Greedy, Divide and Conquer

League of Programmers

ACA, IIT Kanpur

October 5, 2013

Outline

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem
- Greedy Algorithms are algorithms that

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem
- Greedy Algorithms are algorithms that
  - try to maximize the function by making greedy choice at all points. In other words, we do at each step what seems best without planning ahead

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem
- Greedy Algorithms are algorithms that
  - try to maximize the function by making greedy choice at all points. In other words, we do at each step what seems best without planning ahead
  - ignores future possibilities (no planning ahead)

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem
- Greedy Algorithms are algorithms that
  - try to maximize the function by making greedy choice at all points. In other words, we do at each step what seems best without planning ahead
  - ignores future possibilities (no planning ahead)
  - never changes choice (no backtracking)

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem
- Greedy Algorithms are algorithms that
  - try to maximize the function by making greedy choice at all points. In other words, we do at each step what seems best without planning ahead
  - ignores future possibilities (no planning ahead)
  - never changes choice (no backtracking)
- Usually O(n) or O(n lg n) Solutions.

Greedy Algorithms

- Greedy algorithms are generally used in optimization problems
- There is an optimal substructure to the problem
- Greedy Algorithms are algorithms that
  - try to maximize the function by making greedy choice at all points. In other words, we do at each step what seems best without planning ahead
  - ignores future possibilities (no planning ahead)
  - never changes choice (no backtracking)
- Usually O(n) or O(n lg n) Solutions.
- But, We must be able to prove the correctness of a greedy algorithm if we are to be sure that it works

Problem

Problem 1:

A family goes for picnic along a straight road. There are k hotels in their way situated at distances $d_1, d_2 \ldots d_k$ from the start. The last hotel is their destination. The can travel at most L length in day. However, they need to stay at a hotel at night. Find a strategy for the family so that they can reach hotel k in minimum number of days.

Problem

Solution:

The first solution that comes to mind is to travel on a particular day as long as possible, i.e. do not stop at a hotel iff the next hotel can be reached on the same day.

Problem

Proof:

Proof strategy 1 for greedy algorithms:

- Greedy solution stays ahead
- Suppose there is a better solution which stops at hotels $i_1 < i_2 \ldots i_r$ and greedy solution stops at $j_1 < j_2 \ldots j_s$ and $r < s$.
- Consider the first place where the two solutions differ and let that index be p. Clearly $i_p < j_p$
- On a given day when non greedy solution starts from hotel $i_q < j_q$, non greedy solution cannot end up ahead of greedy solution.
- Therefore Greedy solution stays ahead

Problem

Problem 2:

There are n processes that need to be executed. Process i takes a preprocessing time p_i and time t_i to execute afterwards. All preprocessing has to be done on a supercomputer and execution can be done on normal computers. We have only one supercomputer and n normal computers. Find a sequence in which to execute the processes such that the time taken to finish all the processes is minimum.

Problem

Solution:

Sort in descending order of t_i and execute them. At least this is the solution for which no counterexample is available.

Problem

Proof:

- Let there be any other ordering $\sigma(1)$,$\sigma(2)\ldots\sigma(n)$ such that $t\sigma(i) < t\sigma(i+1)$ for some i.
- Prove that by executing $\sigma(i+1)$ before $\sigma(i)$, we will do no worse.
- Keep applying this transformation until there are no inversions w.r.t. to t. This is our greedy solution.

Problem

Problem 3:

A thief breaks into a shop only to find that the stitches of his bag have become loose. There are N items in the shop that he can choose with weight W[i] and value V[i]. The total weight that the bag can carry is W. How should he choose the items to maximize the total value of the things he steals?

Outline

Divide and Conquer

Divide and Conquer

- Suppose P(n) is the problem we are trying to solve, of size n

Divide and Conquer

- Suppose P(n) is the problem we are trying to solve, of size n
- We can solve P(n) directly, for sufficiently small n

Divide and Conquer

- Suppose P(n) is the problem we are trying to solve, of size n
- We can solve P(n) directly, for sufficiently small n
- Divides the problem P(n) into subproblems

$$P_1(n_1);\ P_2(n_2);\ :\ :\ :\ ;\ P_k(n_k)$$

for some constant k

Divide and Conquer

- Suppose P(n) is the problem we are trying to solve, of size n
- We can solve P(n) directly, for sufficiently small n
- Divides the problem P(n) into subproblems

$$P_1(n_1);\ P_2(n_2);\ :\ :\ :\ ;\ P_k(n_k)$$

  for some constant k
- Combine the solutions for $P_i(n_i)$ $(1 \leq i \leq k)$ to solve P(n)

Problem

Problem 1:

- **Merge-sort is divide-n-conquer.**
- Analysis of merge-sort.

$$T(n) \leq 2T(n/2) + n$$

- Problem- Find number of inversions by modifying code for merge-sort slightly.

