DEVIN MAYA 1761702 7/3/24

CSE 102 Homework 1

EX.1 The famous game "Towers of Hanoi" asks you to move 3 (more generally n) wooden disks from the left rod to the right rod such that a larger disk never lands on top of a smaller disk. The goal is to understand how many moves it would take to perform the task. A move is considered an action that takes one disk from any rod and puts it in any other rod. Let $T(n)$ be the runtime of an algorithm to complete the task if there were n disks in total (play with $n = 3$ first). Provide such an algorithm whose runtime obeys:

$$T(n) = 2T(n-1) + 1$$

(hint: recursion/induction). Then, prove that $T(n) = 2n - 1$, $\forall n$. Bonus: As algorithm designers, the next question here is "can we do better?"

Solution.

Algorithm Description

The algorithm is executed recursively with the following detailed steps:

Base Case: If $n = 1$, simply move the disk from the source rod to the destination rod. This operation requires exactly one move.

Recursive Steps: For $n > 1$:

1. Move the top $n-1$ disks from the source rod to the intermediate rod while utilizing the destination rod as a temporary holding area.
2. Move the largest disk which is the n-th disk directly from the source rod to the destination rod.
3. Transfer the $n-1$ disks from the intermediate rod to the destination rod, positioning them atop the largest disk.

In class this was shown to take 7 moves considering 3 disks and 3 poles.

Proof of Correctness

Inductive Approach:

Base Case: For $n = 1$, the move is trivially correct as it complies with all rules.

Inductive Hypothesis: Assume the procedure correctly solves the problem for $n - 1$ disks.

Inductive Step: Given the hypothesis, moving the $n-1$ disks to the intermediate rod clears the way for the largest disk to move directly to the destination rod. Subsequently, moving the $n-1$ disks atop the largest disk at the destination rod completes the process without contravening the constraints, thus proving the method's correctness for n disks by induction.

Runtime Analysis

Recurrence Relation: The total number of moves required by the algorithm is defined by the recurrence relation:

$$T(n) = 2T(n-1) + 1$$

where $T(n)$ represents the total moves for n disks.

Proof by Induction:

Base Case: $T(1) = 1$ is trivial as there is only one move involved.

Inductive Hypothesis: Assume $T(k) = 2^k - 1$ holds for all $k < n$.

Inductive Step: For $T(n)$, using the hypothesis:

$$T(n) = 2T(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

confirming that the relation holds for n, thus $T(n) = 2^n - 1$.

Optimality

The recursive algorithm is optimal in terms of the number of moves, as no fewer than $2^n - 1$ moves can solve the problem, evidenced by the recursive solution to the defined recurrence relation. This mathematical proof shows that every move in the sequence is necessary and sufficient to achieve the goal under the given constraints.

EX. 2 In algorithms, we use the Big-O notation to abstract away unnecessary details about the runtime of an algorithm.

Here is the big-O definition (may be useful for other exercises too):

Let $n > 0$ (usually n is the size of the input) and let $f(n)$ and $g(n)$ be two monotone and non-negative functions. We say that $f(n) = O(g(n))$ if there exists a constant c and a constant n_0 such that: $f(n) \leq cg(n), \forall n > n_0$.

The exercise asks you to use the definition to prove the following:

- Let $f(n) = 100n^2 + 10n + 1000$. Then $f(n) = O(n^2)$.
- Let $f(n) = 100n + 0.001n \log n$. Then $f(n) = O(n \log n)$.

- Let $f(n) = 50n \log n + 30n$. Then $f(n) = O(n^3)$.

Feel free to use standard properties or inequalities for the logarithm without proof.

Solution.

Proving $f(n) = 100n^2 + 10n + 1000 = O(n^2)$

Given Function $f(n) = 100n^2 + 10n + 1000$

The objective is to show that $f(n) = O(n^2)$.

To establish this relationship, we need constants c and n_0 such that:

$$100n^2 + 10n + 1000 \leq cn^2 \quad \text{for all } n > n_0.$$

Select $c = 111$ and $n_0 = 1$. For $n \geq 1$,

$$100n^2 + 10n + 1000 \leq 100n^2 + 10n^2 + 1000 = 110n^2 + n^2 = 111n^2.$$

Thus, $f(n)$ is bounded by $111n^2$ for all $n \geq 1$, verifying that $f(n) = O(n^2)$.

