TCSS 343 - Week 4 - Wednesday

Jake McKenzie

February 12, 2019

Master Method and some more Divide and Conquer

"What we choose means more than what was handed to us by chance".

...

Ada Palmer

"Great perils have this beauty, that they bring to light the fraternity of strangers".

...

Victor Hugo

"Truth persuades by teaching, but does not teach by persuading".

...

Tertullian

1. In this problem use the Master Theorem to find and prove tight bounds for the following recurrence.

$$T(n) = \begin{cases} \text{c} & \text{if } n \leqslant 1 \\ 9T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + 27n & \text{if } n > 1 \end{cases}$$

$27n \in ?\left(n^{\log_3 9 \pm e}\right)$

$27n \in O\left(n^{2-1}\right)$ $\varepsilon = 1$

case 1

$T(n) \in \Theta\left(n^2\right)$

2. In this problem use the Master Theorem to find and prove tight bounds for the following recurrence.

$\frac{n}{\frac{7}{2}}$ $b = \frac{7}{2}$

$$T(n) = \begin{cases} \text{c} & \text{if } n \leqslant 1 \\ 2T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + 3n & \text{if } n > 1 \end{cases}$$

7

$\log_{\frac{7}{2}} 2 \approx$ 0.553294755665...

$3n \in ?\left(n^{0.55\ldots \pm \varepsilon}\right)$

$3n \in \Omega\left(n^{0.55\ldots + .1}\right)$

check normalization

$\cancel{3}\left(2\left(\frac{n}{\frac{7}{2}}\right)\right) \leq d(\cancel{3}n)$

$\frac{4}{7}\cancel{n} \leq d\cancel{n}$

$\frac{4}{7} \leq d$ $d = \frac{1}{7}$

Case 3

$T(n) \in \Theta(n)$

3. In this problem use the Master Theorem to find and prove tight bounds for the following recurrence.

$$T(n) = \begin{cases} \text{c} & \text{if } n \leqslant 1 \\ 70T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 12n^3 & \text{if } n > 1 \end{cases}$$

$\log_4 70 \approx 3.06464...$

$12n^3 \in ?(n^{3.06464... \pm \varepsilon})$

$12n^3 \in O(n^{3.06464... - 0.01})$ $\varepsilon = 0.01$

$12n^3 \in O(n^{3.06464...})$

case 1

$T(n) \in \Theta(n^{3.06464...})$

4. Design an algorithm that accepts an unsorted array of integers and finds the subarray with the maximum possible sum.

   For example, consider the array $\{2, -4, 1, 9, -6, 7, -3\}$. The maximum subarray would be $\{1, 9, -6, 7, -3\}$, which sums to 11.

   A naive solution that considers every possible subarray would take $O(n^2)$ time. Design a more efficient algorithm that uses divide and conquer and runs in $O(n \log n)$ time.

```java
class GFG {

    // Find the maximum possible sum in arr[]
    // such that arr[m] is part of it
    static int maxCrossingSum(int arr[], int l,
                                int m, int h)
    {
        // Include elements on left of mid.
        int sum = 0;
        int left_sum = Integer.MIN_VALUE;
        for (int i = m; i >= l; i--)
        {
            sum = sum + arr[i];
            if (sum > left_sum)
            left_sum = sum;
        }

        // Include elements on right of mid
        sum = 0;
        int right_sum = Integer.MIN_VALUE;
        for (int i = m + 1; i <= h; i++)
        {
            sum = sum + arr[i];
            if (sum > right_sum)
            right_sum = sum;
        }

        // Return sum of elements on left
        // and right of mid
        return left_sum + right_sum;
    }

    // Returns sum of maxium sum subarray
    // in aa[l..h]
    static int maxSubArraySum(int arr[], int l,
                                    int h)
    {
    // Base Case: Only one element
    if (l == h)
        return arr[l];

    // Find middle point
    int m = (l + h)/2;

    /* Return maximum of following three
    possible cases:
    a) Maximum subarray sum in left half
    b) Maximum subarray sum in right half
    c) Maximum subarray sum such that the
    subarray crosses the midpoint */
    return Math.max(Math.max(maxSubArraySum(arr, l, m),
                    maxSubArraySum(arr, m+1, h)),
                    maxCrossingSum(arr, l, m, h));
    }

    /* Driver program to test maxSubArraySum */
    public static void main(String[] args)
    {
    int arr[] = {2, 3, 4, 5, 7};
    int n = arr.length;
    int max_sum = maxSubArraySum(arr, 0, n-1);

    System.out.println("Maximum contiguous sum is "+
                                    max_sum);
    }
}
```

5. Given an array containing elements of type E design an algorithm that finds the majority element – that is, an element that appears more then $\frac{n}{2}$ times. If no majority element exists, return null.

