Lecture 4: Divide and Conquer

subject	TDT4121
topics	
created	@September 24, 2022 11:15 AM

Chapter 4 - Divide and Conquer

> Divide and conquer refers to a class of algorithmic techniques in which one breaks the input into several parts, solves the problem in each part recursively, and then combines the solutions to these subproblems into an overall solution. In many cases, it can be a simple and powerful method.

Goals 🏁

The student should be able to:

- Understand the divide and conquer algorithm design method ✅
- Understand mergesort ✅
  - Understand how its ideas are used to solve the Counting Inversions problem
- Understand the Finding the Closest Pair of Points problem ✅
- Understand the Integer Multiplication problem
- Solve a recurrence by “unrolling” it
- Solve a recurrence by using *substitution* ✅

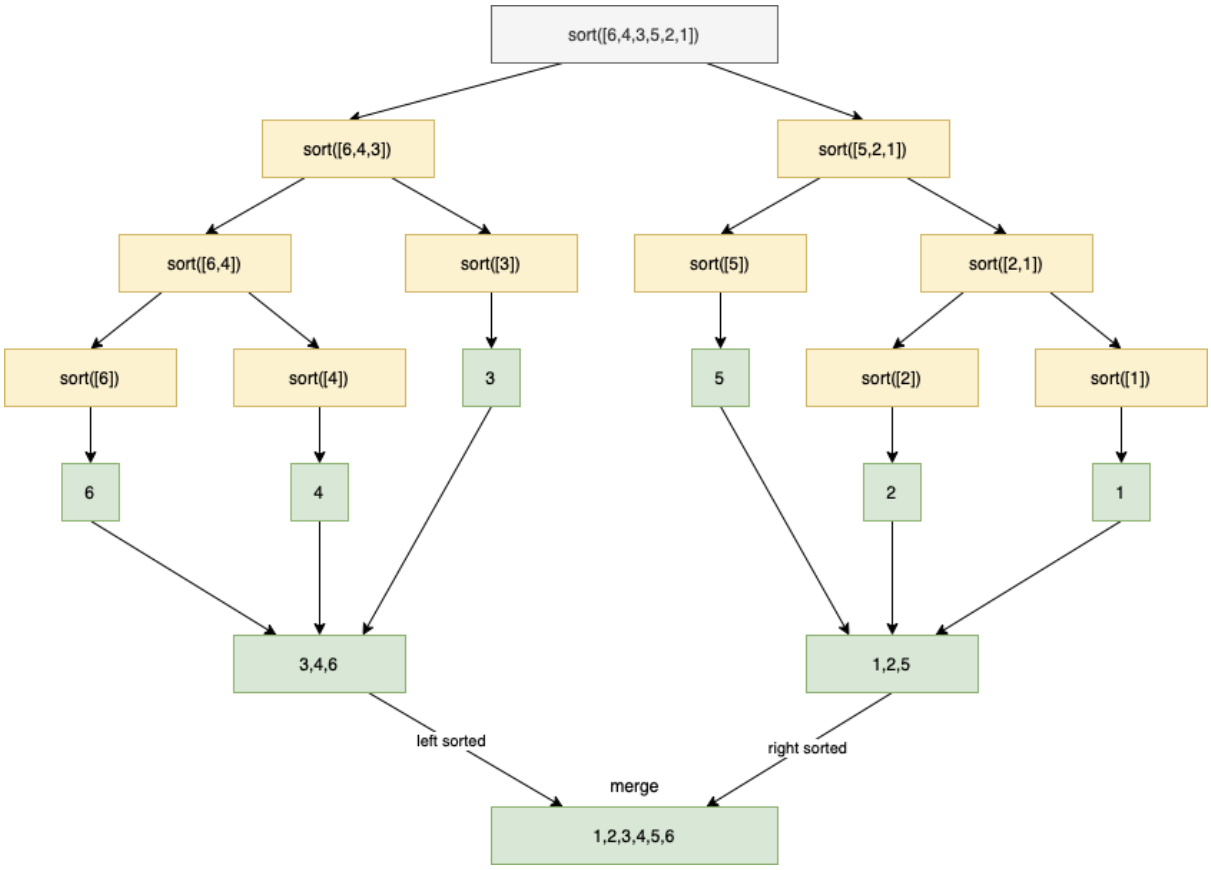
Divide and Conquer Algorithm Design Method