ClosestPair Problem

Closest Pair of points

Given N points in a plane, find the pair of points that are closest to each other.

ClosestPair Problem

ClosestPair Problem

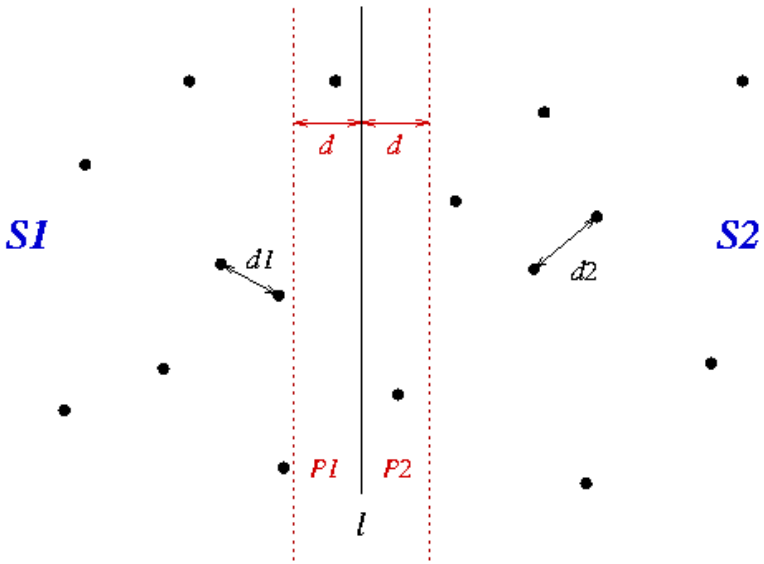

Figure: 1

ClosestPair Problem

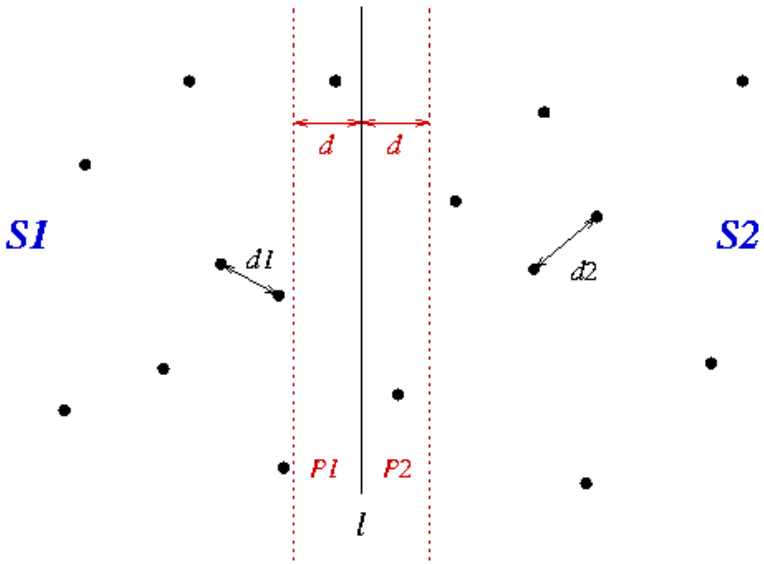

Figure: 1

ClosestPair Problem

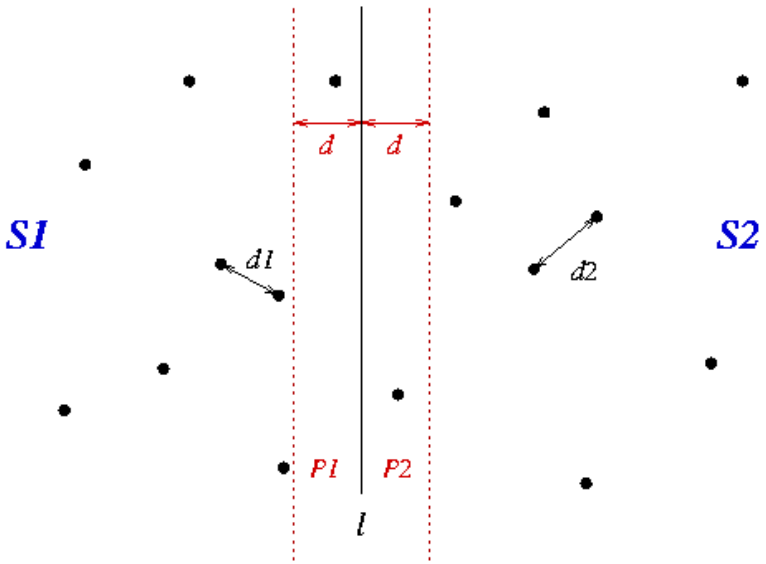

Figure: 1

- First sort the points in increasing order of X coordinate.

ClosestPair Problem

