DEVIN MAYA 1761702 7/11/24

CSE 102 Homework 2

Exercise 1: More Recursions. Solve the following recursive relations (assume all base cases $T(1) = 1$ and/or that n is a power of 2 if it helps you):

(i) $T(n) = 2T(\sqrt{n}) + \log n$ (Hint: Maybe replacing n with 2^m will help you.)

(ii) $T(n) = 2T(n/2) + n \log n$ (Hint: You may want to check again the proof of the Master Theorem.)

(iii) $T(n) = 2T(\sqrt{n}) + 1$

(iv) $T(n) = T(n/2) + T(n/3) + T(n/10) + T(n/20) + n$ (Hint: Check again the substitution method we saw in class.)

Solution.

Solution.

(i) $T(n) = 2T(\sqrt{n}) + \log n$

We use the substitution method:

- Let $m = \log n$, so $n = 2^m$ and $\sqrt{n} = 2^{m/2}$
- Define $S(m) = T(2^m)$

This gives us:

$$S(m) = 2S(m/2) + m$$

Now we can apply the Master Theorem:

- $a = 2$, $b = 2$, $f(m) = m$
- $\log_b a = \log_2 2 = 1$
- $f(m) = m = \Theta(m^{\log_b a})$, so we're in Case 2

Therefore, $S(m) = \Theta(m \log m)$.
Substituting back: $T(n) = \Theta(\log n \log \log n)$.

Solution.

(ii) $T(n) = 2T(n/2) + n \log n$

This doesn't fit the Master Theorem directly. We use the substitution method:

Guess: $T(n) \le cn \log^2 n$ for some constant c.

Inductive step: Assume true for all values less than n.

$$\begin{aligned} T(n) &= 2T(n/2) + n \log n \\ &\le 2\left(c\frac{n}{2} \log^2\left(\frac{n}{2}\right)\right) + n \log n \\ &= cn\left(\log^2 n - 2\log n + \log^2 2\right) + n \log n \\ &= cn \log^2 n - 2cn \log n + cn + n \log n \end{aligned}$$

For our guess to be correct:

$$cn \log^2 n - 2cn \log n + cn + n \log n \le cn \log^2 n$$

Simplifying:

$$n \log n \le 2cn \log n - cn$$

$$\log n \le (2c - 1) \log n - c$$

This holds for large n and $c > 1$.

Therefore, $T(n) = O(n \log^2 n)$.

Solution.

(iii) $T(n) = 2T(\sqrt{n}) + 1$

Similar to (i), we use substitution:

- Let $m = \log n$, so $n = 2^m$ and $\sqrt{n} = 2^{m/2}$
- Define $S(m) = T(2^m)$

This gives us:

$$S(m) = 2S(m/2) + 1$$

Applying the Master Theorem:

- $a = 2$, $b = 2$, $f(m) = 1$
- $\log_b a = \log_2 2 = 1$
- $f(m) = 1 = \Theta(m^0)$, and $0 < \log_b a$, so we're in Case 1

Therefore, $S(m) = \Theta(m)$.
Substituting back: $T(n) = \Theta(\log n)$.

Solution.

(iv) $T(n) = T(n/2) + T(n/3) + T(n/10) + T(n/20) + n$

We use the substitution method, guessing a linear solution:

Guess: $T(n) \leq cn$ for some constant c.

Inductive step: Assume true for all values less than n.

$$\begin{aligned} T(n) &\leq c\left(\frac{n}{2}\right) + c\left(\frac{n}{3}\right) + c\left(\frac{n}{10}\right) + c\left(\frac{n}{20}\right) + n \\ &= cn\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{10} + \frac{1}{20}\right) + n \\ &= cn\left(\frac{31}{60}\right) + n \end{aligned}$$

For our guess to be correct:

$$cn\left(\frac{31}{60}\right) + n \leq cn$$

Simplifying:

$$n \leq cn\left(\frac{29}{60}\right)$$

$$\frac{60}{29} \leq c$$

This holds for $c \geq \frac{60}{29} \approx 2.07$.

Therefore, $T(n) = O(n)$.

