CS161L: Implementation of Algorithms

Friday, May 16, 2014

What Makes a Problem Greedy?

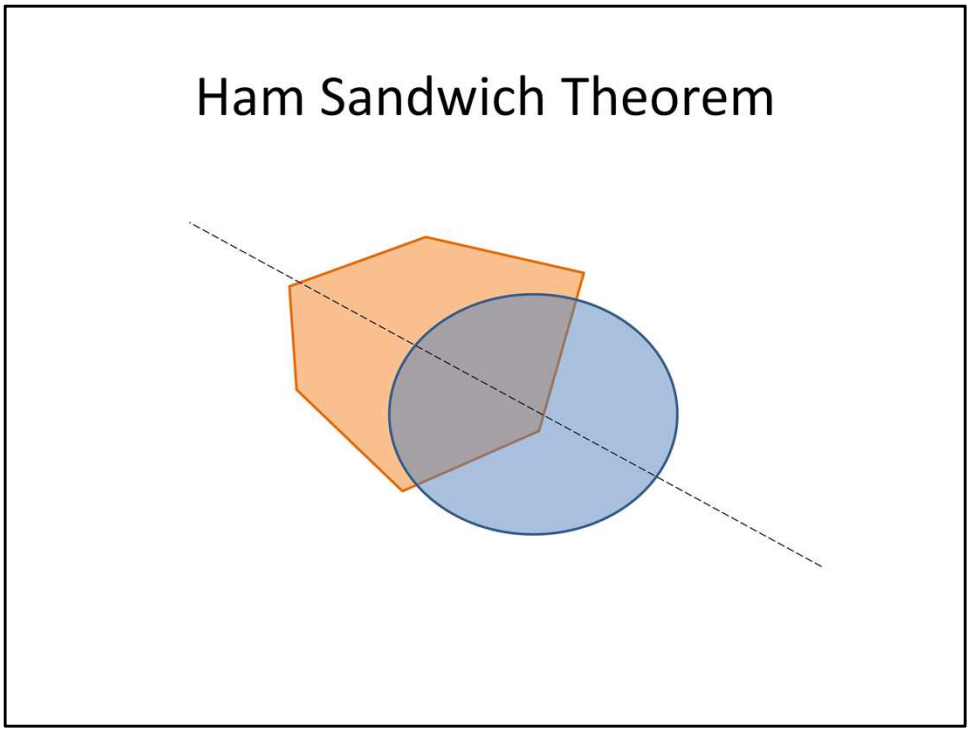
- Natural ordering of a small number of progressively larger subproblems
- Easy to solve the smallest subproblem
- Easy to solve a subproblem given the answer to the one before it

As Tim Roughgarden mentioned on Monday, greedy algorithms are easy to come up with, easy to code, and difficult to justify. Because this class is focused on coding, we're not going to devote time to coding up 1-liners. There WILL be a greedy problem in Week 9 that involves a decent amount of coding, so you'll get a taste of it then. What we're going to cover today is a problem that, strictly speaking, is not greedy, but does have a lot in common with greedy problems. So what makes a problem greedy? The important thing is that we're able to break the problem down into a bunch of smaller subproblems that have a natural ordering on them, kind of like a set of Russian dolls. If the smallest of these subproblems is easy to solve, and we can use each subproblem in turn to quickly solve the subproblem that's one step after it, then we can use a greedy approach. If this sounds a lot like dynamic programming to you, in some sense it is; you can think of greedy as being related to dynamic programming the way regular induction is related to strong induction.