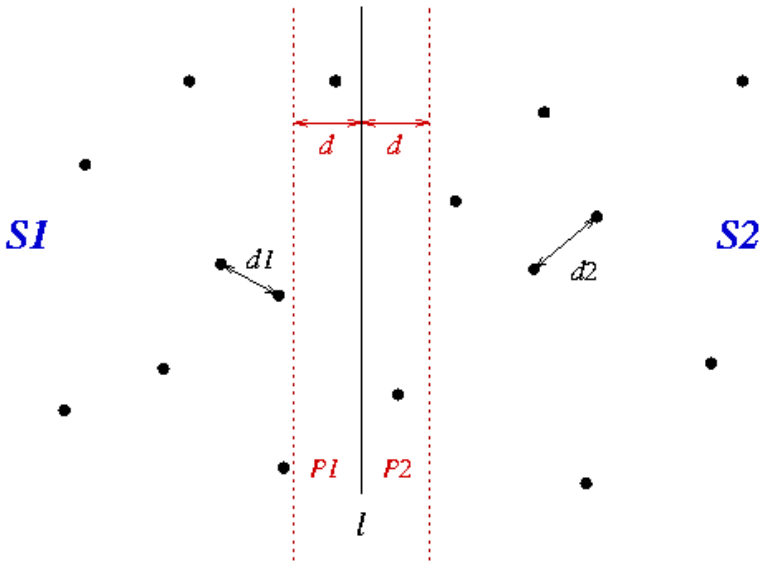

Figure: 1

- First sort the points in increasing order of X coordinate.
- Divide the plane into two halves with equal number of points P1 and P2.

ClosestPair Problem

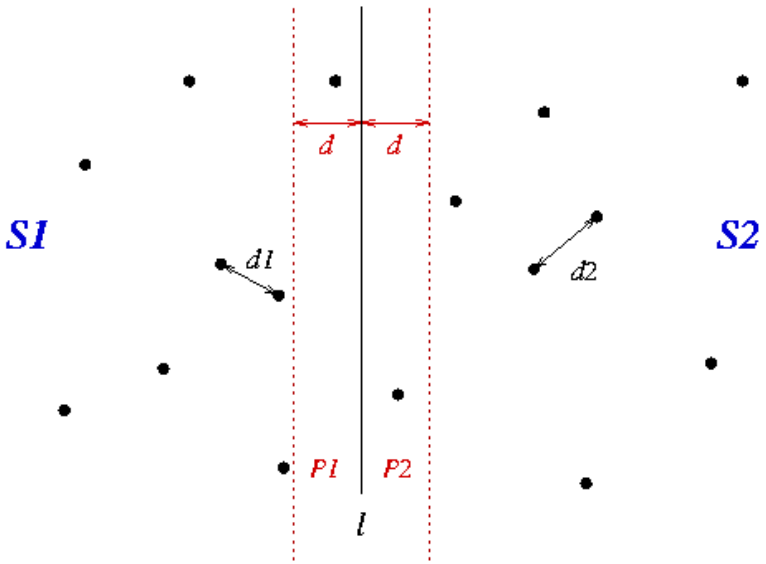

Figure: 1

- First sort the points in increasing order of X coordinate.
- Divide the plane into two halves with equal number of points P1 and P2.
- Let d1 be the minimum distance between closest pair in the P1 and similarly d2 in P2.

