Lecture Notes
Data Structures and Algorithms

The Divide and Conquer Method for Algorithm Design

Overview

Divide and conquer is an algorithmic paradigm that involves dividing a problem into smaller subproblems, solving the subproblems, and then combining the solutions to the subproblems to solve the original problem.

Intuition: The divide and conquer method might be used to help plan a group project. Students might divide the project into smaller subproblems, assign each subproblem to a team member, and then combine the solutions to the subproblems to complete the project.

Divide and conquer algorithms have the following steps:

1. Divide: Divide the problem into smaller subproblems.
2. Conquer: Solve the smaller subproblems recursively.
3. Combine: Combine the solutions to the subproblems to solve the original problem.

The key idea behind divide and conquer algorithms is that they allow us to solve problems more efficiently by breaking them down into smaller pieces. This can lead to significant speed improvements, especially for problems with a large input size.

Some examples of divide and conquer algorithms include:

- Quicksort: A sorting algorithm that divides the input list into smaller sublists, sorts the sublists, and then combines the sorted sublists to form the final sorted list.
- Mergesort: A sorting algorithm that divides the input list into smaller sublists, sorts the sublists, and then merges the sorted sublists to form the final sorted list.
- Binary search: A search algorithm that divides the input list into smaller sublists and searches for the target element in the appropriate sublist.

Subproblems and independence

To use a Divide and Conquer approach, we must be certain that subproblems are independent. To determine if subproblems are independent or not, consider the following:

- Is it possible to solve the subproblems concurrently or in any order? If the subproblems can be solved concurrently or in any order, then they are likely independent.

- Does the solution to one subproblem affect the solution to the other subproblems? If the solution to one subproblem does not affect the solution to the other subproblems, then the subproblems are likely independent.
- Can the subproblems be combined in a meaningful way to solve the original problem? If the subproblems can be combined in a meaningful way, then they are likely independent.

For example, consider the problem of finding the maximum sum of a subarray. In this problem, the subproblems are not independent, because the solution to one subproblem (the maximum sum of a subarray ending at a particular index) depends on the solution to the previous subproblem (the maximum sum of a subarray ending at the previous index).

On the other hand, consider the problem of finding the maximum element in an array. In this problem, the subproblems are independent, because the solution to one subproblem (the maximum element in a subarray) does not depend on the solution to any other subproblem.

Examples of other problems that **cannot** be solved using a divide and conquer approach include the shortest path problem, the longest common subsequence, and the matrix chain multiplication problem. For each of these problems, the subproblems are not independent or the results cannot be combined in a meaningful way, which prevents a divide and conquer approach from working correctly.

Complexity

Divide and conquer algorithms are typically more efficient than brute force algorithms that solve problems by enumerating all possible solutions. However, they do have a higher overhead due to the need to divide and combine the subproblems. Divide and conquer algorithms typically have a time complexity of O(n log n), which is typically more efficient than brute force algorithms. However, divide and conquer algorithms may not be as efficient as dynamic programming algorithms, which can have a time complexity of O(n) or better for some problems.

Common Mistakes

Not properly implementing the base case:

The base case is an important part of any divide and conquer algorithm, as it determines when the recursion will stop. If the base case is not properly implemented, the algorithm may not terminate and could enter an infinite loop. For instance, when the input size is odd.

```java
// Incorrect implementation of base case
public int divideAndConquer(int[] arr) {
    if (arr.length == 1) {
      return arr[0];
    }
    int mid = arr.length / 2;
    int left = divideAndConquer(Arrays.copyOfRange(arr, 0, mid));
    int right = divideAndConquer(Arrays.copyOfRange(arr, mid, arr.length));
    return left + right;
  }
```

In this example, the base case is not properly implemented, as it only handles the case where there is a single element in the array. This means that the recursion will not stop when the array has more than one element, resulting in an infinite loop.

