Algorithms M2–IF TD 4

November 2, 2021

1 Recurrence Relations

Solve the following recurrence relations (it suffices to give an answer in Θ notation).

1. $T(n) = 2T(n/3) + 1$
2. $T(n) = 5T(n/4) + n$
3. $T(n) = 9T(n/3) + n^2$
4. $T(n) = T(n-1) + 2$
5. $T(n) = T(n-1) + n$
6. $T(n) = T(n-1) + 2^n$

2 Median in Two Sorted Arrays

We are given two sorted arrays A, B of sizes n, m. Suppose for simplicity that all their elements are distinct. We are also given an integer k and are asked to return the k-th smallest number of the union of the two arrays.

Clearly, one way to solve the problem is to run the `Merge` procedure on A, B, producing a sorted array C, and output $C[k]$. This will take $O(n+m)$. Give an algorithm for this problem that runs in $O(\log n + \log m)$.

3 Silly-Sort

Consider the following sorting algorithm: given an array A on n elements:

1. If $n \leq 3$ sort A with bubblesort. Otherwise:
2. Sort the array $A[1, \ldots, 2n/3]$
3. Sort the array $A[n/3, \ldots, n]$
4. Sort the array $A[1, \ldots, 2n/3]$

Prove that this algorithm is correct and calculate its complexity.

4 Majority

We are given an unsorted array A of n (not necessarily distinct) integers. We will say that an element x is the "majority" element of A if x appears strictly more than $n/2$ times in A. Of course, A does not necessarily have a majority element. For example if $A = [1, 2, 3, 1, 1]$, then 1 is the majority element, while if $A = [1, 2, 3, 2]$, no majority element exists.

1. Give an $O(n \log n)$-time algorithm which first sorts A to determine if A has a majority element.

   NB: For the remainder of this exercise assume that sorting algorithms cannot be used, because we cannot order elements. That is, the operations $<, >$ do not work, but the operation $==$ does.

2. Give an $O(n \log n)$-time algorithm for the same problem that does not sort A.

3. Give an $O(n)$-time randomized algorithm to determine if A has a majority element and output it.

4. Give an $O(n)$-time deterministic algorithm for the same problem.

5 Semi-sorted Matrix

We are given an $n \times n$ matrix A that contains integers. The matrix is "semi-sorted" in the sense that it obeys the following rules:

- All rows are sorted in increasing order: $A[i, j] \leq A[i, j + 1]$.
- All columns are sorted in increasing order: $A[i, j] \leq A[i + 1, j]$.

We are given an integer x and are asked to either find i, j so that $A[i, j] = x$ or return that x is not in A. In the remainder, let A_1, A_2, A_3, A_4 be the four $(n/2) \times (n/2)$ matrices which are the four quadrants of A

1. Consider the following recursive algorithm: if $n \leq 1$, check if $A[1, 1] = x$; otherwise, recurse on A_1, A_2, A_3, A_4. What is the complexity of this algorithm? Does it work if the matrix is not sorted?

2. Improve on the complexity of the above algorithm by comparing x to $A[n/2, n/2]$ before recursing.

3. Consider the following algorithm: we perform binary search on each row looking for x. What is its complexity?

4. Give a linear-time algorithm for this problem.

5. Show that no algorithm can solve this problem using a sub-linear number of comparisons (you may assume that searching an unsorted array of size n cannot be solved using a sub-linear number of comparisons).

6 Squaring vs Multiplying

Let $T(n)$ be the complexity of the best algorithm for multiplying two n-bit integers (in class we saw that $T(n) = O(n^{\log 3})$, but better algorithms exist). Consider now an easier problem: we are given an n-bit integer a and are asked to calculate a^2. Let $Q(n)$ be the complexity of the best algorithm for this problem.

1. Observe that $Q(n) \leq T(n)$.
2. Observe that $Q(n) = \Omega(n)$.
3. Show that $Q(n) = \Omega(T(n))$. To show this assume for contradiction that $Q(n) = o(T(n))$ and show how you could use a fast squaring algorithm to improve the complexity of the best multiplication algorithm.