ClosestPair Problem

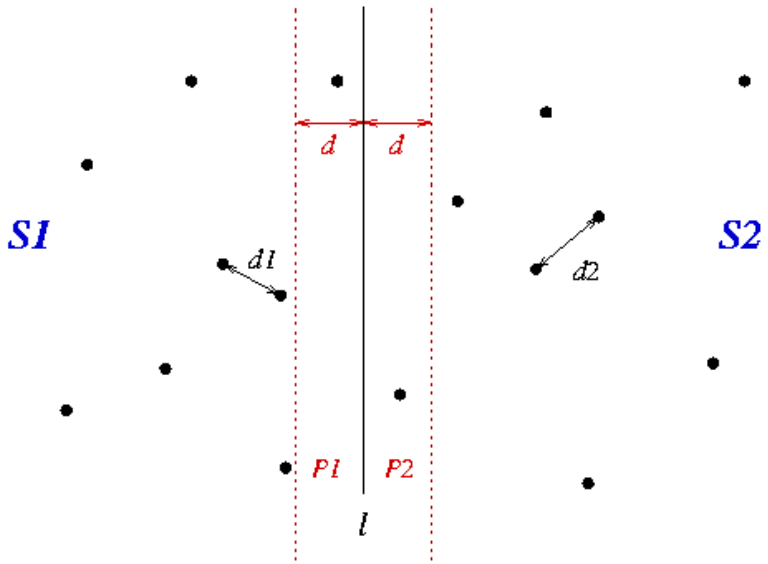

Figure: 1

- First sort the points in increasing order of X coordinate.
- Divide the plane into two halves with equal number of points P1 and P2.
- Let d1 be the minimum distance between closest pair in the P1 and similarly d2 in P2.
- d1: minimum distance of P1

ClosestPair Problem

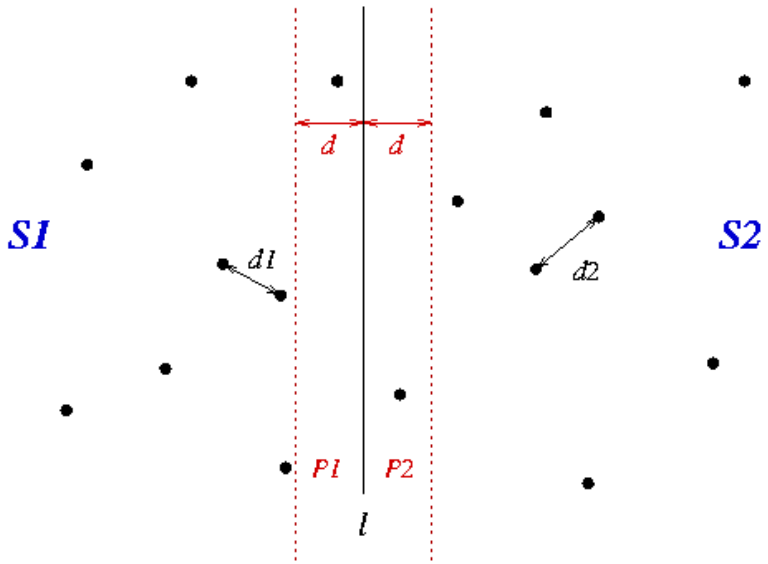

Figure: 1

- First sort the points in increasing order of X coordinate.
- Divide the plane into two halves with equal number of points P1 and P2.
- Let d1 be the minimum distance between closest pair in the P1 and similarly d2 in P2.
- d1: minimum distance of P1
- d2: minimum distance of P2

