Divide and Conquer

Rana Barua

Visiting Scientist, IAI, TCG CREST Kolkata

1 Divide and Conquer

In the Divide and Conquer paradigm, the original problem is sub-divided into smaller problems which are solved recuresively, and finally the solutionas are combined to obtain a solution of the original problem. We shall illustrate this with three different types of problem *viz* **Mergesort, Counting Inversions** and finding the **Closest Pair** of points. We first consider Merge Sort.

1. **Merge Sort**
   This is a sorting problem in which we are given a sequence of numbers and we need to sort it into a non-decreasing sequence.
   In *mergesort* we are given a sequence of numbers $x_1, x_2, \ldots, x_n$. We first divide the sequence into two sequences of almost equal lengths. We recursively sort the two smaller sequences and then "**merge**" them to obtain a single sorted sequence. For simplicity, we assume that n is a power of 2.
   Mergesort makes use of two procedures. The first procedure is **MERGE(**S, T**)**, that takes two sorted sequences S and T as input, and output a sequence consisting of the elements of S and T in a sorted order. It works by repeatedly selecting the larger of the largest elements remaining on S and T and then deleting the element selected. Ties may be broken in favour of S. Since both S and T are sorted, this procedure requires at most $|S| + |T| - 1$ comparisons.
   Our next procedure is $SORT(i, j)$ which sorts the subsequence $x_i, \ldots, x_j$. The procedure is described below. Here also we assume that the length of the subsequence is 2^k, for some integer $k \geq 0$.
   Procedure $SORT(i, j)$.
   ```
       if i = j then return x_i
       else
           begin
             m ← (i + j − 1)/2;
             S ← SORT(i, m);
             T ← SORT(m + 1, j);
             return MERGE(S, T)
           end
   ```
   Complexity: Let $T(n)$ denote the number of comparison required by mergesort for a sequence of length n. Then we have the following recurrence

   $$T(n) = \begin{cases} 0 & \text{if } \ n = 1 \\ 2T(n/2) + (n-1) & \text{if } \ n > 1 \end{cases}.$$

   It is not hard to see that the solution of this recurrence is $T(n) = O(n \log n)$.

   Exercise 1. (a) Write a pseudo-code of the procedure $MERGE$.
   (b) Give a formal solution of the above recurrence relation.
   (c) In the general case, $T(n)$ satisfies the following in the worst case.

   $$T(n) \leq \begin{cases} 0 & \text{if } \ n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } \ n > 1 \end{cases}.$$

   Show that $T(n) \leq n \lceil \log n \rceil$.

2. **Counting Inversions**

Our next example of the divide-and-conquer paradigm is the problem of counting *inversions* in a permutation.

Definition 1. *Let $A[1..n]$ be an array of n distinct numbers or elements from a linearly ordered set. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.*

Clearly, for a sorted array the count is 0.

Algorithm SORT-AND-COUNT(L).
Input: A list or array. L
Output: The number of inversions in L and L in a sorted order.
    If L has one element
        then return $(0, L)$
    else Divide the list L into two halves A and B
        $(r_A, A) \leftarrow$ SORT-AND-COUNT(A)
        $(r_B, B) \leftarrow$SORT-AND-COUNT(B)
        $(r_{AB}, L) \leftarrow$ MERGE-AND-COUNT(A, B).
    return $(r_A + r_B + r_{AB}, L)$.

How do we combine the two subproblems? The following procedure counts the number of inversions (a, b) with $a \in A$ and $b \in B$, assuming that A and B are sorted.

Procedure MERGE-AND-COUNT(A, B)

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$ then a_i not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A. Increase the count of inversions by $|A|$.
- Append the smaller element to the sorted list C.

Theorem 1. *The SORT-AND-COUNT algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.*

Proof. The worst-case running time $T(n)$ satisfies the recurrence

$$T(n) = \begin{cases} 0 & \text{if } \ n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } \ n > 1 \end{cases}.$$

The solution of this recurrence is $O(n \log n)$. □

Exercise 2. (a) Prove the correctness of the algorithm SORT-AND-COUNT.
(b) Prove Theorem 1.

3. **Closest Pair of Points**

We now consider the problem of finding a closest pair of points in a set Q of $n \geq 2$ points in a plane. A brute-force algorithm will clearly take $O(n^2)$ time. Here we will present a divide-and-conquer algorithm that takes $O(n \log n)$ time.

1.1 The Divide-And-Conquer Algorithm

Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and two arrays X and Y, each of which contains all the points of P. The array X is sorted according to a monotonically increasing x-coordinate. Similarly, the array Y is sorted by monotonically increasing y-coordinate. Note that we cannot afford to sort in each recursive call of the algorithm; since, otherwise the running time would be $O(n \log^2 n)$ (*Exercise*).

A given recursive invocation with inputs P, X, Y first checks if $|P| \leq 3$. If so, the the recursive invocation simply uses the brute-force method. Otherwise, the recursive invocation carries out the divide-and-conquer paradigm as follows.