Proving $f(n) = 100n + 0.001n \log n = O(n \log n)$

Given Function $f(n) = 100n + 0.001n \log n$ Prove that $f(n) = O(n \log n)$. find constants c and n_0 such that:

$$100n + 0.001n \log n \leq cn \log n \quad \text{for all } n > n_0.$$

Let $c = 101$ and $n_0 = 2$. Then for $n \geq 2$,

$$100n + 0.001n \log n \leq 100n \log n + 0.001n \log n = 100.001n \log n.$$

This shows that $f(n)$ grows no faster than $n \log n$, verifying $f(n) = O(n \log n)$.

Proving $f(n) = 50n \log n + 30n = O(n^3)$

Given Function $f(n) = 50n \log n + 30n$ show that $f(n) = O(n^3)$.

Find c and n_0 such that:

$$50n \log n + 30n \leq cn^3 \quad \text{for all } n > n_0.$$

Choose $c = 1$ and $n_0 = 2$, ensuring:

$$50n \log n + 30n \leq n^3 \quad \text{for } n \geq 2.$$

This shows that $f(n)$ is majorly bounded by n^3, hence $f(n) = O(n^3)$.

EX. 3 Imagine you had an algorithm with runtime $T(n)$ satisfying the recursion:

$$T(n) = T(n/2) + 1 \quad \text{and} \quad T(1) = 1$$

You have probably seen the binary search algorithm that exactly obeys that recursion. Prove that $T(n) = O(\log n)$.

Then, imagine another algorithm with $Q(n)$ satisfying:

$$Q(n) = Q(n/2) + n \quad \text{and} \quad Q(1) = 1$$

Prove that $Q(n) = O(n)$.

Solution.

Analysis of $T(n)$

Recursive Definition: Given $T(n) = T(n/2) + 1$ with $T(1) = 1$ the objective is to prove that $T(n) = O(\log n)$.

Proof:

Base Case: $T(1) = 1$. The base case satisfies the claim as $\log 1 = 0$ and any positive constant c.

Inductive Hypothesis: Assume for some $k < n$, $T(k) = O(\log k)$ is true through the implementation of induction.

Inductive Step: Consider $T(n)$:

$$T(n) = T(n/2) + 1 = O(\log(n/2)) + 1 = O(\log n - \log 2) + 1 = O(\log n).$$

Therfore $T(n)$ satisfies the algorithm complexity of $O(\log n)$ since it halves the problem size at each step and adds a constant operation.

Analysis of $Q(n)$

Recursive Definition: Given $Q(n) = Q(n/2) + n$ with $Q(1) = 1$, shown that $Q(n) = O(n)$.

Proof:

As shown in class when referencing a recursion tree and unpacking the relation. Each level of the recursion tree contributes n to the total cost but the number of elements handled reduces by .5:

$$n + n + n + \ldots \text{ for } \log n \text{ levels.}$$

The summation of costs across all levels forms a geometric progression totaling $n \log n$. However, each recursive call contributes significantly less as the depth increases. Given that the dominant term is n from the highest recursion level, and noting the diminishing contributions of subsequent levels, we conclude:

$$Q(n) \leq cn, \text{ where } c \text{ is a constant,}$$

ensuring $Q(n) = O(n)$.

EX 4. Describe an $O(n \log n)$-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x.

Solution.

Algorithm Description

The algorithm uses sorting and a two-pointer technique to find two elements in a set S of n integers that sum to a given integer x.

Algorithm Steps:

- Sort the elements of S using mergesort or heapsort, ensuring $O(n \log n)$ complexity for the sorting phase.
- Initialize two pointers:
  - $i = 1$ (Start pointer at the beginning of S)
  - $j = n$ (End pointer at the end of S)
- While $i < j$:
  - Calculate sum = $S[i] + S[j]$.
  - If sum = x, output True and terminate (a pair is found).
  - If sum < x, increment i to increase the sum.
  - If sum > x, decrement j to decrease the sum.
- If no valid pair is found by the time pointers meet, output False.

