Algorithms M2–IF TD 4

November 8, 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. $T(n) = T(n-1) + 2^n$

Solution:

1. $n^{\log_3 2}$ (Master Theorem)
2. $n^{\log_4 5}$ (Master Theorem)
3. $n^2 \log n$ (Master Theorem)
4. $2n$
5. $\Theta(n^2)$
6. $\Theta(2^n)$

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 $O(\log n + \log m)$.

Solution:

We first observe that if $n \leq 1$ (or $m \leq 1$) the problem is easy: we use binary search to insert the unique element of A into B, and thus obtain C in $O(\log m)$ time. We therefore want a procedure that halves n or m in each step.

Consider the following algorithm: we compare $A[n/2]$ to $B[m/2]$. Suppose without loss of generality that $A[n/2] > B[m/2]$. We now have two cases:

1. If $k < n/2 + m/2$, then $A[n/2]$ would be after position k in the sorted array C. As a result, the elements in $A[n/2 + 1, \ldots, n]$ are too large and can be deleted. This decreases the size of A by a factor of 2.
2. If $k > n/2 + m/2$, then $B[m/2]$ would be before position k in the sorted array C. So, the elements in $B[1, \ldots, m/2 - 1]$ are too small and can be deleted. This decreases the size of B by a factor of 2.

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.

Solution:

Correctness can be established by induction on n. The main observation is that after the second recursive call, the $n/3$ largest elements of A will appear in sorted order in positions $A[2n/3 + 1, \ldots, n]$. To see this, we note that after the first recursive call, the $n/3$ largest elements of A are in positions $n/3 + 1, \ldots, n$. Now, assuming that after the second recursive call the $n/3$ largest elements of A are in their correct positions, the third recursive call correctly sorts the rest of the array.

For complexity we have

$$T(n) \leq 3T(2n/3) + O(1)$$

Using the Master theorem we have $a = 3, b = 3/2, d = 0$, so $\log_b a = \frac{\log 3}{\log 3-1} \approx \frac{1.6}{0.6} \approx 2.67$. So the running time is very slow: $O(n^{2.67})$.

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.

Solution:

1. Consider the following algorithm. First we sort A ($(O(n \log n)$ time) and then we execute the algorithm below:

```
counter = 1
for i=2 to n
        if A[i]==A[i-1] then
                counter++
                if counter>n/2 then return A[i]
        else
                counter = 1
return NO
```

   The algorithm clearly runs in $O(n)$ time. Its idea is that the variable `counter` keeps track of the number of times we have seen the current element. If it is different from the previous element, then `counter` gets value 1. Otherwise, the value of the counter is increased by 1 and we check if the current element is already a majority.

2. To achieve the same complexity without sorting, we can use divide&conquer. We first note that if we are given a candidate integer x, checking if x is the majority element of A can be done in $O(n)$ time (count how many times x appears in A).

   Let A_1 be an array that contains the first $\lfloor n/2 \rfloor$ elements of A and A_2 an array that contains the rest. Suppose x is a majority element in A. Then we claim that x must be a majority element of at least one of A_1, A_2. To see this, suppose that x appears at most $|A_1|/2$ times in A_1 and $|A_2|/2$ times in A_2. Then, it appears at most $\frac{|A_1|+|A_2|}{2} \leq \frac{n}{2}$ times in A, contradiction.

   Our algorithm now recursively finds a majority element x_1 in A_1 (if it exists) and a majority element x_2 in A_2. It then verifies if x_1 or x_2 are majority elements of A. The running time is $T(n) \leq 2T(n/2) + O(n)$ which gives complexity $O(n \log n)$.

3. As observed in the previous question, if we have a candidate element x, the problem is easily solvable in $O(n)$. A randomized algorithm could do the following: pick a random element $x = A[i]$, and then check if x is a majority element. If it is, output x. If not, output that no majority element exists.

   Clearly, this algorithm runs in time $O(n)$. If it outputs x, then the output is correct (since we verified that x is a majority element). So we need to calculate the probability that the algorithm incorrectly outputs that no majority element exists. However, this is at most the probability that the selected random element is not the majority element, therefore it is at most $1/2$. So, with probability at least $1/2$ this algorithm is correct. Repeating the algorithm 10 times brings the error probability down to 0.1%.

4. This is trickier. We do the following: for each $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$ we check if $A[2i] == A[2i+1]$. If the two elements are not equal we **delete** both of them from A; if they are equal, we delete one of them from A. This produces a new array A' with size at most $\lceil n/2 \rceil$. We repeat this until the current array has size $O(1)$. Then we run any simple algorithm to find the majority element in the current array, if it exists. We output this element, or NO if it does not exist.