ClosestPair Problem

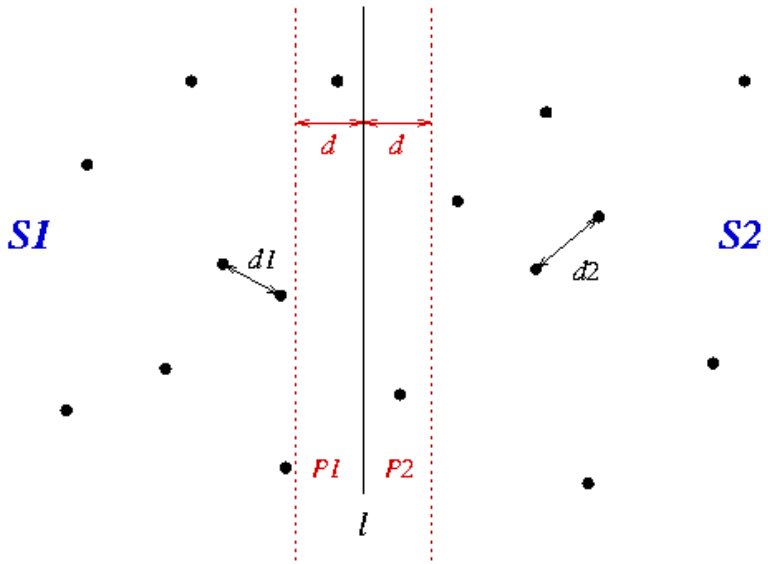

Figure: 1

- First sort the points in increasing order of X coordinate.
- Divide the plane into two halves with equal number of points P1 and P2.
- Let d1 be the minimum distance between closest pair in the P1 and similarly d2 in P2.
- d1: minimum distance of P1
- d2: minimum distance of P2
- $d < \min\{d1, d2\}$.

ClosestPair Problem

- Now, region of interest is the band of 2d around the dividing line.

ClosestPair Problem

- Now, region of interest is the band of 2d around the dividing line.
- But, In fact, all of the points could be in the strip! This is disastrous, because we would have to compare n^2 pairs of points to merge the set, and hence our divideandconquer algorithm wouldn't save us anything in terms of efficiency.

ClosestPair Problem

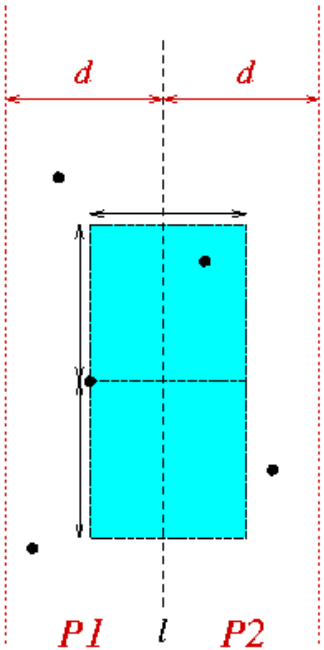

Figure 3.3

ClosestPair Problem

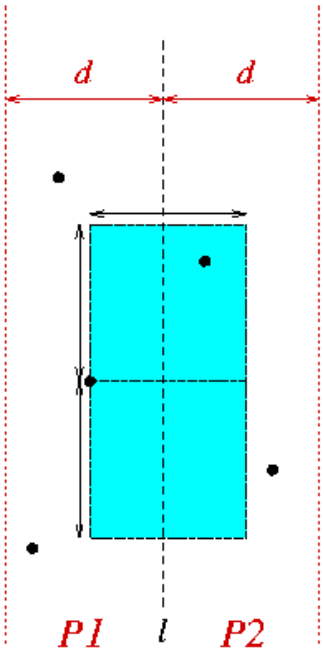

Figure 3.3

Thankfully, we can make another life saving observation at this point. For any particular point p in one strip, only points that meet the following constraints in the other strip need to be checked:

1. Those points within d of p in the direction of the other strip
2. Those within d of p in the positive and negative y directions

because points outside of this bounding box cannot be less than d units from p.

ClosestPair Problem

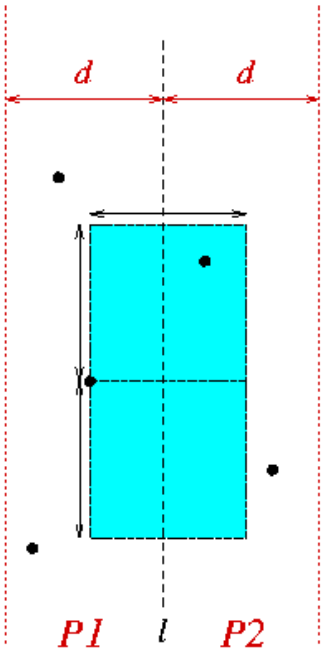

Figure 3.3

- It just so happens that because every point in this box is at least d apart, there can be at most six points within it.