Exercise 2: k-way merge operation. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(i) Here's one strategy: Using the merge procedure from class, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

(ii) Give a more efficient solution to this problem, using divide-and-conquer.

Solution.

Exercise 2: k-way Merge Operation

Problem

We have k sorted arrays, each containing n elements. Our goal is to combine them into a single sorted array of kn elements.

(i) Iterative Merging Approach

Let's examine the time complexity of merging k sorted arrays iteratively:

- Initial merge of arrays 1 and 2: $O(2n)$
- Merging result with array 3: $O(3n)$
- Merging new result with array 4: $O(4n)$
- $\vdots$
- Final merge with array k: $O(kn)$

To calculate the total time, we sum up all these steps:

$$\begin{aligned}\text{Total Time} &= 2n + 3n + 4n + \cdots + kn \\ &= n(2 + 3 + 4 + \cdots + k) \\ &= n\sum_{i=2}^{k} i\end{aligned}$$

We can simplify this sum using the formula for the sum of an arithmetic sequence:

$$\begin{aligned}\sum_{i=2}^{k} i &= \frac{k(k+1)}{2} - 1 \\ &= \frac{k^2 + k - 2}{2}\end{aligned}$$

Therefore, the total time complexity is:

$$\begin{aligned}T(k,n) &= n \cdot \frac{k^2 + k - 2}{2} \\ &= O(nk^2)\end{aligned}$$

Thus, the time complexity of this iterative approach is $O(nk^2)$.

(ii) Recursive Divide-and-Conquer Approach

A more efficient solution employs a recursive partitioning approach:

1. Divide the k arrays into two equal groups.
2. Recursively merge each group.
3. Combine the two merged groups.

Time complexity analysis:

$$T(k,n) = 2T\left(\frac{k}{2}, n\right) + M(k,n) \tag{1}$$

where $M(k,n) = O(kn)$ is the time to merge two sorted arrays of total length kn.

This recurrence can be solved using the Master Theorem:

- $a = 2$ (subproblems)
- $b = 2$ (size reduction factor)
- $f(k) = M(k,n) = O(kn)$

We find that $\log_b a = \log_2 2 = 1$, and $f(k) = \Theta(k)$ when considering n as a constant.

This puts us in Case 2 of the Master Theorem, which states that if $f(k) = \Theta(k^{\log_b a} \log^p k)$ for some $p \geq 0$, then $T(k) = \Theta(k^{\log_b a} \log^{p+1} k)$. In our case, $p = 0$, so:

$$T(k,n) = \Theta(k \log k) \cdot n = O(kn \log k) \tag{2}$$

Implementation steps:

1. Base case: If $k = 1$, return the array.
2. Divide: Split into two groups of $k/2$ arrays.
3. Conquer: Recursively merge each group.
4. Combine: Merge the two sorted results.

Comparison

The divide-and-conquer method ($O(kn \log k)$) is more efficient than the iterative method ($O(nk^2)$), especially for large values of k.

For example, if $k = 1000$ and $n = 1000$:

- Iterative method: $O(1000^2 \cdot 1000) = O(10^9)$
- Divide-and-conquer: $O(1000 \cdot 1000 \cdot \log 1000) \approx O(10^7)$

The divide-and-conquer method is about 100 times faster in this case. This recursive method significantly outperforms the iterative approach for large k values, reducing complexity from $O(nk^2)$ to $O(kn \log k)$.

Exercise 3: Interview Question. You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

Solution.

Exercise 3: Locating the k-th Element in the Merged Sequence of Two Sorted Arrays

Problem

Given:

- Two sorted arrays X of length $p = m$ and Y of length $q = n$
- An integer k where $1 \leq k \leq p + q$

Objective: Design an algorithm with $O(\log p + \log q)$ time complexity to find the k-th smallest element in the hypothetical merged array of X and Y.

Algorithmic Approach: Adaptive Partition Search

We propose an adaptive partition search method that efficiently narrows down the location of the k-th element without actually merging the arrays.

Key Concept

The algorithm dynamically adjusts partitions in both arrays to converge on the k-th element's position.

