}\,du = \ln n$$

Step 4: Determine $T(n)$

Using the Akra-Bazzi theorem:

$$T(n) = \Theta\left(n^{p_0}\left(1 + \int_1^n \frac{f(u)}{u^{p_0+1}}\,du\right)\right)$$

Substitute $p_0 = -1$:

$$T(n) = \Theta\left(n^{-1}\left(1 + \ln n\right)\right) = \Theta\left(\frac{\ln n}{n}\right)$$

Final Answer for Part (b)

$$T(n) = \Theta\left(\frac{\log n}{n}\right)$$

Summary of Final Answers

(a) $T(n) = \Theta\left(n\,(\log n)^2\right)$

(b) $T(n) = \Theta\left(\frac{\log n}{n}\right)$

Exercise 5: (20 marks)

Suppose you are consulting for a bank that is concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards. Each bank card is a small plastic object, containing a smart chip with some encrypted data, and it corresponds to a unique account in the bank. Each bank can have many bank cards corresponding to it, and we will say that two bank cards are equivalent if they correspond to the same account.

It is very difficult to read the account number off a bank card directly, but the bank has a high- tech equivalence tester that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Their question is the following: among the collection of n cards, is there a set of more than n/2 of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them into the equivalence tester.

a) Give an algorithm to decide the answer to their question with only $O(nlogn)$ invocations of the equivalence tester.

b) Prove the correctness of your algorithm.

c) Using a recurrence relation to define the worst-case running time in terms of the number of invocations of the equivalence tester, and prove the running time is $O(nlogn)$.

Answers to Exercise 5

(a) **Algorithm:**

We will use a *divide-and-conquer* algorithm to determine whether there is a majority account among the n bank cards. The algorithm works recursively by splitting the set of cards into halves, finding majority candidates in each half, and then combining the results.

Steps of the Algorithm:

(a) **Base Case:** If there is only one card, return that card as the majority candidate.

(b) **Recursive Case:**

i. **Divide:** Split the set of cards into two halves.

ii. **Conquer:** Recursively find the majority candidate in each half.

iii. **Combine:**

A. Use the equivalence tester to compare the two candidates from each half.

B. **If** they are equivalent, return that candidate.

C. **Else:**

D. Count the occurrences of each candidate in the combined set using the equivalence tester.

E. **If** either candidate occurs more than $n/2$ times, return that candidate.

F. **Else**, return `None`, indicating no majority.

This algorithm ensures that at each level of recursion, we perform $O(n)$ equivalence tests, leading to a total of $O(n \log n)$ equivalence tests.

(b) **Proof of Correctness:**

We will prove the correctness of the algorithm using *mathematical induction* on the number of cards n.

Base Case ($n = 1$):

- When there is only one card, it is trivially the majority since $1 > n/2$ when $n = 1$.
- The algorithm correctly returns this card as the majority candidate.

Inductive Step:

Assume that the algorithm correctly identifies a majority among any set of $k < n$ cards.

For a set of n cards:

(a) **Divide:** The set is split into two halves of sizes n_1 and n_2, where $n_1 + n_2 = n$.

(b) **Conquer:** By the induction hypothesis, the algorithm correctly identifies the majority candidate in each half.

(c) **Combine:**

i. **Same Candidate:**

- If both halves return the same candidate, that candidate is a potential majority in the combined set.

ii. **Different Candidates:**

- If the candidates are different, we count the occurrences of each in the combined set using the equivalence tester.
- If one occurs more than $n/2$ times, it is the majority.
- If neither occurs more than $n/2$ times, there is no majority.

Since all possibilities are covered and the counts are accurate due to the equivalence tester, the algorithm correctly determines whether a majority exists.

(c) **Running Time Analysis:**

Let $T(n)$ denote the worst-case number of equivalence tester invocations for n cards.

The recurrence relation for the algorithm is:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

- $2T\left(\frac{n}{2}\right)$: The two recursive calls on the halves.
- cn: The time to combine results, which includes:
  - Comparing the two candidates ($O(1)$ test).
  - Counting occurrences of each candidate in the combined set ($O(n)$ tests).

Using the *Master Theorem* for divide-and-conquer recurrences:

- $a = 2$, $b = 2$, and $f(n) = cn = O(n)$.
- $\log_b a = \log_2 2 = 1$.
- Since $f(n) = O(n^{\log_b a} \cdot \log^k n)$ with $k = 0$,
- The recurrence solves to $T(n) = O(n \log n)$.

