Algorithms Analysis and Design

Chapter 4

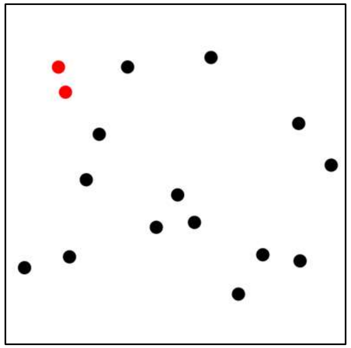
Divide and Conquer
Part 4

Divide-and-Conquer Examples

- **Sorting**:
  - merge sort
  - quicksort
- **Binary tree traversals**
- **Mathematics**
  - Multiplication of large integers
  - Matrix multiplication: Strassen's algorithm
  - Exponentiation problem
- **Computational geometry**
  - Closest-pair
  - convex-hull algorithms
- **Searching**:
  - Binary search: decrease-by-half (or degenerate divide&conq.)

Closest-pair problem

- **Recall**: In chapter 3, we discussed the brute-force approach to solve the closest-pair problem.
- Closest-pair problem: Find the two closest points in a set of n points (in the two-dimensional Cartesian plane for example).



- **Brute-force algorithm**
  - Compute the distance between every pair of distinct points.
  - and return the indexes of the points for which the distance is the smallest.

This problem can be solved by brute force algoristms in $\theta(n^2)$

Closest-Pair Problem by Divide-and-Conquer

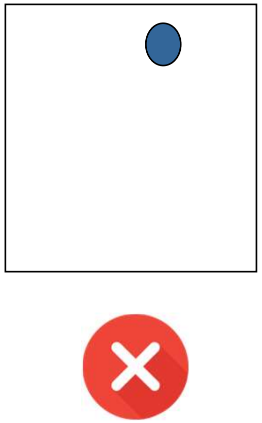
Assumptions:

- For the sake of simplicity, we assume that the points are distinct.
- We can also assume that the points are **ordered in nondecreasing order of their x coordinate**. (If they were not, we could sort them first by an efficient sorting algorithm such as mergesort.)
- It will also be convenient to have the points sorted in a separate list in nondecreasing order of the y coordinate; we will denote such a list Q.

Closest-Pair Problem by Divide-and-Conquer

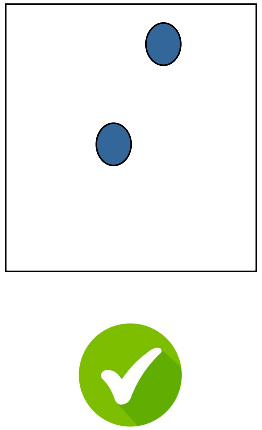
Base cases:

n=1?



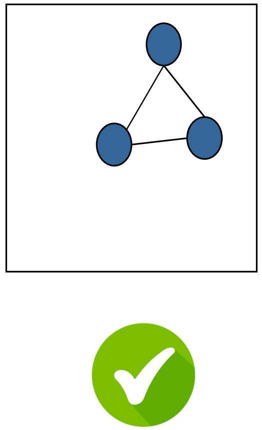
We should at least have a pair

n=2?



If we have two points in the set, we can just say this is the closest pair

n=3?

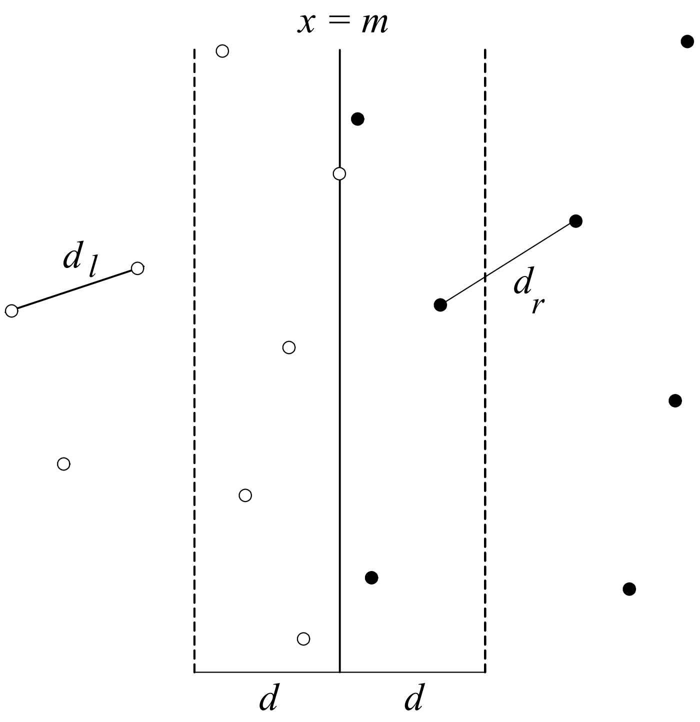


If we are dividing a set of three points, one half must be left with one point which is not a valid base case.