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Merge-Sort

Merge Sort is a divide and conquer algorithm. It works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. So Merge Sort first divides the array into equal halves and then combines them in a sorted manner.

Example of merge sort of array [6,4,3,5,2,1]



As one can see, the algorithm divides the array in half until it is left with a single element. It then compares each element to each other to decide whether the element is larger or smaller than the other, and sorts it accordingly. After the left and right side is sorted, they are then merged into a sorted array.

Runtime

With merge sort algorithms, we have the number of comparisons given by

$$C(n) = n * log(n)$$

Counting Inversions

Finding the Closest Pair of Points problem

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. We have the following formula for distance between two points p and q

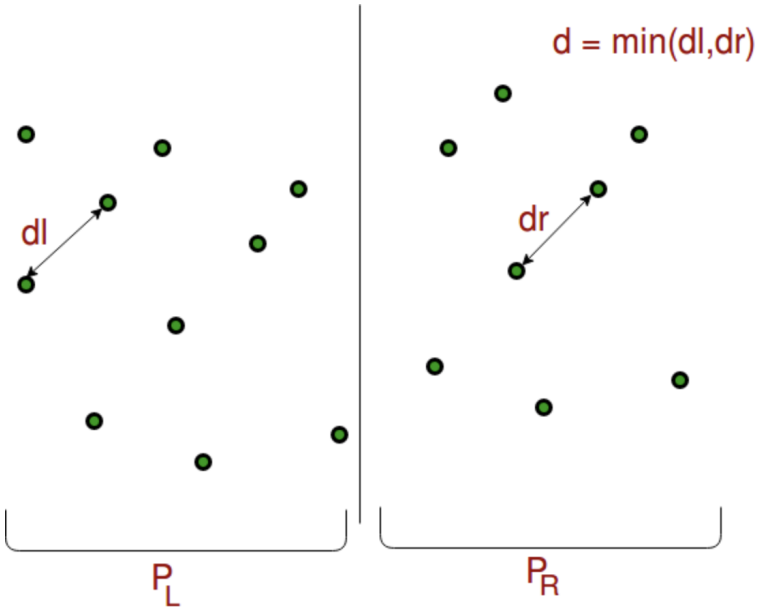
$$\|pq\| = \sqrt{(p_x.q_x)^2 + (p_y - q_y)^2}$$

Runtime

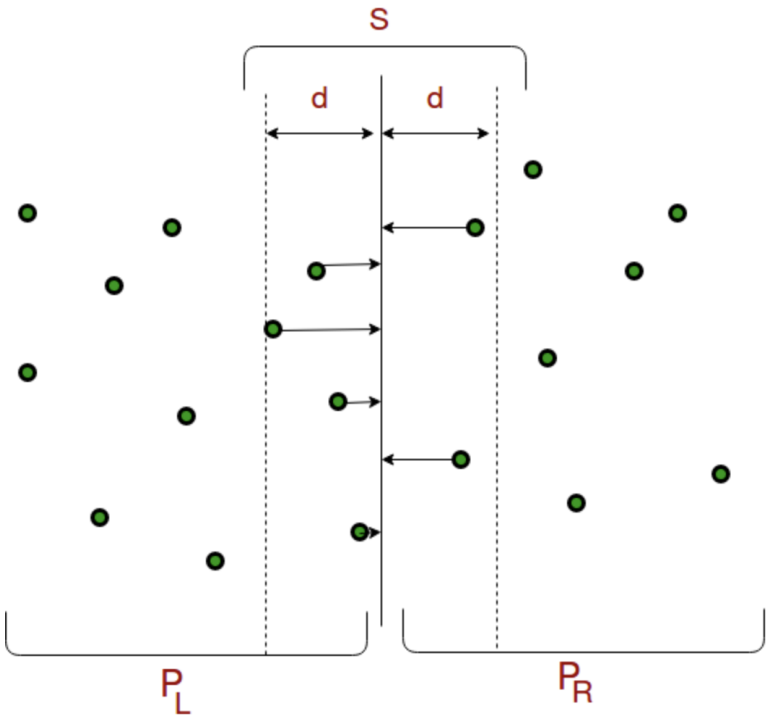
The brute force solution if $O(n^2)$, and computes the distance between each pair and returns the smallest.