Algorithm Outline

1. Ensure $p \leq q$ by swapping X and Y if necessary.
2. Initialize search boundaries:
    - $\alpha = \max(0, k - q)$
    - $\beta = \min(p, k)$
3. While $\alpha \leq \beta$:
    - Set $\gamma = \lfloor (\alpha + \beta)/2 \rfloor$
    - Calculate $\delta = k - \gamma$
    - Evaluate partition validity:
        - If $X[\gamma - 1] \leq Y[\delta]$ and $Y[\delta - 1] \leq X[\gamma]$:
            - Return $\max(X[\gamma - 1], Y[\delta - 1])$
        - Else if $X[\gamma - 1] > Y[\delta]$:
            - Update $\beta = \gamma - 1$
        - Else:
            - Update $\alpha = \gamma + 1$

Complexity Analysis

- Time Complexity: $O(\log p)$ for binary search on X, plus $O(\log q)$ for initial boundary calculations.
- Space Complexity: $O(1)$ as only a constant number of variables are used.

Correctness Justification

The algorithm maintains an invariant that the k-th element lies within the current search boundaries. Each iteration refines these boundaries, ensuring convergence to the correct partition that separates the k smallest elements from the rest.

Edge Case Handling

- $k = 1$: Return $\min(X[0], Y[0])$
- $k = p + q$: Return $\max(X[p - 1], Y[q - 1])$
- Empty array: Return the k-th element from the non-empty array

Illustrative Example

Consider $X = [2, 4, 6, 8]$, $Y = [1, 3, 5, 7, 9]$, and $k = 6$

- Initial state: $\alpha = 1$, $\beta = 4$
- First iteration: $\gamma = 2$, $\delta = 4$
- Compare $X[1] = 4$, $X[2] = 6$, $Y[3] = 5$, $Y[4] = 7$
- Condition $X[1] \leq Y[4]$ and $Y[3] \leq X[2]$ is satisfied
- The 6th smallest element is $\max(X[1], Y[3]) = \max(4, 5) = 5$

Exercise 4: Factorial Complexity. Recall the factorial of n, denoted $n!$, which is equal to the product $1 \cdot 2 \cdot 3 \cdots n$. Show that $\log(n!) = \Theta(n \log n)$. (You can use Stirling's approximation for the factorial, but there is a way to do it without it too.)

Solution.

Exercise 4: Analyzing the Growth Rate of $\log(n!)$

Goal

We aim to demonstrate that $\log(n!) = \Theta(n \log n)$.

Method

We'll establish both a lower and an upper bound for $\log(n!)$ using fundamental properties of logarithms and simple inequalities.

Lower Bound Derivation

Let's focus on the larger terms in the factorial:

$$\begin{aligned}
\log(n!) &= \log(1 \times 2 \times 3 \times \cdots \times n) \\
&= \log 1 + \log 2 + \log 3 + \cdots + \log n \\
&\geq \log\left(\frac{n}{2} + 1\right) + \log\left(\frac{n}{2} + 2\right) + \cdots + \log n
\end{aligned}$$

Here, we've considered only the upper half of the terms. Now, let's simplify:

$$\begin{aligned}&\geq \frac{n}{2}\log\left(\frac{n}{2}\right)\\&= \frac{n}{2}(\log n - \log 2)\\&= \frac{1}{2}n\log n - \frac{n}{2}\log 2\end{aligned}$$

This shows that $\log(n!) \geq \frac{1}{2}n\log n - O(n)$, or in other words, $\log(n!) = \Omega(n\log n)$.

Upper Bound Derivation

For the upper bound, we'll use the fact that each term in the sum is at most $\log n$:

$$\begin{aligned}\log(n!) &= \log 1 + \log 2 + \log 3 + \cdots + \log n\\&\leq \log n + \log n + \log n + \cdots + \log n \quad \text{(n times)}\\&= n\log n\end{aligned}$$

This demonstrates that $\log(n!) \leq n\log n$, or $\log(n!) = O(n\log n)$.

