CS5800 Algorithms

Recitations from Sep 23, 2019 to Sep 27, 2019

Ravi Sundaram

Problem 1 (Recurrences)

1. $T(n) = 3T(n/2) + n$ (This is the recursion for Karatsuba's Algorithm)
2. $T(n) = 3T(n/2) + n^2$
3. $T(n) = 2T(n/4) + T(n/2) + n$ (Hint: Think of the number of nodes at each level. You can determine it by solving a recurrence).
4. $T(n) = 5T(n/4) + n$

Solution:

1. At level i of the recursion tree, there are 3^i nodes, and each node has the amount of work $\frac{n}{2^i}$, so we have $(3/2)^i * n$ amount of work at level i. This is an increasing geometric series, so it's dominated by the last level. There are $\log_2 n$ levels, so there are $3^{\log_2 n} = n^{\log_2 3}$ nodes at the last level so $\Theta(n^{\log_2 3})$.
   By the Master Theorem, we note that $(3/2) \cdot n > 1 \cdot n$, and we get the same $\Theta(n^{\log_2 3})$.
2. At level i of the recursion tree, there are 3^i nodes, and each node has the amount of work $(n/2^i)^2$, so we have $(3/4)^i * n^2$ amount of work at level i. This is a decreasing geometric series, so it's dominated by the first level. We simply get $\Theta(n^2)$
   By the Master Theorem, we note that $(3/4) \cdot n^2 < 1 \cdot n^2$, and we get the same $\Theta(n^2)$.
3. At each level we get exactly n amount of work just based on the formula (if not clear, expand out the recursion tree and see what happens). The height of the recursion tree is $\log_2 n$, so we get $\Theta(n \log_2 n)$
4. This is similar to first problem and we get $\Theta(n^{\log_4 5})$

■

Problem 2 (Divide and Conquer)

1. Give an $O(\log n)$ algorithm to find whether there is an index i in a sorted array A (that consists of all distinct elements) such that $A[i] = i$.
2. You're given an array A (of integers) as input. Write an algorithm that returns the number of inversions in the array. We say there's an inversion between indices $i < j$ if $A[i] > A[j]$.
3. Give a fast algorithm that given an array A of n integers, finds two indices $i < j$ such that $A[j] - A[i]$ is maximized.

Solution:

1. We can solve this using basically binary search. We look at the middle element. If its larger than its index, then we know we can ignore the right half of the array (because the array is sorted and all the elements are distinct, so the elements increase at least as much as the indices do). So we recurse on the left half of the array. If the middle element is smaller than its index, then we recurse on the right half of the array. At each step we perform a constant time operation, and we are reducing the size of the input by half so we get $O(\log n)$ run time.

2. Our solution will follow the same structure as merge sort. But when we are merging two sorted subarrays L and R, we count the number of inversions between elements in L and elements in R, andd add that to the number of inversions in L and the number of inversions in R (that's already computed in the previous recursive step). Here's a code snippet for the merge step. Array M is the sorted array that we get after merging L, R.

```
merge(array L, array R, array M)
{
  int i,j,k,inversion_counter = 0
  while (i < L.length() AND  j < R.length()) {
    if(L[i] <= R[j]){
      M[k] = L[i]
      i++
      k++
    }
    if(L[i] > R.length()){
      inversion_counter = inversion_counter + (L.length() - i)
      M[k] = R[j]
      j++
      k++
    }
}
```

   Note that at each step, we either increment i or j, so this extra step of counting inversions is linear. The merging of two arrays is linear as well, and since we have two independent linear operations the whole thing is still linear. The logic works because L, R are sorted. If $L[i] > R[j]$, all the elements in L that are to the right of index i are larger than elements $R[j]$, so we have that many inversions to count. After this, we can increment j. If $L[i] \leq R[j]$, then we know $L[i]$ is less than all the element to the right of $R[j]$, so we can increment i. At each step of recursion we are halving the input, and we have a linear amount of work at each level so the total runtime is $\Theta(n \log n)$

3. In our divide step, we split the array into two subarrays. For our conquer step, let L, R denote the left subarray and right subarray respectively that we are trying to merge. Let M_L denote the max value $L[j] - L[i]$ for some indices $i < j$, and let M_R denote the max value $R[k] - R[l]$ for some indices $k < l$. Let s denote the index of the minimum element in L, and let t denote the index of the maximum element in R. Then we take the value $max(M_L, M_R, R[t] - L[s])$ when we merge. We split the array in half at each step, and perform a linear scan to find the min/max in L, R, so the runtime of this is $\Theta(n \log n)$.

■

Problem 3 (Multiplication)

1. Say you have two complex numbers $a+ib$ and $c+id$. Their product is $(ac-bd)+i(ad+bc)$. The straightforward method to calculate takes 4 multiplications. Find a way to compute the product using just three multiplications.

2. Explain how to multiply two numbers of 4 digits each in base B using only M multiplications of 1-digit numbers in base B, and a constant number of additions in base B. Make M as small as you can (Hint: Remember how Karatsuba works).

Solution:

1. Let $P_1 = ac$, $P_2 = bd$, and $P_3 = (a+b)(c+d)$. Then $ac-bd = P_1 - P_2$, and $ad+bc = P_3 - P_1 - P_2$.

2. We run the Karatsuba recursion, so we split the four digits into two, and perform 3 multiplications, then we split the two digits into one digits and perform 3 multiplications for each of those pairs. So in total we get $3 \cdot 3 = 9$ multiplications.

■

Problem 4 (Extras...)

1. Explain the closest pair of points algorithm for 2-D, and why it's correct. Remember, the key part is the conquer step.