We can calculate the smallest distance in $O(nlogn)$ time using divide and conquer strategy. We have an input array of n points $P[]$, and an output which is the smallest distance between two points in the given array. As a pre-processing step, the input array is sorted according to x coordinates. We do the following steps

1. Find the middle point in the sorted array, we can take $P[n/2]$ as middle point
2. Divide the given array in two halves. The first subarray contains points from $P[0]$ to $P[n/2]$. The second subarray contains points from $P[n/2+1]$ to $P[n-1]$.
3. Recursively find the smallest distances in both subarray. We let the distances be dl and dr. We find the minimum of dl and dr, and let the minimum be d.



4. From the 3 previous steps, we have an upper bound d of minimum distance. Now we need to consider pairs such that one point in pair if from the left half and the other is from the right half. We consider the vertical line passing through $P[n/2]$ and find all points whose x coordinate is closed than d to the middle vertical line. We build an array $strip[]$ of all such points
5. Sort the array strip[] according to y coordinates. This step is $O(nLogn)$. It can be optimised to $O(n)$ by recursively sorting and merging.



6. Find the smallest distance in $strip[]$. This is tricky. From the first look, it seems to be a $O(n^2)$step, but it is actually $O(n)$. It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to y coordinate). See **this** for more analysis
7. Finally return the minimum of d and distance calculated in the above step (step 6)

Integer Multiplication Problem

Large Integer Multiplication is a common procedure in computer-assisted problem solving. Multiplying big numbers is not only difficult, but also time-consuming and error-prone.

Large Integer Multiplication using Grade School Multiplication

In school, we studied the traditional multiplication technique. The multiplicand is multiplied by each digit of the multiplier in that technique, and a partial result of each multiplication is added by performing appropriate shifting. This method is also known as the **Traditional Multiplication Method**.

The following example demonstrates how to perform grade school multiplication

$$\begin{array}{r} 3\,8\,4 \\ \times \quad 5\,6 \\ \hline 2\,3\,0\,4 \\ 1\,9\,2\,0\;\; \\ \hline 2\,1\,5\,0\,4 \end{array}$$

This approach is simple to grasp, yet it is time-consuming and inefficient. If multiplicand has n digits and multiplier has m digits, the complexity of multiplication would be $O(mn)$.

Large Integer Multiplication using Divide and Conquer Approach

Recurrences

A recurrence is **an equation or inequality that reflects the value of a function with smaller inputs**. A recurrence can be used to represent the running duration of an algorithm that comprises a recursive call to itself. Time complexities are readily approximated by recurrence relations in many algorithms, specifically divide and conquer algorithms.

Solving a recurrence by “unrolling” it

Solving a recurrence by using *substitution*

The substitution approach entails predicting the answer’s structure, then using mathematical induction to identify the constants and demonstrate that the answer is correct. When the inductive hypothesis is applied to lower numbers, the estimated answer is substituted for the function, thus the name. This approach is effective, but it can only be used in situations when guessing the form of the response is simple. The substitution technique may be used to provide upper and lower boundaries on recurrences.