   Your algorithm should run in $O(n \log n)$ time (and use only $O(1)$ extra memory).

The $O(n \log n)$ solution works by first splitting the array into two halves. We recurse on both halves and receive back the majority elements for the two halves (if they exists)

Once we finish recursing, there are four different scenarios:

1. The two subarrays have the same majority element.

This means, by definition, that element must also be the majority of the full array.

Why is this? Suppose that there are n elements in the overall array. If A is the majority of the left half, then that means that by definition, there must be $> \frac{n}{4}$ occurrences of A on the left. Similarly, if A is the majority on the right, there must be $> \frac{n}{4}$ occurrences there. Therefore, there must be $> \frac{n}{2}$ occurrences of A overall. So we can just return A without needing to check anything else.

2. The two subarrays have different majority elements.

In that case, we need to figure out which one is the true majority. We take the majority element from the left and loop over the entire array to figure out how many times it appears. We do the same thing with the majority element from the right. This will take O(2n) = O(n) time.

If either of them appear more then $\frac{n}{2}$ time, return that element as the majority. If neither of them appear enough times, return null (or whatever else we're using to indicate that there's no majority).

3. Only one subarray has a majority; the other doesn't.

We do the same sort of looping thing as before, again in O(n) time

4. Neither subarrays have a majority.

We can automatically give up here, for basically the same reason why we could automatically return in case 1.

We end up doing 2T(n) + n work in the worst case in the recursive case, which results in O(n log(n))

6. Suppose you are trying to write a video game containing thousands of different moving elements and want to check if two elements have collided or overlapped.

   A naive way of implementing this would be to use two nested loops and check every pair of elements. This often ends up being too inefficient for most video games, even for only a few thousand elements (especially games that require a high degree of responsiveness).

   Describe how you would design a data structure to store these points in a way that lets you more efficiently check whether two elements are colliding.

   For the sake of simplicity, you may assume that each element is a circle and has a relatively small radius. You may also assume that the elements are moving on a 2d plane – you don't need to worry about collisions in 3d.

   As a hint: think about recursively subdividing the 2d plane.

We can solve this by creating a kind of a data structure known as a "quadtree" – in this case, probably using a variation known as a "region quadtree".

A "quadtree", as its name suggests, is a kind of tree. Each branch node (in a region quadtree) represents one rectangular region within the 2d plane. A branch node also has at most four children that contain all elements located within the upper-left, upper-right, lower-left, and lower-right corners of that rectangle respectively.

Each child can either be another branch node (which contains four more children), or a leaf node (representing a single gameplay element).

Now, suppose we want to insert a new element into an existing quadtree. To do so, we run the following algorithm:

1. We assume the root node is a branch node. We take the coordinate of the point we want to add and determine which of the four quadrants it belongs in.

2. This branch node could either be null, point to a leaf node, or point to another branch node.

a. If it's null, add a new leaf node for that coord there

b. If it's a leaf node, replace the leaf node with a branch node containing both the point that was originally there, as well as the new one.

c. If it's a branch node, recurse and repeat this entire procedure.

This ends up forming a tree, where the regions that contain a lot of points end up being deeply nested, and the regions with few points end up being shallow.

This data structure also lets us search for all points within a certain bounding relatively efficiently: since each branch node stores information about what region in the 2d plane its supposed to contain, we can recursively search and find all leaf nodes that fall within those coordinates without having to search through every single existing point.

Now, to check if a given point is colliding with any other one, we no longer need to compare it against every single other point. Instead, we just find the leaf node corresponding to our point, move up a level or two, and look at all of the children to get the list of all points that happen to be close by.

The core idea is we were able to speed up traversal by recursively dividing up our points into different regions based on their x-y coordinates in the planes.

7. Give an efficient algorithm to compute the binomial coefficient $\binom{n}{k}$, reminder: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Use the recursive definition below to help you along.

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{if } 0 < k < n \end{cases}$$

What is the time complexity of your algorithm? Is it a polynomial time algorithm? Explain.

```
class GFG {

  // Returns value of Binomial
  // Coefficient C(n, k)
  static int binomialCoeff(int n, int k)
  {

    // Base Cases
    if (k == 0 || k == n)
      return 1;

    // Recur
    return binomialCoeff(n - 1, k - 1) +
          binomialCoeff(n - 1, k);
  }

  /* Driver program to test above function */
  public static void main(String[] args)
 {
    int n = 5, k = 2;
    System.out.printf("Value of C(%d, %d) is %d ",
                n, k, binomialCoeff(n, k));
  }
}
```