In this case we just look at the three possible pairs and choose the closes one (solve it using brute force)

Closest-Pair Problem by Divide-and-Conquer

Step 1 Divide the points given into two subsets P_l and $\mathrm{P_r}$ by a vertical line $x = m$ so that half the points lie to the left or on the line and half the points lie to the right or on the line.



Distance between two points:

$p_1 = (x_1, y_1)$

$p_2 = (x_2, y_2)$

$$p_1p_2 = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2)}$$

Closest Pair by Divide-and-Conquer (cont.)

Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 Set $d = \min\{d_l, d_r\}$

We can limit our attention to the points in the symmetric vertical strip S of width $2d$ as possible closest pair. (The points are stored and processed in increasing order of their y coordinates.)

Step 4 Scan the points in the vertical strip S from the lowest up. For every point $p(x,y)$ in the strip, inspect points in in the strip that may be closer to p than d. There can be no more than 5 such points following p on the strip list!

Unfortunately, d is not necessarily the smallest distance between all pairs of points in left subset (S1) and right subset (S2) because a closer pair of points can lie on the opposite sides separating the line. When we combine the two sets, **we must examine such points.**

Pseudo code

```
closestPair(X,Y)
    n = X.length;
    // base cases:
    if (n==2): return dist(X[1], X[2]);
    if (n==3): return min(dist(X[1], X[2]), dist(X[2], X[3]), dist(X[1], X[3]));

    // divide
    mid = X[n/2];
    dl = closestPair(X[1...mid], Y);
    dr = closestPair(X[mid+1...n], Y);
    d = min(dl, dr);

    // combine
    S = points in Y whose x-coordinates are in the range of [mid.x-d, mid.x+d];
    for i=1 to S.length
        for j=1 to j=7
            d = min(d, dist(S[i], S[i+j]));
    return d;
```

Efficiency of the Closest-Pair Algorithm

Running time of the algorithm is described by

$$T(n) = \mathbf{2}T(n/\mathbf{2}) + M(n), \quad \text{where } M(n) \in O(n)$$

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in O(n \log n)$$

Recall that we assumed that the points are **ordered in nondecreasing order of their x coordinate**. (If they were not, we could sort them first by an efficient sorting algorithm such as mergesort.)

Exercise: Derive the time complexity of the closes pair algorithm assuming that the set of points are not ordered.

Algorithms Analysis and Design

Chapter 5

Decrease and Conquer

Decrease-and-Conquer

1. Reduce problem instance to smaller instance of the same problem
2. Solve smaller instance
3. Extend solution of smaller instance to obtain solution to original instance

- Can be implemented either top-down or bottom-up
- Also referred to as *inductive* or *incremental* approach

3 Types of Decrease and Conquer

- ***Decrease by a constant*** **(usually by 1):**
  - insertion sort
  - topological sorting
  - algorithms for generating permutations, subsets

- ***Decrease by a constant factor*** **(usually by half)**
  - binary search
  - exponentiation by squaring

- *Variable-size decrease*
  - Euclid's algorithm
  - selection by partition
  - Nim-like games

What’s the difference?

Consider the problem of exponentiation: Compute a^n

- Brute Force: ?
- Divide and conquer: ?
- Decrease by one: ?
- Decrease by constant factor: ?

We have discussed this problem previously in this chapter

Insertion Sort

To sort array A[0..*n*-1], sort A[0..*n*-2] recursively and then insert A[*n*-1] in its proper place among the sorted A[0..*n*-2]

- Usually implemented bottom up (nonrecursively)

Example: Sort 6, 4, 1, 8, 5

```
6 | 4   1   8   5
4   6 | 1   8   5
1   4   6 | 8   5
1   4   6   8 | 5
1   4   5   6   8
```

Pseudocode of Insertion Sort

ALGORITHM *InsertionSort*($A[0..n-1]$)

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Analysis of Insertion Sort

- Time efficiency

  $C_{worst}(n) = n(n-1)/2 \in \Theta(n^2)$

  $C_{avg}(n) \approx n^2/4 \in \Theta(n^2)$

  $C_{best}(n) = n - 1 \in \Theta(n)$ (also fast on almost sorted arrays)

- Space efficiency: in-place

- Stability: yes

- Best elementary sorting algorithm overall

- Binary insertion sort

Exercise

- Let $A[1..n]$ be an array of n sortable elements. (For simplicity, you may assume that all the elements are distinct.) A pair $(A[i], A[j])$ is called an inversion if $i < j$ and $A[i] > A[j]$.

1. What arrays of size n have the largest number of inversions and what is this number? Answer the same questions for the smallest number of inversions.
2. Design an efficient Decrease-and-Conquer algorithm for counting the number of inversions.
3. Express the time complexity of the algorithm in (2) using big-O notation.