CS 344 Problem Set 2: Divide and Conquer Algorithms

Kanghwi Lee

1. Let T(n) = T(an) + T(bn) + n for some constants $0 < a \leq b < 1$. Show that :

(a) T(n) = Θ(n) if a + b < 1

(b) T(n) = Θ(n log n) if a + b = 1

(a) Examining the recurrence's time complexity We may visualize the recurrence tree and track the expenses at each level if T(n) = T(an) + T(bn) + n.

The cost is n at the highest level.

The total cost at this level is T(an) + T(bn) since there are two subproblems of sizes a and bn at the following level.

Costs for the i-th level are T(ai * n) and T(bi * n), continuing the pattern for successive levels.

After log_a(n) levels, when we arrive at a subproblem of size 1, the recurrence tree's smallest branch appears.
After log_b(n) levels, the recurrence tree's largest branch appears when we arrive at a subproblem of size 1.

We must add up the costs at each level in order to determine the overall cost in an asymptotic sense. We have a total of O(log_a(n)) levels because the shortest and longest branches define the number of levels.

As a result, the overall expense is roughly:

T(n) = n + (T(an) + T(bn)) + (T(a^2 * n) + T(b^2 * n)) + ... + (T(a^log_a(n) * n) + T(b^log_b(n) * n))

= n * (1 + a/b + a^2/b^2 + ... + a^log_a(n)/b^log_b(n))

= n * (1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_a(n))

= n * (1 + (a/b) + (a/b)^2 + ... + (a/b)^log_b(n))

n * (1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n)))

= n * (1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n)))

= n * (1 + (a/b)(1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n)-1)))

The geometric series (1 + (a/b) + (a/b)2 +... + (a/b)(log_b(n)-1))) adds to a constant number since a/b 1. T(n) = O(n), hence we can draw that conclusion.

(b) In a similar manner, the total cost can be roughly calculated as follows:

T(n) = n * (1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n))) = n * (1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n))) = n * (1 + (a/b)(1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n)-1)))

Since a/b = 1 - a, the geometric series (1 + (a/b) + (a/b)^2 + ... + (a/b)^(log_b(n)-1)) sums up to log_b(n). Therefore, we can conclude that T(n) = O(n log n).

2. (Merging sorted lists) Suppose you are given k sorted arrays, each having n elements (integers). The objective is to combine all these sorted arrays to form one large sorted array having kn elements. Let's call the sorted arrays A1, A2, ..., Ak for convenience.

(a) The temporal complexity can be calculated as follows if the sorted arrays are iteratively combined using the merge procedure:

A1 and A2 must be combined in O(n) time.

Time required to combine B1 and A3 is O(2n) = O(n).

It takes O(3n) = O(n) time to merge B2 and A4.

It takes O((k-1)n) = O(n) time to combine B(k-2) and Ak.

It takes O(kn) = O(n) time to merge B(k-1) and Ak+1.

As a result, the overall time complexity is O(nk2), which is equal to O(n + 2n + 3n +... + kn).

(b) We can more effectively merge the k sorted arrays using the divide and conquer strategy.

First, using the divide and conquer strategy, we split the k arrays into two parts and recursively merge them.

Then, in O(n) time, we use the merge algorithm to combine the two parts.

The recurrence relation determines the time complexity of joining the two halves:

T(k) = O(n) + 2T(k/2)

The temporal complexity of this strategy is O(kn log k), according to the Master theorem or the recurrence tree method.

3. (Median from sorted halves) You are given as input two sorted arrays A and B having n 2 elements (integers) each. Consider the task of finding the n 2 -th smallest element in the union of A and B. For example, if the input arrays are [1 3 5 7 9 11] and [2 26 48 82 100 164] then the answer must be 9.

(a) By using a straightforward procedure, we may determine the second-smallest element in the union of the sorted arrays A and B.

Two pointers should be initialized: one for array A and one for array B.

Examine the components at the pointers.

The smaller element's cursor is moved forward.

Up until the second-smallest element, repeat steps 2 and 3 as necessary.

We can compare elements across arrays and delete elements bigger than the second-smallest element because both arrays are ordered. The algorithm will stop working after a fixed number of comparisons, taking O(n) amount of time to complete.

(b) We can solve this issue in O(log n) time by using the divide and conquer strategy.

Find the medians of the a and b arrays, which we'll refer to as median_A and median_B.

The second-smallest element is the case where median_A == median_B.

Discard the elements in arrays A and B that are less than or equal to median_A and those that are more than or equal to median_B if median_A > median_b. Repeat with the remaining components.

Discard the elements in arrays B and A that are more than or equal to median_A and less than median_B, respectively, if median_A > median_b. Repeat with the remaining components.

The issue size decreases to roughly half in each step by discarding half of the elements after each recursion, leading to an O(log n) time complexity.

4. (Majority element) An array A of n elements is said to have a majority element if strictly more than half of its entries are the same. For example, the array [13 23 13 47 31 13 13] has a majority element – 13, whereas the array [20 40 20 13 7 40] has no majority element. On input array A provide algorithms according to the following requirements that either output the majority element (if one exists) or returns that none exists.

(a) We can use two nested loops to compare each pair of elements in array A in order to calculate the frequencies of all the elements. The inner loop counts how many times each element appears while the outer loop iterates through each element individually. It takes this method O(n2) time.

By monitoring the element with the highest frequency and determining whether it appears more than n/2 times, we can expand this technique to determine the majority element. O(n2) time is still required for this extension.