   The easy part here is the running time: this algorithm has complexity $T(n) \leq T(n/2) + O(n)$ which is $O(n)$ (using the Master theorem).

   The interesting part is why this algorithm is correct. The main claim is that if we apply the process above, A' will have a majority element if and only if A does, and that element will be the same number. We need to prove two statements:

   - If x is the majority element of A', then x is the majority element of A. Suppose $|A|$ is even. Then $|A| \leq 2|A'|$, x appears strictly more than $|A'|/2$ times in A', therefore it appears strictly more than

$|A'| \geq |A|/2$ times in A. Similarly, if $|A|$ is odd, then $|A| \leq 2|A'| - 1$, so $|A'| \geq \frac{|A|+1}{2}$. Now, x appears strictly more than $|A'|/2$ times in $|A'|$, so it appears strictly more than $|A'| - 1$ times in A. We have $|A'| - 1 \geq \frac{|A|-1}{2} = \lfloor \frac{|A|}{2} \rfloor$.

- If x is the majority element of A, then x is the majority element of A'. The proof is similar to the previous part, for simplicity let's assume $|A|$ is even. We observe that every time the algorithm deletes two elements, this does not affect the majority element (that is, if x was a majority element, it remains so). So we only need to focus on the pairs for which $A[2i] = A[2i+1]$. Suppose there are a such pairs where $A[2i] = A[2i+1] = x$ and b such pairs where $A[2i] = A[2i+1] \neq x$. Clearly, $a > b$ (otherwise x is not a majority element). But x appears a times in A', and all other elements together appears at most b times.

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).

Solution:

1. The algorithm gives $T(n) = 4T(n/2) + O(1)$. Using the Master theorem we have $T(n) = O(n^2)$. The algorithm does not use the fact that A is semi-sorted, so it's natural that its complexity is linear in the size of A (it will check all elements).

2. We observe that all elements of A_1 are smaller or equal to $A[n/2, n/2]$ and all elements of A_4 are greater or equal to $A[n/2, n/2]$. So, if $x < A[n/2, n/2]$ we know that A_4 does not need to be searched, while if $x > A[n/2, n/2]$, A_1 does not need to be searched. We have $T(n) = 3T(n/2) + O(1)$, which gives $T(n) = O(n^{\log 3})$.

3. This runs in time $O(n \log n)$.

4. This is trickier. Consider the following algorithm:

```
i=n, j=1
while i>=1 and j<=n do
        if A[i,j]==x then return i,j
        if A[i,j]>x then i=i-1
        else j=j+1
return NO
```

   It is clear that this algorithm will terminate after $O(n)$ steps. To see correctness, we will prove that following invariant: for all i', j', if $i' > i$ or $j' < j$, then $A[i', j'] \neq x$. In other words, $A[i, j]$ is the current element, and everything that is to the left or below it in the matrix cannot contain x. This is clearly true in the beginning, since we start at the bottom-left corner. Then, when we decrease i, we have $A[i, j] > x$, so all element in row i which are to the right of $A[i, j]$ cannot be x. Similarly, when we increase j, we have $A[i, j] < x$, so all elements in column j above $A[i, j]$ cannot be x.

5. Let B be an unsorted array of n elements with minimum value a and maximum value b. Suppose we are looking for x in B. We construct a semi-sorted array A as follows: we put the elements of B in the secondary diagonal from $A[n, 1]$ to $A[1, n]$; we put value $a - 1$ on the upper-left half of A; and value $b + 1$ on all positions of the bottom-right half.

   Now, A is semi-sorted. If we have an algorithm that can find x in A using $o(n)$ comparisons, using the above procedure we can conclude if x belongs in B with the same 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.

Solution:

1. One way to calculate a^2 is to use the standard multiplication algorithm to calculate $a \times a$, obviously.

2. Every algorithm for squaring must read all the digits of the input.

3. Suppose that we have an algorithm for squaring n-bit integers running in time $Q(n) << T(n)$. We will show how to use this to multiply two n-bit integers a, b in time $Q(n) + O(n) = O(Q(n)) = o(T(n))$. This is a contradiction, since we assumed that $T(n)$ is the best possible for calculating products.

   Given a, b we calculate $(a + b)$ ($(O(n)$ time), then $x = (a + b)^2, a^2, b^2$ $(O(Q(n)))$ time. We observe that $(a + b)^2 = a^2 + b^2 + 2ab$. We therefore calculate $y = x - a^2 - b^2$ $(O(n))$ time. Finally, we divide y by 2 $(O(1))$ time. The total time used is therefore $O(Q(n))$.