Conclusion:

The algorithm runs in $O(n \log n)$ time, with $O(n \log n)$ invocations of the equivalence tester, satisfying the requirements of the problem.

Exercise 6: (15 marks)

Professors Howard, Fine, and Howard have proposed a deceptively simple sorting algorithm, named stooge sort in their honor, appearing on the following page.

a) Argue that the call **STOOGE-SORT**$(A, 1, n)$ correctly sorts the array $A[1..n]$.

b) Give a recurrence for the worst-case running time of **STOOGE-SORT** and a tight asymptotic (θ − *notation*) bound on the worst-case running time.

c) Compare the worst-case running time of **STOOGE-SORT** with that of insertion sort, merge sort, and quicksort. Do the professors deserve tenure?

STOOGE-SORT(A, p, r)
if $A[p] > A[r]$ **then**
    exchange $A[p]$ with $A[r]$
if $p + 1 < r$ **then**
    $k \leftarrow \lfloor (r - p + 1)/3 \rfloor$ `// round down`
    STOOGE-SORT$(A, p, r - k)$ `// first two-thirds`
    STOOGE-SORT$(A, p + k, r)$ `// last two-thirds`
    STOOGE-SORT$(A, p, r - k)$ `// first two-thirds again`

Answers to Exercise 6

Answer to (a):

We can prove by induction that the call **STOOGE-SORT**$(A, 1, n)$ correctly sorts the array $A[1..n]$.
Base Cases:

- If $n = 1$, the array consists of a single element, which is trivially sorted.

- If $n = 2$, the algorithm compares $A[1]$ and $A[2]$ and swaps them if necessary to ensure $A[1] \leq A[2]$, thus sorting the array.

Inductive Step:
Assume that **STOOGE-SORT** correctly sorts any array of length less than n. For an array of length $n \geq 3$, the algorithm performs the following steps:

1. **Swap Elements if Necessary:**
   - It checks if $A[p] > A[r]$ and swaps them to ensure $A[p] \leq A[r]$.
2. **Recursive Calls:**
   - It computes $k = \left\lfloor \frac{n}{3} \right\rfloor$.
   - It recursively sorts the first two-thirds of the array: **STOOGE-SORT**$(A, p, r - k)$.
   - It recursively sorts the last two-thirds of the array: **STOOGE-SORT**$(A, p + k, r)$.
   - It recursively sorts the first two-thirds again: **STOOGE-SORT**$(A, p, r - k)$.

Explanation:

- After the first call, the subarray $A[p, .., r - k]$ is sorted.
- After the second call, the subarray $A[p + k, .., r]$ is sorted.
- The overlap between these two subarrays ensures that elements are compared and ordered correctly across the entire array.
- The final call re-sorts $A[p, .., r - k]$ to correct any disorder introduced by the second call.
- By the inductive hypothesis, each recursive call correctly sorts its respective subarray.

Therefore, after these steps, the entire subarray $A[p, .., r]$ is sorted, proving that **STOOGE-SORT** correctly sorts $A[1..n]$.

Answer to (b):
Let $T(n)$ denote the worst-case running time of **STOOGE-SORT** on an array of size n.
Recurrence Relation:

- For $n \leq 2$, $T(n) = \Theta(1)$.
- For $n > 2$, the algorithm makes three recursive calls on subarrays of size approximately $\frac{2n}{3}$ (since $k = \left\lfloor \frac{n}{3} \right\rfloor$, so $r - k \approx \frac{2n}{3}$).

So the recurrence relation is:

$$T(n) = 3 \cdot T\left(\frac{2n}{3}\right) + \Theta(1)$$

Solution to the Recurrence:
We can apply the Master Theorem. The recurrence is of the form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

where:

- $a = 3$
- $b = \frac{3}{2}$ (since $\frac{n}{b} = \frac{2n}{3}$)

- $f(n) = \Theta(1)$

  Compute p such that $a\left(\frac{1}{b}\right)^p = 1$:

$$\begin{aligned} 3\left(\frac{2}{3}\right)^p &= 1 \\ \left(\frac{2}{3}\right)^p &= \frac{1}{3} \\ \ln\left(\left(\frac{2}{3}\right)^p\right) &= \ln\left(\frac{1}{3}\right) \\ p\ln\left(\frac{2}{3}\right) &= \ln\left(\frac{1}{3}\right) \\ p &= \frac{\ln\left(\frac{1}{3}\right)}{\ln\left(\frac{2}{3}\right)} \end{aligned}$$