Runtime Analysis

The sorting step dominates the runtime, adhering to $O(n \log n)$. The search with two pointers takes $O(n)$ time, ensuring that the overall complexity remains within $O(n \log n)$.

EX 5. Follow the $O(\cdot)$ definition & prove:

- Is $2n + 2023 = O(2^n)$?
- Is $2^{2n} = O(2^n)$?

Solution.

Analysis of $2n + 2023 = O(2^n)$

Objective: Prove whether the function $2n+2023$ is bounded above by the exponential function 2^n within a constant factor for all sufficiently large n.

Proof: We need to find constants c and n_0 such that:

$$2n + 2023 \leq c \cdot 2^n \quad \text{for all } n > n_0.$$

Test for constants: Check an initial value of n where $2n + 2023$ might be close to or less than 2^n. Choosing $n = 11$, we calculate:

$$2 \times 11 + 2023 = 2045 \quad \text{and} \quad 2^{11} = 2048.$$

Choice of c and n_0: With $2045 \leq 2048$, setting $c = 1$ and $n_0 = 11$ ensures the inequality holds. General case for large n: As n increases, 2^n grows exponentially observe that it is quickly surpassing the linear growth of $2n+2023$. Thus, for any $n > 11$, the inequality $2n+2023 \leq 2^n$ is satisfied, confirming $2n + 2023 = O(2^n)$.

Analysis of $2^{2n} = O(2^n)$

Objective Determine if the function 2^{2n}, representing exponential growth squared, is bound above by the exponential function 2^n multiplied by any constant c.

Proof: To prove this, we look for constants c and n_0 such that:

$$2^{2n} \leq c \cdot 2^n \quad \text{for all } n > n_0.$$

Rewriting the inequality: Express 2^{2n} as $(2^n)^2$, and rearrange the inequality:

$$(2^n)^2 \leq c \cdot 2^n.$$

Simplification: Simplify to find $2^n \leq c$. Contradiction: As n increases, 2^n grows indefinitely observe it is surpassing any fixed c. Thus, no constant c exists that satisfies the inequality for all sufficiently large n, indicating $2^{2n} \neq O(2^n)$.

EX 6. Imagine you came up with an algorithm that depends on a parameter $k \in [1, n]$ you can

control. After calculations, the runtime ends up being $O(n^k + \frac{n^2}{k})$. What choice of k would you make for your algorithm?

Solution.

Algorithm Analysis

This problem involves finding a balance between an exponential term n^k and an inversely proportional term $\frac{n^2}{k}$ to minimize the overall runtime expression.

Optimal k

Set n^k equal to $\frac{n^2}{k}$ to balance the terms:

$$n^k = \frac{n^2}{k}$$

Applying logarithms on both sides:

$$k \log n = \log n^2 - \log k$$

$$k \log n + \log k = 2 \log n$$

Rearranging gives us:

$$\log k + k \log n = 2 \log n$$

This equation suggests that balancing the two terms involves a complex relationship between k and n, typically solved using numerical methods for precise optimization.

Estimation: Estimate k by considering $k \approx \sqrt{n}$, balancing the growth rates of n^k and $\frac{n^2}{k}$. For large n, testing values around $\sqrt{n}$ can help refine the choice of k as shown in class. The optimal choice of k to minimize the runtime expression $O(n^k + \frac{n^2}{k})$ is approximated by $k \approx \sqrt{n}$.

This equation can be simplified by making an initial estimate that balances the growth of n^k and the decay of $\frac{n^2}{k}$. An approach is to consider k that simplifies the complexity, such as $k \approx \sqrt{n}$. :

$$k \log n \approx 2 \log n$$

$$k \approx \frac{2 \log n}{\log n}$$

$$k \approx 2$$

EX 7. This is from a common coding interview question: Imagine you are given a list of n integers. Assume they are distinct if that helps you. Give an algorithm that finds the smallest and the second smallest element in the list, using at most $n + \log n$ comparisons, and explain why

your algorithm needs at most $n + \log n$ comparisons.