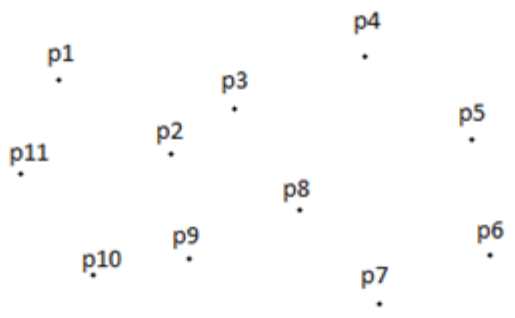
Not properly dividing the subproblems:

```
// Incorrect combination of subproblem results
public int divideAndConquer(int[] arr) {
    if (arr.length == 1) {
      return arr[0];
    }
    if (arr.length == 2) { // Solving the previous problem
        return Math.max(arr[0], arr[1]);
    }
    int mid = arr.length / 2;
    int left = divideAndConquer(Arrays.copyOfRange(arr, 0, mid - 1)); // !!
    int right = divideAndConquer(Arrays.copyOfRange(arr, mid + 1, arr.length)); // !!
    return left + right;
  }
```

In this example, the problem is not properly divided into subproblems, as the subarrays passed to the recursive calls do not contain exactly half of the elements in the original array. This means that the subproblems are not independent and the algorithm will not work correctly.

Example 1: The Closest Pair Problem

In this problem, you are given a set of points in a two-dimensional plane and your goal is to find the pair of points that are closest to each other.



Here is a possible divide and conquer algorithm for solving this problem:

Algorithm: Closest Pair

1. Divide: Divide the set of points into two halves.

2. Conquer: Find the pair of points with the smallest distance in each half using a brute force algorithm.

3. Combine: Consider the pairs of points that cross the dividing line between the two halves. Find the smallest distance among these pairs and return the pair with the smallest distance overall.

This algorithm has a time complexity of O(n log n), which is more efficient than the brute force algorithm with a time complexity of O(n^2).

```java
import java.util.Arrays;
import java.util.Comparator;

public class ClosestPairs {

    // Point class represents a point in 2D space
    public static class Point {
        public double x, y;

        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
    }

    // comparator to sort points by x-coordinate
    public static class XComparator implements Comparator<Point> {
        @Override
        public int compare(Point a, Point b) {
            return Double.compare(a.x, b.x);
        }
    }

    // comparator to sort points by y-coordinate
    public static class YComparator implements Comparator<Point> {
        @Override
        public int compare(Point a, Point b) {
            return Double.compare(a.y, b.y);
        }
    }

    // compute the distance between two points
    public static double distance(Point a, Point b) {
        double dx = a.x - b.x;
        double dy = a.y - b.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    // find the closest pair of points in the given array
    public static double closestPair(Point[] points) {
        // sort the points by x-coordinate (for divide and conquer)
        Arrays.sort(points, new XComparator());

        // recursively find the closest pair of points
        return closestPair(points, 0, points.length - 1);
    }

    // find the closest pair of points in the given subarray
    public static double closestPair(Point[] points, int left, int right) {
        // base case: 2 or 3 points
        if (right - left <= 3) {
            double minDistance = Double.MAX_VALUE;
            for (int i = left; i <= right; i++) {
                for (int j = i + 1; j <= right; j++) {
                    double d = distance(points[i], points[j]);
                    if (d < minDistance) {
                        minDistance = d;
                    }
                }
            }
            return minDistance;
        }

        // divide and conquer
        int mid = (left + right) / 2;
        double minDistance = Math.min(closestPair(points, left, mid),
                closestPair(points, mid + 1, right));

        // merge step: consider points within minDistance of the midpoint
        Point[] strip = new Point[right - left + 1];
        int k = 0;
        for (int i = left; i <= right; i++) {
            if (Math.abs(points[i].x - points[mid].x) < minDistance) {
                strip[k++] = points[i];
            }
        }

        // sort the points by y-coordinate
        Arrays.sort(strip, 0, k, new YComparator());

        // check for pairs of points in the strip that are closer than minDistance
        for (int i = 0; i < k; i++) {
            for (int j = i + 1; j < k && strip[j].y - strip[i].y < minDistance; j++)
{
                double d = distance(strip[i], strip[j]);
                if (d < minDistance) {
                    minDistance = d;
                }
            }
        }

        return minDistance;
    }

    public static void main(String[] args) {
        Point[] points = {
                new Point(2, 3),
                new Point(12, 30),
                new Point(40, 50),
                new Point(5, 1),
                new Point(12, 10),
                new Point(3, 4)
        };
        double minDistance = closestPair(points);
        System.out.println("The minimum distance is: " + minDistance);
    }
}
```

- The P**oint** class represents a point in 2D space with x and y coordinates. It is used to store the points that are being considered by the algorithm.
- The **XComparator** and **YComparator** classes are comparators that are used to sort the points by their x-coordinate and y-coordinate, respectively. These comparators are used in the Arrays.sort() method to sort the points in the necessary order for the divide and conquer method.