Combining the Results

Putting our lower and upper bounds together:

$$\frac{1}{2}n\log n - \frac{n}{2}\log 2 \leq \log(n!) \leq n\log n$$

Both the lower and upper bounds are of the form $cn\log n + O(n)$, where c is a constant and $\frac{1}{2} \leq c \leq 1$.

Conclusion

Since we have bounded $\log(n!)$ both above and below by expressions that are $\Theta(n\log n)$, we can conclude:

$$\log(n!) = \Theta(n\log n)$$

Intuitive Explanation

The factorial function $n!$ grows extremely rapidly, faster than any polynomial but not as fast as exponential functions like n^n. When we take the logarithm of $n!$, we "dampen" this explosive growth, resulting in a function that increases at the same rate as $n\log n$.

Our lower bound comes from the observation that at least half of the numbers in the factorial are greater than or equal to $n/2$. The upper bound reflects the fact that none of the numbers in the product exceed n itself.

Exercise 5: Function Growth Rates Sort the following functions in increasing order of magnitude, that is from the smaller to larger growth rate. If two of them are asymptotically of the same order, you should note it. Explain your answer by comparing every two consecutive functions in your sorted order:

$$2^{2^n} \; n! \; n2^n \; 10n \; \log(n!) \; \log^{10} n \; n^{\log\log n} \; \left(\frac{\log n}{\log\log n}\right)^{\frac{\log n}{loglogn}} \; \log n^3 \; \frac{n}{\log n} \; \frac{n^2}{\log^{10} n} \; \frac{\log n}{n} \; 3n^6 \; e^n \; \sqrt{n!} \; \binom{n}{4}$$

Solution.

Exercise 5: Function Growth Rates

Sorted in increasing order of growth rate:

1. $\frac{\log n}{n}$
2. $\log n^3$
3. $\frac{n}{\log n}$
4. $\log^{10} n$
5. $\log(n!)$
6. $10n$
7. $\frac{n^2}{\log^{10} n}$
8. $n^{\log\log n}$
9. $\sqrt{n!}$
10. $\binom{n}{4}$
11. $3n^6$
12. e^n
13. $\left(\frac{\log n}{\log\log n}\right)^{\frac{\log n}{\log\log n}}$
14. n^{2^n}
15. $n!$

16. 2^{2^n}

Explanations for consecutive comparisons:

1. $\frac{\log n}{n}$ vs $\log n^3$: As n grows, n in the denominator makes $\frac{\log n}{n}$ decrease faster than $\log n^3$ grows.
2. $\log n^3$ vs $\frac{n}{\log n}$: For large n, n grows faster than any power of $\log n$.
3. $\frac{n}{\log n}$ vs $\log^{10} n$: Eventually, $\log^{10} n$ surpasses $\frac{n}{\log n}$ as n increases.
4. $\log^{10} n$ vs $\log(n!)$: Since $\log(n!)$ is approximately $n \log n$, it grows faster than $\log^{10} n$.
5. $\log(n!)$ vs $10n$: Linear growth eventually exceeds logarithmic growth.
6. $10n$ vs $\frac{n^2}{\log^{10} n}$: The n^2 term dominates $\log^{10} n$ for large n.
7. $\frac{n^2}{\log^{10} n}$ vs $n^{\log \log n}$: For large n, $n^{\log \log n}$ grows faster.
8. $n^{\log \log n}$ vs $\sqrt{n!}$: $\sqrt{n!}$ is approximately $\Theta(n^{n/2})$, which grows faster than $n^{\log \log n}$.
9. $\sqrt{n!}$ vs $\binom{n}{4}$: $\sqrt{n!}$ grows faster as it's approximately $\Theta(n^{n/2})$, while $\binom{n}{4}$ is $\Theta(n^4)$.
10. $\binom{n}{4}$ vs $3n^6$: $\binom{n}{4}$ is $\Theta(n^4)$, which grows slower than $3n^6$.
11. $3n^6$ vs e^n: Exponential growth eventually surpasses any polynomial.
12. e^n vs $\left(\frac{\log n}{\log \log n}\right)^{\frac{\log n}{\log \log n}}$: The latter grows faster than e^n for very large n.
13. $\left(\frac{\log n}{\log \log n}\right)^{\frac{\log n}{\log \log n}}$ vs n^{2^n}: n^{2^n} grows much faster for large n.
14. n^{2^n} vs $n!$: While both grow extremely fast, $n!$ eventually surpasses n^{2^n} for very large n.
15. $n!$ vs 2^{2^n}: Double exponential growth is the fastest growing function in this list