Solution.

Algorithm Description

The method relates to the pairwise comparison for min/max algorithm shown in class among the elements, efficiently identifying the smallest and second smallest elements.

Algorithm Steps:

Pairwise Comparisons as shown in class: Pair up elements and compare each pair to determine a winner as smaller element and a loser as larger element. This process reduces the number of candidates for the smallest element by half in each round, ultimately requiring $\log n$ rounds to identify the smallest element.Tracking Candidates for Second Smallest: Record elements that lose directly to the eventual smallest element. These elements are the only candidates for the second smallest because they were the ones directly compared and lost to the smallest. Determining the Second Smallest: Perform a final comparison among all elements that directly lost to the smallest element. The number of these comparisons is exactly the number of matches the smallest element won, which is at most $\log n$.

Complexity Analysis

Finding the Smallest: The process involves $n - 1$ comparisons in total to identify the smallest element, as each element except the smallest loses exactly once.

Finding the Second Smallest: Comparing the runners-up (those who lost directly to the smallest) involves at most $\log n$ additional comparisons.

Total Comparisons: The total number of comparisons is therefore:

$$(n - 1) + \log n \leq n + \log n,$$

meeting the requirement of at most $n + \log n$ comparisons.

Explanation

The method is particularly effective for this problem because it systematically reduces the number of candidates for the smallest and second smallest through direct eliminations, ensuring no unnecessary comparisons. Direct competitors of the smallest element are logically the only possible candidates for the second smallest, optimizing the search and comparison process.

EX 8. (Binary Search, Unlimited) You are given an infinite array $A[\cdot]$ in which the first n cells

contain integers in sorted order and the rest of the cells are filled with ∞. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

Solution.

Algorithm

The algorithm operates in two main steps first determining the boundary of the sorted segment and then executing a binary search within this established boundary.

Algorithm Steps: Establishing the Bounds

Initialize: Start with an index $i = 1$. Exponential Search: Increase i exponentially so $i \leftarrow 2i$ until $A[i] = \infty$. This approach quickly pinpoints the upper boundary of the sorted elements. Set Bounds: Assign $high = i$ and $low = i/2$ to define the range likely containing x.

Binary Search

Search Execution: Perform a binary search on the interval from low to $high - 1$:

Calculate the midpoint $mid = \lfloor (low + high)/2 \rfloor$. If $A[mid] = x$, return mid as the index where x is located. Adjust search bounds based on comparison results:

If $A[mid] < x$, set $low = mid + 1$. If $A[mid] > x$, set $high = mid - 1$.

If no match is found by the time low exceeds $high$, conclude that x does not exist within the sorted bounds.

Complexity Analysis

Exponential Search: Finding the upper limit of sorted elements with exponential search ensures that the boundary is located in $O(\log n)$ steps, as it doubles the search index at each step until overshooting n. Binary Search: Conducting a binary search within the identified bounds takes $O(\log n)$ time since the range of search is halved with each iteration.

EX 9. (Index-Value Match) Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

Solution.

Algorithm Description

This algorithm implements the binary search method to specifically adress the case where the index and its value are equal in a sorted and distinct integer array.

Initialization: Set $low = 1$ and $high = n$ to define the bounds of the search space.

Modified Binary Search Process:

- Execute a loop while $low \leq high$:
  1. Compute the midpoint $mid = \left\lfloor \frac{low+high}{2} \right\rfloor$ to divide the search space.
  2. Assess the relationship between mid and $A[mid]$:
     - If $A[mid] = mid$, a match has been found; return mid.
     - If $A[mid] < mid$, then the search continues in the upper half of the array. Set $low = mid + 1$.
     - If $A[mid] > mid$, the search shifts to the lower half. Set $high = mid - 1$.
- If the loop concludes without finding an index, it indicates that no element in the array satisfies $A[i] = i$.

Complexity Analysis

The loop halving the search at each iteration guarantees a logarithmic decrease in problem size, thereby ensuring a time complexity of $O(\log n)$. Each comparison and subsequent adjustment of the search bounds is performed in constant time, aligning with the efficiency requirements of the divide-and-conquer approach.