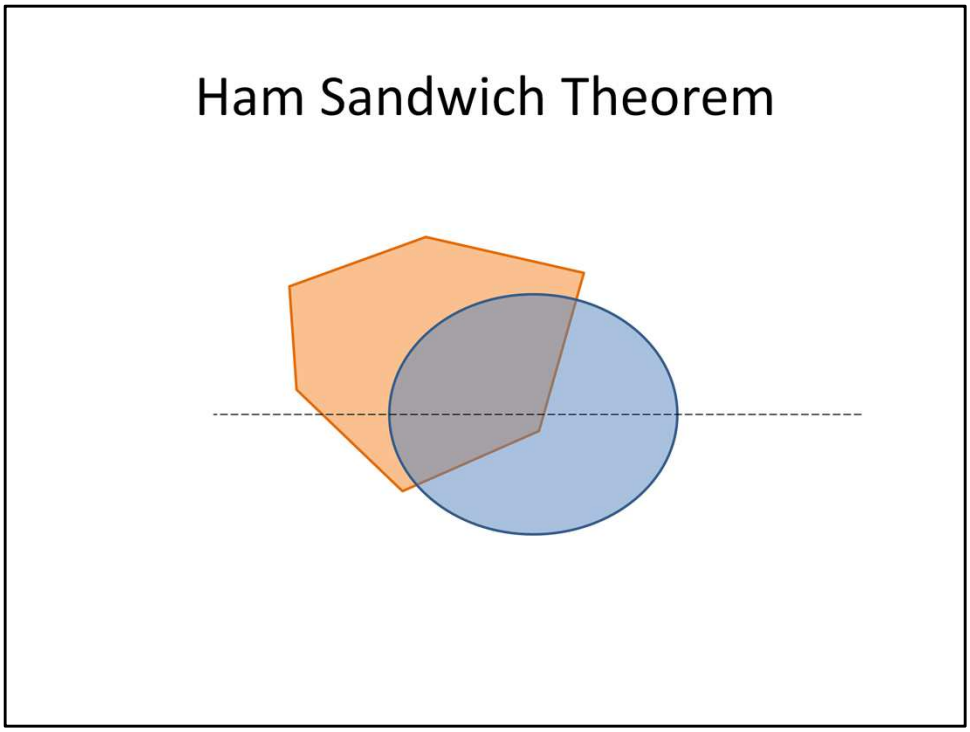
Problem for Today

- Natural ordering of a small number of potential solutions to the whole problem
- Easy to create the first potential solution
- Easy to create a potential solution from the one before it

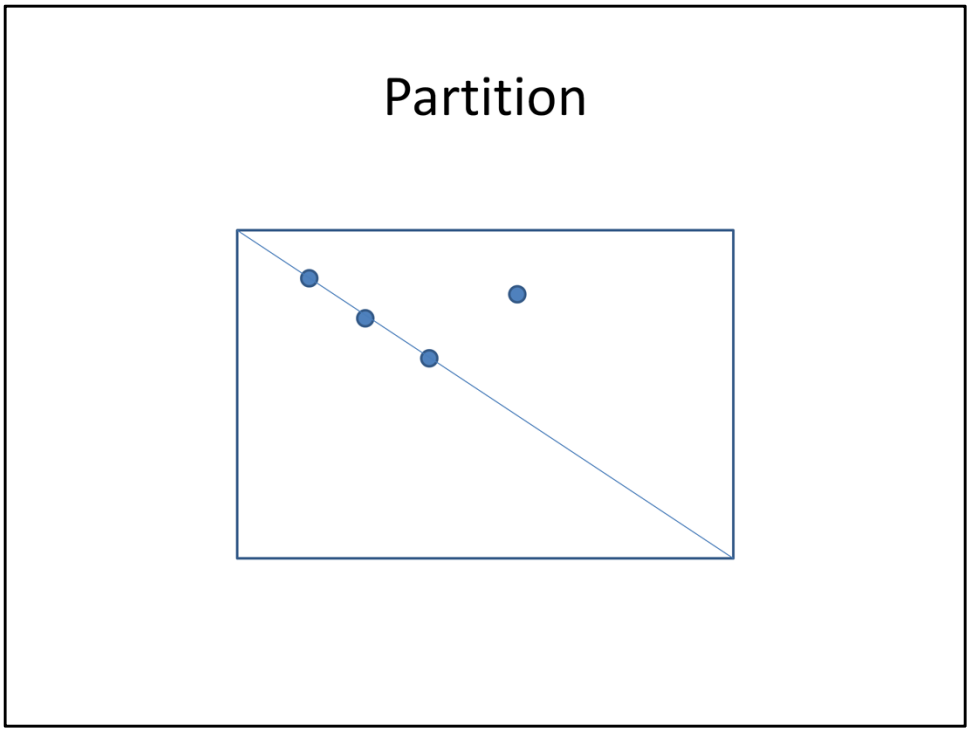
The problem for today is going to be a slight variation on that. In this problem, we're going to be able to identify a small number of potential solutions to the whole problem that we can order in some way. Then we're going to find an efficient way to start from one potential solution and iterate through the rest, until we find the one we're looking for.