- **Divide:** Find a vertical line ℓ that divides the set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil, |P_R| = \lfloor |P|/2 \rfloor$ and all points in P_L are on or to the left of the line ℓ and all points in P_R are on or to the right of ℓ. Divide the array X into two arrays X_L and X_R that contains the points of P_L and P_R respectively, each sorted by monotonically increasing x-coordinate. Similarly, divide the array Y into two arrays Y_L and Y_R containing the points of P_L and P_Y respectively, sorted by monotonically increassing y-coordinate.
- **Conquer:** We now make two recursive calls, one to find the closest pair of points in P_L and the other to find the closest pair of points in P_R. The inputs to the first call are the set P_L and the arrays X_L and Y_L, while the inputs to the other call are P_R, X_R and Y_R. Let the closest-pair distaces returned for P_L and P_R be δ_L and δ_R respectively. Let $\delta = min\{\delta_L, \delta_R\}$.
- **Combine:** The closest pair is either the pair with distance δ found by one of the recursive calls or it is a pair of points with one point in P_L and the other point in P_R and whose distance is less that δ. The algorithm will find if there is a pair of points with one point in P_L and the other point in P_R and whose distance is less than δ. Note that if such a pair exists then both the points must be within δ units of the line ℓ. Thus both the points must lie in the 2δ-vertical strip centered at the line ℓ. To find such a pair, if one exists, do the following.
  i. Form an array Y' from Y by removing all the points that are not in the 2δ vertical strip. The array Y' is sorted w.r.t the y-coordinate.
  ii. For each point p in the array Y' try to find points in Y' that are within δ units of p. We shall show below that only 7 points in Y' that follow p need to be considered. Compute the distance from p to each of these 7 points and keep track of the closest distance δ' found over all pairs of ponits in Y'.
  iii. If $\delta' < \delta$, then the veritical strip does contain a closer pair of points than those returned by the recursive calls. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ returned by the recursive calls.

We now show the correctness of this algorithm

Correctness. Suppose at some stage of the recursion, the closest pair is $p_L \in P_L$ and $p_R \in P_R$. and their distance $\delta' < \delta$.The point p_L must be on or to the left of the line ℓ and is less that δ units away from ℓ. Similarly p_R is on or to the right of ℓ and is less that δ units away. Moreover, p_L and p_R cannot be more than δ units apart vertically. Thus p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at ℓ.

We now show that at most 8 points of P can lie within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square to the left of the line ℓ. If 5 points of P lie within this square, then at least two points would be in a $\delta/2 \times \delta/2$ sub-square, and their distance would be $\leq \delta/\sqrt{2} < \delta$, a contradiction. Thus at most 4 points of P_L can reside within this square. Similarly, at most 4 points pf P_R can reside within the square to the right of ℓ. Thus at most 8 points of P can lie within the $\delta \times 2\delta$ rectangle.

Assuming that the closest pair is p_L and p_R and that p_L precedes p_R in Y', even if p_L occurs early and p_R occurs late, p_R is in one of the 7 positions following p_L. This completes the correctness proof.

1.2 Implementation and Running Time

Our main aim is to have the recurrence for the running time to be

$$T(n) = 2T(n/2) + O(n),$$

where $T(n)$ is the running time for a set of n points. The crucial observation is that in each recursive call, we need to construct a sorted subset of a sorted array. For instance, a particular invocations receives a subset and an array Y, sorted by y-coordinate. Having partitioned P into P_L and P_R, we need to form the arrays Y_L and Y_R, which are sorted by y-coordinate in linear time. This can be done by the following procedure. Let l be the length of the array Y.

1. Let $Y_L[1, \ldots, l_1]$ and $Y_R[1...l_2]$ be the new arrays.
2. $l_1 = l_2 = 0$
3. **for** $i = 1$ **to** l **do**
4.   **if** $Y[i] \in P_L$ **then**
5. $l_1 \leftarrow l_1 + 1$
6. $Y_L[l_1] = Y[i]$
7.   **else** $l_2 \leftarrow l_2 + 1$
8. $Y_R[l_2] = Y[i]$

We simply scan the array Y in order. If a point $Y[i] \in P_L$, we simply append it to the end of Y_L; otherwise $Y[i]$ is appended to the end of array Y_R. In the first step, we **presort** the points *i.e.* we sort the points once and for all and then pass on these sorted arrays during the first recursive call. Presorting adds an $O(n \log n)$ term to the running time. But each recursive call now only takes linear time. If $T(n)$ is the running time for the recursive call and $T'(n)$ is the running time for the entire algorithm, then we have

$$T'(n) = T(n) + O(n \log n),$$

and

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } \ n > 3 \\ O(1) & \text{if } \ n \leq 3 \end{cases}.$$

Thus $T(n) = O(n \log n)$ and so $T'(n) = O(n \log n)$. □

Another example: Divide the set of n points in $\Theta(n)$ time into two subsets; one contaning the leftmost $\lceil n/2 \rceil$ points and the other contaning the rightmost $\lfloor n/2 \rfloor$ points. Recursively compute the convex hulls of these two subsets and then combine the hulls in $O(n)$ time. The running time is given by the recurrence

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

and so the running time is $O(n \log n)$.