# Closest Pair of Points

Another surprising and satisfying application of the D&C method is the following algorithm for finding the closest pair of points in a set of points in the x-y plane.

Assume we have n points, each identified by its x and y coordinates. Our goal is to find the two points with the minimum distance. Obviously we can solve this problem in O (n^2) time by computing the distance between all pairs of points. It may seem unlikely that we can reduce this, but by clever design we can eliminate enough of the pairwise distance calculations that we can achieve a lower complexity.

For reasons that will become clear later, we start by creating a list of all the points sorted by their x-coordinate (X-List) and another list of all the points sorted by their y-coordinate (Y-List). This is a pre-processing phase that sets up the rest of the algorithm.

In well established D&C fashion, we split the problem into two subproblems. In this case we split the points into a left half and a right half, based on the x-coordinate. We then recursively solve the problem on the left side and the right side. As always with a D&C algorithm, when the problem size is <= some specific number we solve the problem directly. In this case we might decide that when the number of points is <= 3, we will compute all the distances. This takes O(1) time, which means we can treat it as a constant.

However, after we split the point set into a left side and a right side and solve the problem recursively on both sides, we are not done. We still have to deal with the possibility that the two points that are closest together are on opposite sides of the dividing line. If we compute all the distances between points on the left and points on the right, we are back with O(n^2) complexity.

Fortunately we are able to avoid most of the potential computations. Let d1 be the minimum distance on the left side, and d2 be the minimum distance on the right side - we get these values from the recursive applications of the algorithm

to the left and right halves of the set. Let delta = min(d1,d2). We need to determine if there are two points, one on each side of the dividing line, that have distance less than delta from each other. We can eliminate all points that have distance more than delta from the dividing line, since they cannot be less than delta from any point on the other side of the line.

Imagine a vertical panel or strip, 2*delta wide, centred on the dividing line between the left and right sides. The only points we need to consider are within this panel. Take these points in ascending order by y-coordinate (oh, so this is why we sorted the points before we started!). For each point, compute its distance to the points in the panel above it that might possibly be less than delta away. By a simple geometric argument there cannot be more than 7 (see my note [1] below) such points. Thus for each point in the vertical panel, we need to compute no more than 7 distances to other points in the panel. Even if all n points are in the panel, the complexity of computing the necessary distances within the panel is O(n).

Thus the complexity of the algorithm is given by the recurrence relation

$$T(n) = 2*T(n/2) + c*n \quad \text{when } n > 3$$
$$T(n) = \text{constant} \quad \text{when } n <= 3$$

We know this recurrence relation - it is the same one that describes the complexity of merge-sort. We know that it works out to O(n*log n). This is the same as the complexity of the pre-processing step, which means that the pre-processing step is effectively free.

It is worth noting that if we had to re-sort the points at the beginning of each recursive call, the complexity would be higher. Fortunately we don't - each reduced set of points is just a subset of the set at the previous level and the relative order of the points does not change.

---

1Note: I have followed the text-book here by saying that there are no more than 7 points to consider. In class I presented an argument that there can be no more than 5 such points. The argument for 7 such points depends on allowing identical points in the set, and having the identical pairs on "opposite sides" of the dividing line. I think this situation can be resolved as a special case. Either way, the number of point-combinations that must be considered is strictly linear.

# Greedy Algorithms

Suppose we are planning a trip from Kingston to Montreal, driving along the 401. Our car will need several stops to fill up the gas tank. We know the location of all the service stations on the 401 between Kingston and Montreal, and we know exactly how far we can travel on a tank of gas. We want to minimize the number of stops we make along the way.

The intuitive answer, of course, is to go as far as possible on each tank - that is, stop at a service station iff we can't make it to the next one. It turns out that in this case intuition can be trusted - this is exactly the right solution.

We can formalize this idea into an algorithm as follows. I'll show a recursive formulation first, then two iterative versions.

```
1.  Sort the stations according to how far they are from Kingston
    Let S = {s1,s2, ... sn} be the sorted set, with sn = Montreal

2.  Road-Trip(n)        # n is the number of the next station coming
                        # up.
                        # We know we have enough gas to reach this
                        # station
       if n == Montreal
           Exit

       if we have enough gas to reach station n+1
          # ignore station n
          Road-Trip(n+1)
       else
          fill up at station n
          Road-Trip(n+1)
```

We find the solution by calling Road-Trip(1)

Iterative version 1 - this is basically the version we used in class:

```
1.  Sort the stations according to how far they are from Kingston
    Let S = {s1,s2, ... sn} be the sorted set, with sn = Montreal


2.  Stops={}           # we want to minimize the size of Stops
    i = 1
    while i < n
       if we have enough gas to reach station i+1
          # skip station i

       else:
          fill up with gas at station i
          Stops.append(i)
       i += 1
    Stops.append{n}      # make sure we stop in Montreal
```

Iterative version 2 - this is the same as the last version, except that we remember "Stop" or "Skip" for each station:

```
1.  Sort the stations according to how far they are from Kingston
    Let S = {s1,s2, ... sn} be the sorted set, with sn = Montreal


2.  Decisions={}          # we want to maximize the number of "Skip"
entries in Decisions
    i = 1
    while i < n
       if we have enough gas to reach station i+1
          # skip station i
          Decisions.append("Skip")
       else:
          fill up with gas at station i
          Decisions.append("Stop")
       i += 1
    Decistion.append("Stop")       # stop in Montreal
```

To complete our consideration of this problem, we need to do several things:

1. Show that the problem does have an optimal solution (some problems don't, either because there is no solution or there is no bound on the value of the potential solutions).