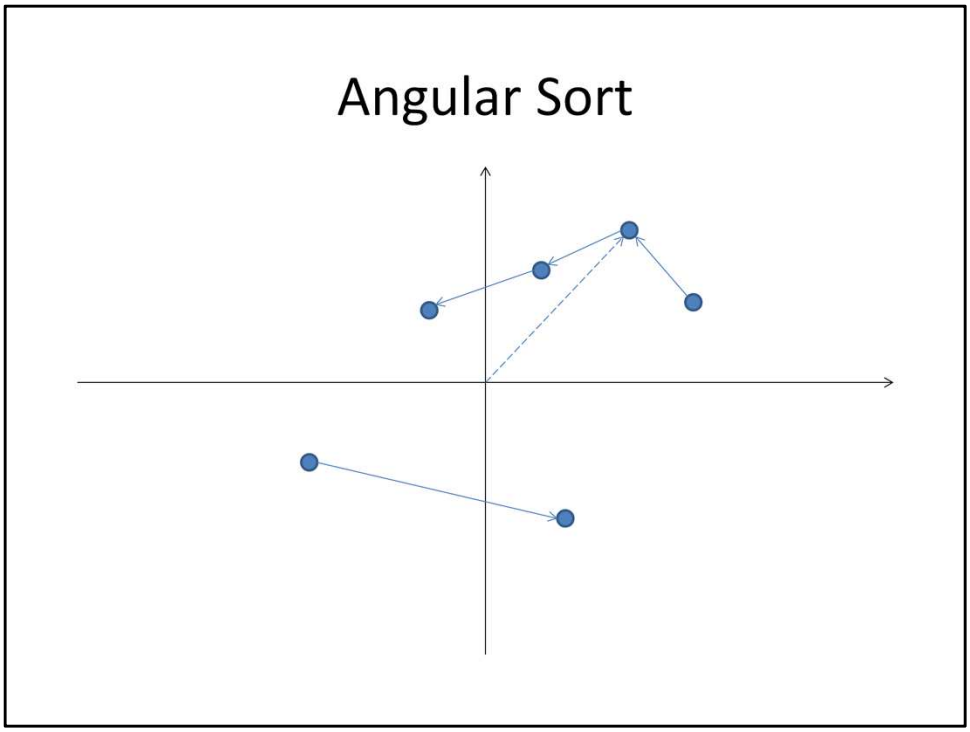
OK, let's start discussing today's problem, which is inspired by what is known as the ham sandwich theorem. This is a cute little geometric theorem which says if you have two objects in the plane, say a piece of bread and a piece of ham, it is possible to draw a single straight line that cuts both objects in half in terms of area. Why is this the case?



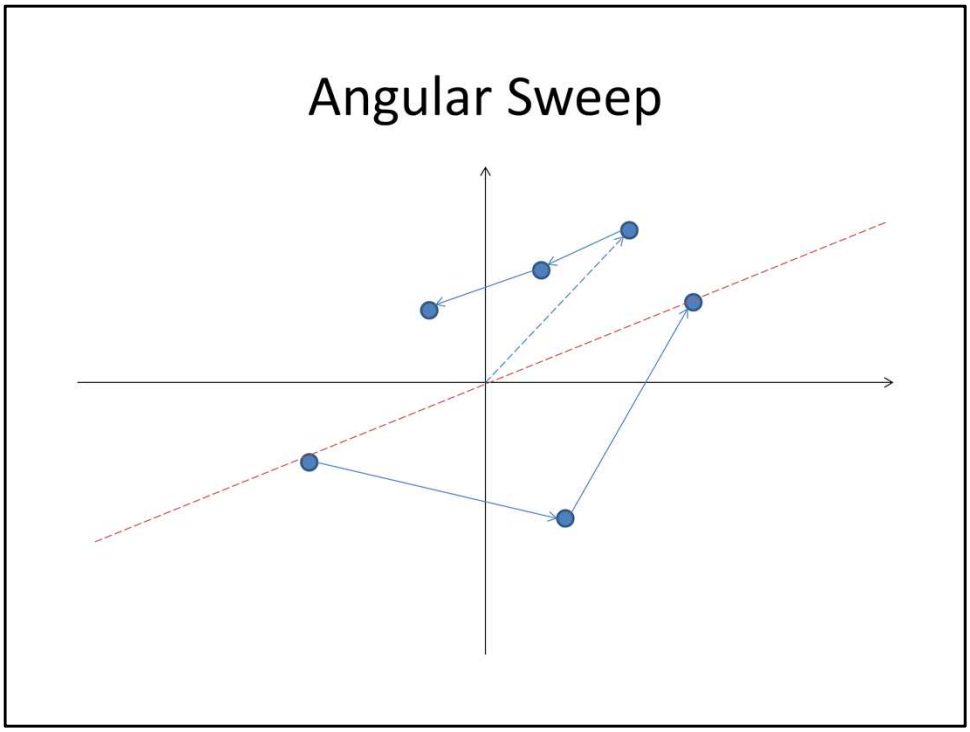
Well, let's imagine we cut one of the two objects in half along a specific direction. If we cut the other object in half, we're done; otherwise, more than half is on one side and less than half is on the other side. Now let's vary the direction continuously; we can imagine effectively rotating this line around 180 degrees. By the time we made it all the way around, less than half of the other object is now on the first side, and more than half is now on the other side. Now, the amount of the other object on one side of our line is a continuous function of the direction of the line, so we can use the same theorem we used four weeks ago in our rootfinding algorithm, the Intermediate Value Theorem. Since we started with more than half on one side and ended with less than half on that side, at some point we needed to have exactly half on that side.