(b) We can calculate the frequencies of all elements in O(n log n) time by employing a more effective technique.

Use a counter variable to tally the frequency of each member as you iterate through the sorted array A.

To identify the majority element, look for any element that occurs more than n/2 times.

The sorting phase takes O(n log n) of time and dominates the total time complexity.

5. (Range queries) Consider an array A having n integers. We define middle half of A to be the elements of A having ranks in the range [n 4 , 3n 4] (both inclusive). For example, the middle half of the array [20 13 50 2 38 21 -3 59] is the set {20, 13, 2, 38, 21}.

(a) The techniques below can be used to return the center half of an array A of n integers in O(n log n) time.

An array's sorting takes O(n log n) time.

The elements from index n/4 to 3n/4, inclusive, should be returned.

It takes O(n log n) time to sort, and O(n/2) = O(n) time to return the middle half. The total temporal complexity is, therefore O(n log n).

(b) A faster approach that returns the middle half in O(n) time exists:

Use the quick select algorithm or any linear-time median finding approach to find the array A's median in O(n) time.

Divide the elements of the array A around the median into those that are less than or equal to it and those that are greater than it. In O(n) time, this is possible.

Repeat step 2 recursively on the larger partition if the two partition sizes are both higher than n/2.

Repeat step 2 recursively on the smaller partition if the two partitions' sizes are both less than n/2.

We have located the middle half if one division is larger than n/2 and the other is smaller than n/2.

The time complexity is O(n) since the partitioning step halves the size of the issue in half for each iteration.

6. (Extra Credit) (Out of order pairs) You are given an array A of n integers. We call a pair of elements (ai , aj) to be out of order if i < j and ai ≥ aj . For example, the array [20 13 50 2 38 21] has 7 out of order pairs - (20,13), (20,2), (13,2), (50,2), (50,38), (50,21) and (38,21).

1. (a) We can use two nested loops to compare each pair of elements in array A, which contains n numbers, to count the number of out-of-order pairs. The inner loop iterates from the subsequent element to the final element whereas the outer loop loops backwards from the initial element to the second-to-last element. Increase a counter variable if a pair that is out of order is discovered. It takes this method O(n2) time.

(b) With a temporal complexity of T(n) = 2T(n/2) + O(n log n), we can count the number of out of order pairs more effectively by using the divide and conquer method.

1. Create two equal-sized halves of the array A.

2. Count the number of out-of-order pairs in each half iteratively.

3. Add the counts together by adding the out-of-order pairs that contain components from both halves.

4. Give me the overall tally.

According to the provided recurrence relation, the time complexity is O(n log n) by halving the size of the problem in each recursion and aggregating the results.

More details for the pset2

1. (a) Divide and conquer is represented by the recurrence relation T(n) = T(an) + T(bn) + n that is given. We consider the recurrence tree to examine its time complexity. The cost is equal to the sum of the costs of the two recursive calls (T(an) and T(bn)) plus an additional cost of n, at each level. Because the size of the problem shrinks by a factor of either an or b with each recursive iteration, we note that the tree has a total of O(log n) levels.

We can observe that the price of each level reduces geometrically in the scenario where a + b 1. As a result, the top-level cost, which is n, will account for the majority of the total cost across all levels. T(n) therefore, equals O(n).

The cost of each level remains constant in the scenario when a + b = 1, resulting in a total cost of O(log n) levels times a fixed value of n. T(n) therefore, equals O(n log n).

2. (a) Pairwise merging of the sorted arrays creates a total of k/2 merged arrays in the iterative merging method. It takes O(n) time to combine two arrays of size n. As a result, O(nk2) is the global time complexity.

(b) The k-sorted arrays can be merged more quickly using the divide-and-conquer strategy. Recursively splitting the arrays into halves and merging them allows us to reach an O(kn log k) time complexity. Thus, there are O(log k) layers and merging k arrays at each level requires O(n) time per level.

3. (a) We can loop through the arrays using two pointers and compare elements until we get the second-smallest member in the sorted arrays A and B union. Because we perform a fixed number of comparisons, this method has a linear time complexity of O(n).

(b) We can delete elements bigger than the second-smallest element by iteratively comparing the medians of the two arrays. As a result, the problem size is reduced by half with each recursion, yielding a logarithmic time complexity of O(log n).

4. (a) The brute-force method has an O(n2) time complexity for calculating the frequencies of the array's items. By extending this technique to see if any element appears more than n/2 times, we can identify the majority of element with the same O(n2) time complexity.

(b) We can obtain an O(n log n) time complexity for computing the frequencies by first sorting the array in O(n log n) time and then linearly counting the frequencies. Due to this, we can also calculate the majority element in O(n log n) time.

5. (a) It is possible to return the middle half of array A by sorting the array in O(n log n) time and then choosing the elements in the middle, which results in an O(n log n) time complexity.

(b) Nevertheless, there is a more effective algorithm that, without sorting, instantly determines the center half. Locating the elements with ranks in the range's center half can be done in linear time O(n).

6. (a) The brute-force method, which has an O(n2) time complexity, compares every pair of elements to count the number of out-of-order pairs in array A.

(b) We can count the number of out-of-order pairings more quickly by employing the divide-and-conquer strategy. The temporal complexity is represented by the recurrence relation T(n) = 2T(n/2) + O(n log n), where each recursion requires splitting the array in half and combining the results in O(n log n) time. By doing so, we can produce an algorithm that is more effective than the brute-force method.