2. Show that the greedy algorithm finds an optimal solution

3. Determine the complexity of the algorithm.

First, once the stations are sorted we can easily determine if there are any feasible solutions: the trip is possible iff there is no gap between stations too long for us to cover on one tank of gas. Assuming then that there are feasible solutions, it is clear that there is at least one optimal solution since we can rank all the feasible solutions according to their cardinality and choose one with minimum cardinality.

It is also clear that the algorithm (in any of the three forms shown above) will find some solution to the problem: it never lets us run out of gas, and it gets us to the destination.

Let's also dispose of the complexity issue: sorting the n stations takes O(n*log n) time. Choosing the stations where we stop takes O(n) time since we look at each station exactly once, and decide in constant time whether or not to stop there. Thus the entire algorithm takes O(n*log n) time.

Now the proof of correctness. This is the interesting part. We use Proof by Induction. I'm going to base this on the the second version of the algorithm above - the notation is just easier. In this version, our goal is to minimize the size of Stops.

**Base Case:** If |S| = 1, the only possible stop is Montreal. If there is any feasible solution (ie if we can reach Montreal on a single tank), then the algorithm's action is correct (it makes no stops before Montreal). Thus for |S| = 1, the algorithm finds an optimal solution.

**Inductive Hypothesis**

We assume that the algorithm finds an optimal solution when the number of service stations still ahead of us is <= n, for some n >= 1

**Inductive Step**

Suppose there are n+1 service stations still ahead of us.

First we show that there exists at least one optimal solution that contains the algorithm's first choice. To do this, let O be some optimal solution to the problem, i.e. O = {o1, o2, ... , ok} where each element is a station where we should fill up with gas, in order (that is, o1 is our first stop, etc.)

Let the algorithm's solution be Stops = {a1, a2, .... }

Consider a1 - the algorithm's first stop. We know it is not possible to reach the station after a1 on the first tank of gas.. Therefore **o1** must be <= a1

But now consider o2, the second stop in O. It must be reachable from o1 ... which means it is also reachable from a1. Thus we can create a new feasible solution:

Let O* = {a1, o2, ...., ok}

Note that |O*| = |O|, so O* is also optimal. Thus there is an optimal solution that contains a1.

Now we know that the algorithm's first decision is 'safe' - ie there is at least one optimal solution that matches this decision.

When we make a decision about the first stop, the problem reduces in size to <= n (i.e. the

number of remaining stations between us and Montreal is reduced). By the IH, the algorithm will find an optimal solution to this reduced problem.

The last step of our proof is to show that when we combine the algorithm's first choice with its solution to the reduced problem, we get an optimal solution to the original problem.

We know that there is at least one optimal solution that matches the algorithm's first choice a1 - let X be such an optimal solution:

X = {a1, x2, x3, ..., xt}. |X| = t

Let the algorithm's solution be Stops = {a1, a2, a3, ..., as}, as before. |Stops| = s

Due to the structure of the algorithm, we know Stops is a feasible solution.

Observe that {a2 ... as} and {x2, x3, ..., xt} are both solutions to the problem of getting from station a1 to Montreal, based on the decision made to fill up at station a1. In fact, we know {a2 ... as} is an **optimal** solution to this reduced problem of getting from station a1 to Montreal.

Thus |{a2, ..., as}| is <= |{x2, x3, ..., xn}|. Since Stops and X both start with a1, it is clear that |Stops| <= |X|. Thus Stops is also an optimal solution.

Based on this example of a greedy algorithm we can state the general paradigm:

The fundamental principle of greedy algorithms is "take the best thing first".

More formally, suppose we are trying to solve an **optimization problem** that involves choosing objects from a set, subject to some feasibility constraint. We want to choose the objects that will maximize (or minimize) the total value of the solution, while satisfying the constraint. The **Greedy Paradigm** looks like this:

1. Sort the objects according to some criterion
2. repeat
   select the next item in the list if it does not violate the feasibility constraint
   until no more selections can be made

Such a simple idea clearly will not find optimal solutions for all problems (for example, it is very easy to define greedy algorithms that find solutions to NP-Complete problems ... but

there will be instances of the problem for which the algorithm's solutions won't be optimal). However, there is a huge class of problems for which a greedy algorithm **will** always find the optimal solution. This means that our big job with respect to Greedy Algorithms will not be coming up with the algorithm, but finding a proof that the algorithm gives the optimal solution for all instances of the problem.