Now let's move on to the problem that we want to solve. We have a rectangle, and we have an even number of points inside the rectangle. We want to partition the rectangle AND the point set in half with a single straight line. Notice that we can choose to which half to assign each point that lies exactly on the dividing line. We'll have to make use of that in cases like this, where this is the only valid dividing line, and it's valid because we can assign two of the points to the lower half. Notice that the set of lines that cut the rectangle in half are exactly the set of lines that go through the center of the rectangle, so we can set that point to be our origin and parameterize our set of solutions by the angle our line makes with the x-axis, say. This are still an infinite number of lines, so let's try to narrow down the number of potential solutions we have to examine. Notice that if the line doesn't actually intersect any of the points, the balance of points isn't changed by a perturbation of the line. The only time the balance changes is when the line passes through one of the points, so those are the only lines we have to check. So now we've managed to identify a linear number of potential solutions. If we wanted to stop here, we could just try each of the lines by counting up the number of points that end up on one side of it, being careful to put aside the points exactly on the line to assign where they're most needed. This would give us a quadratic solution. But let's see if we can find an ordering of these lines that we can use to speed up our evaluation.



Let's start with the horizontal line. It's easy to do this; we just split the points by y-value. For now, let's assign points on the negative x-axis to the top half and points on the positive x-axis to the bottom half. Now, within each half, we sort the points by the angle they make with the positive x-axis. There's a handy function that gives you this angle called atan2, which is a 2-argument arctangent function. atan2 takes in the y-value followed by the x-value, and will spit out an angle between 0 and pi for the top half and an angle between –pi and 0 for the bottom half. We can then dump each half into its own queue.



Then, advancing to the next line is the same as removing a point from one queue and adding it to the other queue. That means that it takes constant time to advance to the next line. We stop as soon as the queues are of equal size. So it takes us n log n time to set up our first line, and constant time per line to check the remaining n lines. This means that our total runtime is n log n, which is better than the n^2 bound we would have gotten by trying all the lines in an arbitrary order. Of course, you'll have to check the respective angles to make sure that doing so doesn't violate the fact that each queue is supposed to lie on one half of the line.