Ex.6 (Binary Search Lower Bound) In class we saw how sorting n numbers using only comparisons, requires $\Omega(n \log n)$ time, that is **no** algorithm (no matter how clever it is) can sort every input array in significantly less time than $n \log n$. Consider now the problem of searching a number x in a sorted array A (that may or may not contain x). We know binary search does this in $O(\log n)$ time.

Can we do better? In other words, is there a searching algorithm (that uses comparisons) that runs in time $o(\log n)$ and correctly determines whether or not an element x appears in A? (hint: think of a tree representation for the execution of any algorithm, as we did in class for sorting. How many leaves are there in the tree? What should the depth of this tree be?)

Solution. We know binary search finds an element in a sorted array in $O(\log n)$ time. Is it possible to create a faster algorithm that uses comparisons and runs in less than $O(\log n)$ time?

Answer

No, it's not possible. Binary search is already the fastest we can do when we're limited to using comparisons. Here's why:

Explanation

Step 1: Thinking About All Possible Outcomes

First, let's consider all the possible results of our search:

- The element could be at any of the n positions in the array.
- The element might not be in the array at all.

So, there are $n + 1$ possible outcomes in total.

Step 2: Visualizing the Search Process

Imagine the search process as a tree:

- Each time we make a comparison, we're at a branching point in the tree.
- After each comparison, we go left or right based on the result.
- We keep doing this until we reach a final answer (a leaf of the tree).

Step 3: Counting the Leaves

Remember, we need at least $n + 1$ leaves in our tree to represent all possible outcomes.

Step 4: Relating Tree Height to Number of Leaves

In a binary tree (where each node has at most 2 children):

- If the tree has height h, it can have at most 2^h leaves.
- We need at least $n + 1$ leaves.

So, we can write: $2^h \geq n + 1$

Step 5: Solving for the Minimum Height

Taking the logarithm of both sides:

$$[h \geq \log_2(n+1)]$$

This means the tree must have a height of at least $\log_2(n+1)$.

Step 6: Interpreting the Result

The height of the tree represents the number of comparisons in the worst case:

- We need at least $\log_2(n+1)$ comparisons.
- This is essentially the same as $\log_2(n)$ for large n.

Conclusion

Any algorithm that uses comparisons to search a sorted array must sometimes use at least $\log_2(n)$ comparisons. This means:

- We can't create an algorithm that always runs faster than $O(\log n)$ time.
- Binary search, which runs in $O(\log n)$ time, is already as fast as possible.

Simple Analogy

Think of it like a guessing game for a number between 1 and 100:

- The best strategy is to eliminate half the possibilities with each guess.
- You'll always find the number in at most 7 guesses ($\log_2(100) \approx 6.64$).
- There's no way to guarantee finding the number in fewer guesses without extra information or lucky guessing.

Exercise 7: Inversions. Describe and analyze an algorithm to count the number of inversions in an array A of n numbers is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2} = n(n-1)/2$ (if the array is sorted backwards). Convince yourselves about the last statement.

Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. (Hint: Modify Mergesort.)

Solution.

Exercise 7: Counting Inversions in an Array

Range

The number of inversions in an n-element array can be:

- **Minimum:** 0 (when the array is in ascending order)
- **Maximum:** $\binom{n}{2} = \frac{n(n-1)}{2}$ (when the array is in descending order)

Why $\frac{n(n-1)}{2}$? In a reverse-sorted array, each element forms an inversion with every element after it. This sum is equivalent to adding the first $(n-1)$ natural numbers. We'll modify the standard Merge Sort algorithm to count inversions while sorting.

Objective

During merging, if we place an element from the right subarray before elements from the left, those left elements form inversions with the right element.