  Calculating the logarithms:

$$\begin{aligned} p &= \frac{-\ln 3}{\ln\left(\frac{2}{3}\right)} \\ &= \frac{-\ln 3}{\ln 2 - \ln 3} \\ &= \frac{-1.0986}{0.6931 - 1.0986} \\ &= \frac{-1.0986}{-0.4055} \\ &\approx 2.7095 \end{aligned}$$

  Asymptotic Bound:
  By the Master Theorem, since $f(n) = \Theta\left(n^{\log_b a - \varepsilon}\right)$ for some $\varepsilon > 0$, the solution is:
  $T(n) = \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{2.7095}\right)$
  Thus, the tight asymptotic bound on the worst-case running time is:
  $T(n) = \Theta\left(n^{\log_{(3/2)} 3}\right)$

  Answer to (c):
  Comparison of Worst-Case Running Times:

- **Stooge Sort:** $\Theta\left(n^{2.7095}\right)$
- **Insertion Sort:** $\Theta(n^2)$
- **Merge Sort:** $\Theta(n \log n)$
- **Quicksort:**
  - **Average Case:** $\Theta(n \log n)$
  - **Worst Case:** $\Theta(n^2)$

  Evaluation:

- **Stooge Sort** has a worse worst-case running time than **Insertion Sort**, which itself is less efficient than **Merge Sort** and the average case of **Quicksort**.
- Stooge Sorts time complexity is significantly higher, making it impractical for sorting large datasets.

Exercise 7: (15 marks)

You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values - so there are $2n$ values total - and you may assume that no two values are the same. You would like to determine the median of this set of $2n$ values, which we will define to be the n^{th} smallest value.

However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the databases and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Given an algorithm that finds the median value using at most $O(\log n)$ queries.

Answers to Exercise 7

To find the median of two databases, each containing n unique numerical values, we can use a binary search algorithm that performs $O(\log n)$ queries.

Algorithm Steps

1. **Initialize Variables:**
   - Set $low = 0$.
   - Set $high = n$.
2. **Begin Binary Search Loop:**
   - **While** $low \leq high$, perform steps 3 to 6.
3. **Calculate Indices:**
   - $i = \left\lfloor \dfrac{low + high}{2} \right\rfloor$.
   - $j = n - i$.
4. **Query Elements from Databases:**
   - If $i > 0$, set A_{i-1} = the $(i-1)$-th smallest element in Database A; else, $A_{i-1} = -\infty$.
   - If $i < n$, set A_i = the i-th smallest element in Database A; else, $A_i = +\infty$.
   - If $j > 0$, set B_{j-1} = the $(j-1)$-th smallest element in Database B; else, $B_{j-1} = -\infty$.
   - If $j < n$, set B_j = the j-th smallest element in Database B; else, $B_j = +\infty$.
5. **Check Conditions:**
   - **If** $A_{i-1} \leq B_j$ **and** $B_{j-1} \leq A_i$:
     - The median is $\max(A_{i-1}, B_{j-1})$.
     - **Terminate** the algorithm.
   - **Else if** $A_{i-1} > B_j$:
     - Set $high = i - 1$.
   - **Else**:
     - Set $low = i + 1$.
6. **Repeat** steps 2 to 5 until the median is found.

Explanation

- **Binary Search Mechanism:** The algorithm uses binary search to partition the combined datasets. By adjusting the indices i and j, it ensures that the left partition contains the n smallest elements.
- **Queries:** At each iteration, the algorithm queries up to four elements, handling edge cases by assigning $+\infty$ or $-\infty$ appropriately.
- **Conditions for Median:** The algorithm checks if the largest element on the left partition of A (A_{i-1}) is less than or equal to the smallest element on the right partition of B (B_j), and vice versa. If this condition is met, the median is the maximum of A_{i-1} and B_{j-1}.
- **Adjusting Search Range:** If the condition is not met, the algorithm adjusts the search range by modifying *low* or *high* based on the comparison results.
- **Efficiency:** Since the search range is halved in each iteration, the algorithm runs in $O(\log n)$ time.

Notes

- **Edge Cases:** Assigning $+\infty$ and $-\infty$ for out-of-bound indices ensures that comparisons remain valid without accessing invalid array positions.
- **Uniqueness of Elements:** The assumption that all elements are unique simplifies the comparison logic.

Acknowledgement

I would like to acknowledge that I used ChatGPT to help refine my grammar and sentence structure in this document. However, the solutions and ideas presented here are entirely my own.