LinkedList

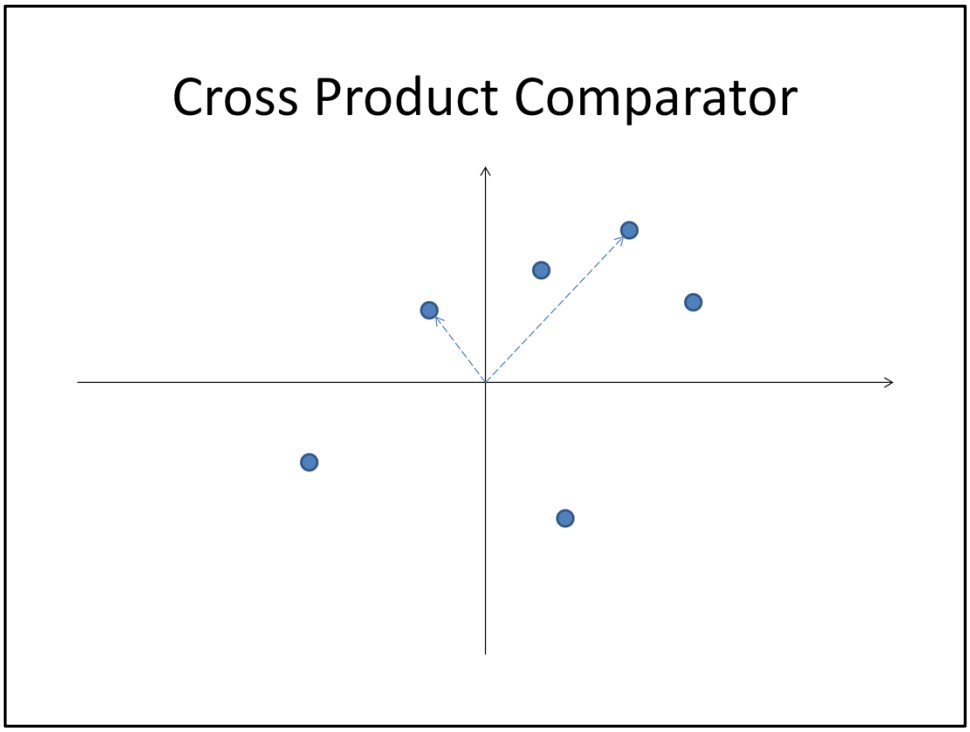
```
import java.util.*;
import java.awt.Point;
...
LinkedList<Point> queue = new LinkedList<Point>();
queue.addFirst(new Point(0, 1));
queue.addLast(new Point(2, 3));
Point p = queue.removeFirst();
Point q = queue.getLast();
int x = p.x;
int y = p.y;
```

So how do we make a queue in Java? Well, you might recall from CS106 that one implementation of a queue is a linked list. There's a LinkedList class in Java that has all of the methods we'll need for this problem. Also on this slide is a class called Point, which will be useful for storing points in the plane. You'll need one more import statement to take advantage of it.

Comparators and Sorting

```
import java.util.*;
import java.awt.Point;
...
static class AngleComparator implements Comparator<Point> {
  public int compare(Point a, Point b) {
    double theta1 = Math.atan2(a.y, a.x);
    double theta2 = Math.atan2(b.y, b.x);
    if (theta1 == theta2) return 0;
    return theta1 < theta2 ? -1 : 1;
  }
}
...
LinkedList<Point> list = new LinkedList<Point>();
...
Collections.sort(list, new AngleComparator());
```

At this point in the course, we've done enough work with sorting that we shouldn't need to do so from scratch every time. So this time around, we're going to use the Java library implementation of sort. To do this, we write a Comparator class, which needs to have a compare method that takes in two of the objects it's designed to compare, and then return a negative number, 0, or a positive number depending on whether the first object is less than, equal to, or greater than the second object, respectively. So the Comparator we see here compares Points based on the angle they make with the x-axis starting at the origin. I should warn you that in the problem specification, the rectangle is NOT centered at the origin, so this code won't work without some modification. Once you have your comparator, though, you can pass it into the Collections.sort function. Notice that we can pass in a LinkedList even though it doesn't have random access; this is fine because Java's library sort dumps out the contents to an array and does the sorting there, and then puts the result back in your data structure.



Bonus Slide: There's a trick we can use to perform the angular sort we need for the Partition problem without resorting to a trig function. Remember that we don't care about the actual angles, we just care about how they are ordered relative to each other. One way to tell whether a point comes before or after another point in this angular ordering is to take the cross product of the vectors from the origin to those two points. The sign of the cross product then tells us their ordering. Notice that this only works if all of our points lie strictly on one side of the line; otherwise, we don't obtain a valid comparator. But we split the points using x-axis and were careful with the points that were directly on the axis, so the top and bottom point sets each lie strictly on one side of some line. The reference solution uses this trick to get shorter, faster code that can use exact math for all its computations.