Algorithm

1. If the array has 0 or 1 element, return 0.
2. Otherwise:
   (a) Split array in half.
   (b) Count inversions in left half (recursive call).
   (c) Count inversions in right half (recursive call).
   (d) Count inversions across the split while merging.
   (e) Sum up all inversions.

Pseudocode

```
1. CountInversions(A, start, end)
    (a) If start $\geq$ end, return 0.
    (b) Set mid = $\left\lfloor \frac{\text{start}+\text{end}}{2} \right\rfloor$.
    (c) Compute left_inversions = CountInversions(A, start, mid).
    (d) Compute right_inversions = CountInversions(A, mid + 1, end).
    (e) Compute split_inversions = MergeAndCount(A, start, mid, end).
    (f) Return left_inversions + right_inversions + split_inversions.
```

2. **MergeAndCount(A, start, mid, end)**

    (a) Initialize left = A[start:mid+1] and right = A[mid+1:end+1].

    (b) Initialize inversions = 0, i = 0, j = 0, k = start.

    (c) While $i <$ len(left) and $j <$ len(right):

    - If left[i] $\leq$ right[j]:
        - A[k] = left[i]
        - i += 1
    - Else:
        - A[k] = right[j]
        - j += 1
        - inversions += len(left) - i
    - k += 1

    (d) Copy remaining elements of left and right into A, if any.

    (e) Return inversions.

Time Complexity

- This algorithm follows Merge Sort's structure.
- Recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$.
- Solving this, we get: $T(n) = O(n \log n)$.

Correctness

- For 0 or 1 element, it's trivially correct.
- For n elements, assuming it works for smaller arrays:
    - It correctly counts inversions in each half.
    - It accurately counts split inversions during merging.
    - The sum gives the total inversions.

Example

For array $[3, 1, 4, 2]$:

- Split into $[3, 1]$ and $[4, 2]$.

- Left half has 1 inversion ($3 > 1$).
- Right half has 1 inversion ($4 > 2$).
- Merging: 1 more inversion ($3 > 2$).
- Total: $1 + 1 + 1 = 3$ inversions.

Exercise 8: Tiling. In front of you, there is a 2-by-n chessboard, that you want to completely cover with tiles that have size 1-by-2. Let $T(n)$ denote the number of ways that you can tile the chessboard using the tiles. Write down a recursive relation for $T(n)$ and then using it, compute the total number of possible tilings of the chessboard.

Solution.

Exercise 8: Covering a 2-by-n Board with Dominoes

Finding the Pattern

Let's start by counting the ways for small boards:

For a 2-by-1 board:

Only one way: stand the domino vertically So, T(1) = 1

For a 2-by-2 board:

Two ways: two vertical dominoes, or two horizontal dominoes So, T(2) = 2

For a 2-by-3 board:

We can either: a) Place a vertical domino at the start, then cover the remaining 2-by-2 area (T(2) ways) b) Place two horizontal dominoes at the start, then cover the remaining 2-by-1 area (T(1) way) So, T(3) = T(2) + T(1) = 2 + 1 = 3

The Recursive Relation

We can see a pattern forming. For any n ¿ 2:

We can start with a vertical domino and then cover the rest (2-by-(n-1) board) in T(n-1) ways Or we can start with two horizontal dominoes and cover the rest (2-by-(n-2) board) in T(n-2) ways

This gives us the recursive relation: T(n) = T(n-1) + T(n-2) With base cases: T(1) = 1 T(2) = 2

Calculating More Values

Let's calculate a few more values: T(4) = T(3) + T(2) = 3 + 2 = 5 T(5) = T(4) + T(3) = 5 + 3 = 8 T(6) = T(5) + T(4) = 8 + 5 = 13

Observations

This sequence (1, 2, 3, 5, 8, 13, ...) is the Fibonacci sequence. Each number is the sum of the two before it. The number of ways to tile the board grows quickly as n increases. We've found that the number of ways to cover a 2-by-n board with dominoes follows the Fibonacci sequence. We can find T(n) by adding the two previous values in the sequence.