- The **distance()** method computes the Euclidean distance between two points. It is used to calculate the distance between pairs of points being considered by the algorithm.
- The **closestPair()** method is the main method for the algorithm. It takes an array of points and sorts it by x-coordinate using the **XComparator** class. It then calls the **closestPair()** method with the left and right indices of the subarray, which is the recursive method that performs the divide and conquer.
- The **closestPair()** method with left and right indices as parameters is the recursive method that performs the divide and conquer. It has a base case for when there are 2 or 3 points in the subarray, in which case it checks the distance between all pairs of points and returns the minimum. Otherwise, it divides the subarray into two halves and recursively calls itself on each half to find the minimum distance in each half. It then compares the minimum distance from each half and also considers any points that are within that distance of the midpoint, which may belong to the other half. These points are sorted by y-coordinate using the YComparator class and checked for any pairs that are closer than the current minimum distance. The final minimum distance is returned.

Complexity

The closest pairs algorithm is a divide and conquer method for finding the minimum distance between any two points in a set of points. It works by sorting the points by x-coordinate and recursively dividing the set into two halves. It then compares the minimum distance between the points in the two halves, and also considers any points that are within that distance of the midpoint, which may belong to the other half. These points are sorted by y-coordinate and checked for any pairs that are closer than the current minimum distance.

The base case of the recursion occurs when there are 2 or 3 points in the subarray, in which case the distance between all pairs of points is checked and the minimum is returned.

The line with the highest time complexity in the algorithm is the nested loop that checks all pairs of points in the base case:

```
for (int i = 0; i < k; i++) {
    for (int j = i + 1; j < k && strip[j].y - strip[i].y <
minDistance; j++) {
        double d = distance(strip[i], strip[j]);
        if (d < minDistance) {
            minDistance = d;
        }
    }
}
```

This loop has a time complexity of O(n^2) in the worst case, when there are 2 or 3 points in the subarray.

However, this only occurs at the bottom of the recursion, and the overall time complexity of the algorithm is O(n log n) due to the divide and conquer method.

Proof

A proof that the closest pairs algorithm using the divide and conquer method leads to the optimal solution:

We will prove the claim by contradiction. Suppose that the algorithm does not find the optimal solution, and let **d** be the minimum distance between any two points in the set of points being considered by the algorithm, and let d' be the distance returned by the algorithm. We will show that **d'** must be less than or equal to **d**, which contradicts the assumption that **d**' is not the optimal solution.

The algorithm works by dividing the set of points into two halves and recursively considering each half. At each step, it compares the minimum distance between the two halves and also considers any points that are within that distance of the midpoint, which may belong to the other half.

Let **p** and **q** be two points in the set of points such that the distance between them is **d**. Without loss of generality, assume that **p** is in the left half and q is in the right half.

Since **p** is in the left half, the algorithm will consider it when it recursively considers the left half. Similarly, since **q** is in the right half, the algorithm will consider it when it recursively considers the right half.

At the root of the recursion, the algorithm compares the minimum distance between the left and right halves and considers any points that are within that distance of the midpoint. Since **p** and **q** are both within d of the midpoint (since d is the minimum distance between any two points in the set), they will be considered at the root of the recursion.

Therefore, **p** and **q** will be considered at some point in the algorithm, and their distance **d** will be compared to the current minimum distance **d'**. Since d is the minimum distance between any two points in the set, it must be less than or equal to d', which means that d' is less than or equal to **d**.

This contradicts the assumption that **d'** is not the optimal solution, so it must be the case that the algorithm finds the optimal solution.

Example 2: A problem that cannot be solved using Divide and Conquer

One example of a problem for which it may seem that divide and conquer is a valid approach, but the subproblems are not independent and the algorithm will not work correctly, is the following:

Given an array of integers, find the maximum sum of a subarray.

At first glance, it may seem that this problem can be solved using a divide and conquer approach by dividing the array into two subarrays and recursively finding the maximum sum of the subarrays. However, this approach will not work, because the maximum sum of a subarray may span multiple subarrays.

For example, consider the following array:

```
[1, -3, 2, -5, 7, 6, -1, -4, 11, -23]
```

If we divide this array into two subarrays, the maximum sum of a subarray in each subarray will not be the maximum sum of the entire array. For example, the maximum sum of the left subarray is 7, and the maximum sum of the right subarray is 11. However, the maximum sum of the entire array is 20, which spans both subarrays.

Therefore, this problem cannot be solved using a divide and conquer approach, as the subproblems are not independent and the results cannot be combined in a meaningful way.

To solve this problem, we need to use dynamic programming.