ClosestPair Problem

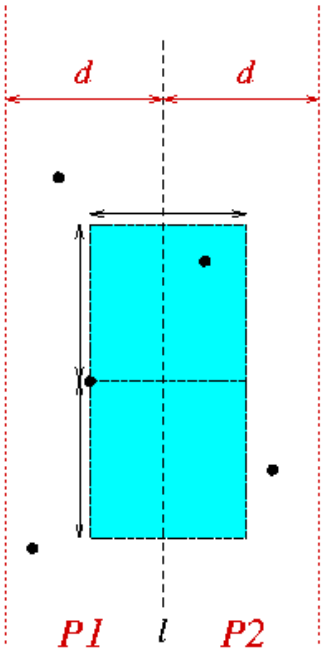

Figure 3.3

- It just so happens that because every point in this box is at least d apart, there can be at most six points within it.
- Now we don't need to check all n^2 points. All we have to do is sort the points in the strip by their y-coordinates and scan the points in order, checking each point against a maximum of 6 of its neighbors.

ClosestPair Problem

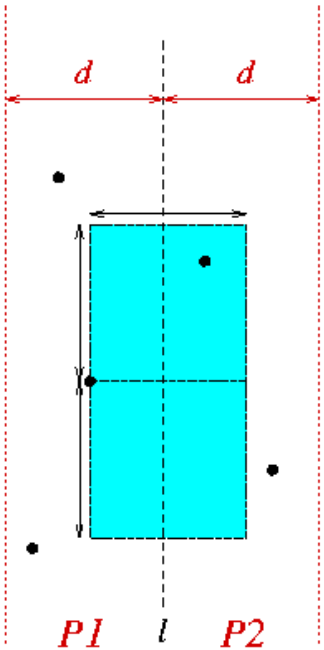

Figure 3.3

- It just so happens that because every point in this box is at least d apart, there can be at most six points within it.
- Now we don't need to check all n^2 points. All we have to do is sort the points in the strip by their y-coordinates and scan the points in order, checking each point against a maximum of 6 of its neighbors.
- This means at most $6 * n$ comparisons are required to check all candidate pairs.

ClosestPair Problem

Summary

ClosestPair Problem

Summary

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.

ClosestPair Problem

Summary

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.

ClosestPair Problem

Summary

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.
3. Eliminate points that lie farther than d apart from l

ClosestPair Problem

Summary

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.
3. Eliminate points that lie farther than d apart from l
4. Sort the remaining points according to their y-coordinates

ClosestPair Problem

Summary

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.
3. Eliminate points that lie farther than d apart from l
4. Sort the remaining points according to their y-coordinates
5. Scan the remaining points in the y order and compute the distances of each point to its five neighbors.

ClosestPair Problem

Summary

1. Divide the set into two equal sized parts by the line l, and recursively compute the minimal distance in each part.
2. Let d be the minimal of the two minimal distances.
3. Eliminate points that lie farther than d apart from l
4. Sort the remaining points according to their y-coordinates
5. Scan the remaining points in the y order and compute the distances of each point to its five neighbors.
6. If any of these distances is less than d then update d.

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time
- Step 3 takes O(n) time

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time
- Step 3 takes O(n) time
- Step 4 is a sort that takes O(nlogn) time

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time
- Step 3 takes O(n) time
- Step 4 is a sort that takes O(nlogn) time
- Step 5 takes O(n) time (as we saw in the previous section)

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time
- Step 3 takes O(n) time
- Step 4 is a sort that takes O(nlogn) time
- Step 5 takes O(n) time (as we saw in the previous section)
- Step 6 takes O(1) time

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time
- Step 3 takes O(n) time
- Step 4 is a sort that takes O(nlogn) time
- Step 5 takes O(n) time (as we saw in the previous section)
- Step 6 takes O(1) time

Hence the merging of the sub-solutions is dominated by the sorting at step 4, and hence takes O(nlogn) time.

ClosestPair Problem

Analysis

Steps 2-6 define the merging process which must be repeated log(n) times because this is a divide and conquer algortithm:

- Step 2 takes O(1) time
- Step 3 takes O(n) time
- Step 4 is a sort that takes O(nlogn) time
- Step 5 takes O(n) time (as we saw in the previous section)
- Step 6 takes O(1) time

Hence the merging of the sub-solutions is dominated by the sorting at step 4, and hence takes O(nlogn) time.
This must be repeated once for each level of recursion in the divide-and-conquer algorithm, hence the whole of algorithm ClosestPair takes O(logn*nlogn) = $O(n log^2 n)$ time.

ClosestPair Problem

Improving the Algorithm

We can improve on this algorithm slightly by reducing the time it takes to achieve the y-coordinate sorting in Step 4. This is done by asking that the recursive solution computed in Step 1 returns the points in sorted order by their y coordinates. This will yield two sorted lists of points which need only be merged in Step 4 in order to yield a complete sorted list.

ClosestPair Problem

Improving the Algorithm

Hence the revised algorithm involves making the following changes:

ClosestPair Problem

Improving the Algorithm

Hence the revised algorithm involves making the following changes:

- Step 1: Divide the set into. . . , and recursively compute the distance in each part, returning the points in each set in sorted order by y-coordinate.

ClosestPair Problem

Improving the Algorithm

Hence the revised algorithm involves making the following changes:

- Step 1: Divide the set into. . . , and recursively compute the distance in each part, returning the points in each set in sorted order by y-coordinate.
- Step 4: Merge the two sorted lists into one sorted list in O(n) time. Hence the merging process is now dominated by the linear time steps thereby yielding an O(nlogn) algorithm for finding the closest pair of a set of points in the plane.

Divide and Conquer

When to use divide and conquer

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems
- The subproblems may be solved independently

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems
- The subproblems may be solved independently
- The size of each subproblem is much smaller than the original

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems
- The subproblems may be solved independently
- The size of each subproblem is much smaller than the original

Divide and Conquer is a bad choice when:

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems
- The subproblems may be solved independently
- The size of each subproblem is much smaller than the original

Divide and Conquer is a bad choice when:

- There are too many subproblems

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems
- The subproblems may be solved independently
- The size of each subproblem is much smaller than the original

Divide and Conquer is a bad choice when:

- There are too many subproblems
- The subproblems are not independent

Divide and Conquer

When to use divide and conquer

Divide and conquer works well when:

- Divide step produces a constant number of subproblems
- The subproblems may be solved independently
- The size of each subproblem is much smaller than the original

Divide and Conquer is a bad choice when:

- There are too many subproblems
- The subproblems are not independent
- The subproblems are too large

Outline

Binary Search

Problem: Finding a value in a sorted sequence

For example, find 55 in the sequence

0, 5, 13, 15, 24, 43, 55, 65, 72, 85, 96

Binary Search

Problem: Finding a value in a sorted sequence

For example, find 55 in the sequence

0, 5, 13, 15, 24, 43, 55, 65, 72, 85, 96

What is the time complexity?

Binary Search

Problem: Finding a value in a sorted sequence

For example, find 55 in the sequence

0, 5, 13, 15, 24, 43, 55, 65, 72, 85, 96

What is the time complexity?

At each step, we are discarding half of the array.

Binary Search

Code

```
binary_search(A, target):
        lo = 1, hi = size(A)
        while lo <= hi:
                mid = lo + (hi-lo)/2
                if A[mid] == target:
                        return mid
                else if A[mid] < target:
                        lo = mid+1
                else:
                  hi = mid-1

        // target was not found
```

Problem

In a building with 100 (=n) storeys a person is required to find starting from which floor an egg if thrown to the ground will be broken. He has been given just 2(=m) eggs for this task. All he can do is to go to certain storeys and throw his egg and record whether the egg broke or not. Assume that for each storey, an egg will either always break or will never break. Since, climbing stairs (there's no lift!), throwing eggs and going downstairs again to check whether the egg broke or not and to take the egg again to the next floor being checked can be a tiring process, devise a strategy following which the person will find the required storey having after minimum number of throws

Generalization

Extending the idea of Problem 1

The same algorithm described above on any monotonic function f whose domain is the set of integers. The only difference is that we replace an array lookup with a function evaluation: we are now looking for some x such that f(x) is equal to the target value

Problem

Problem:

CodeJam

Outline

Problems

Links:

1. http://spoj.com/problems/BALIFE
2. http://www.spoj.com/problems/KOPC12A
3. http://spoj.com/problems/AGGRCOW
4. http://spoj.com/problems/ARRANGE
5. http://www.spoj.com/problems/BAISED
6. http://www.spoj.com/problems/DONALDO
7. http://www.spoj.com/problems/PIE
8. http://www.spoj.com/problems/CISTFILL

Acknowledgements

Image Credits:
http://www.cs.mcgill.ca/ cs251/ClosestPair/ClosestPairDQ.html