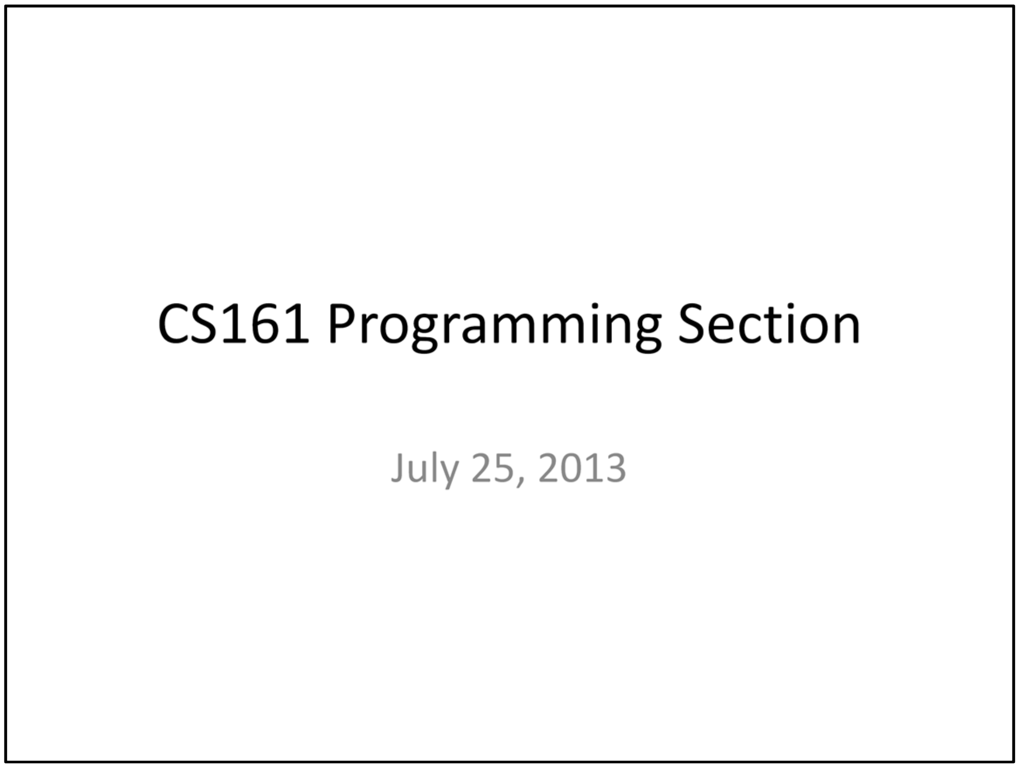
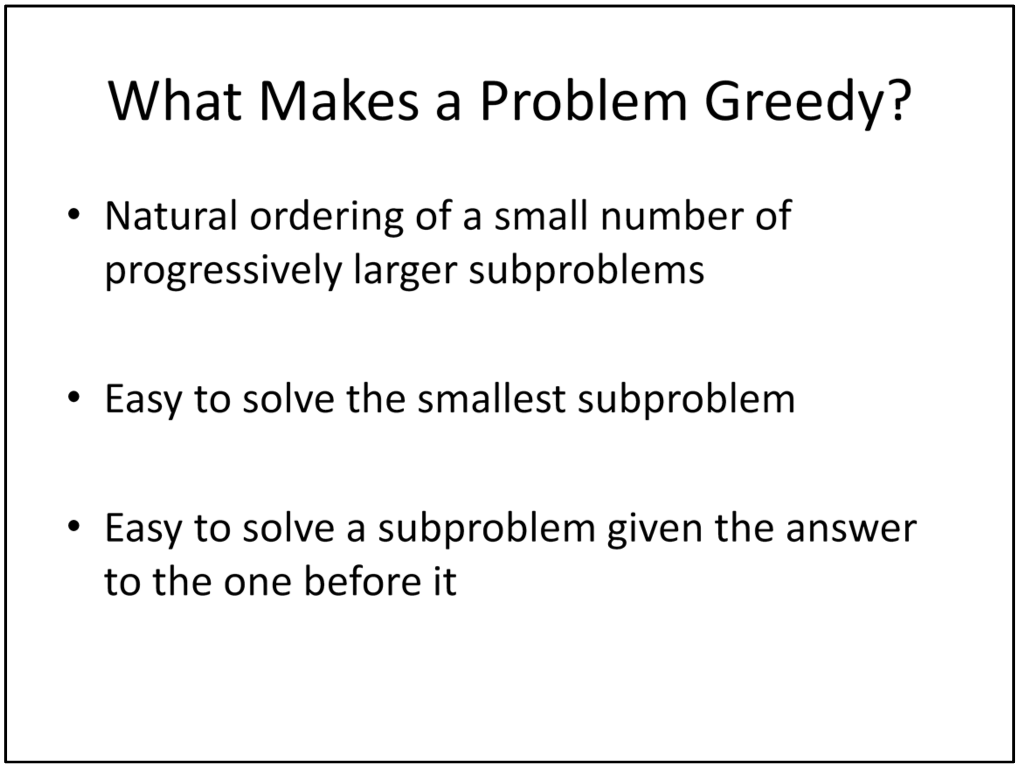


CS161 Programming Section

July 25, 2013

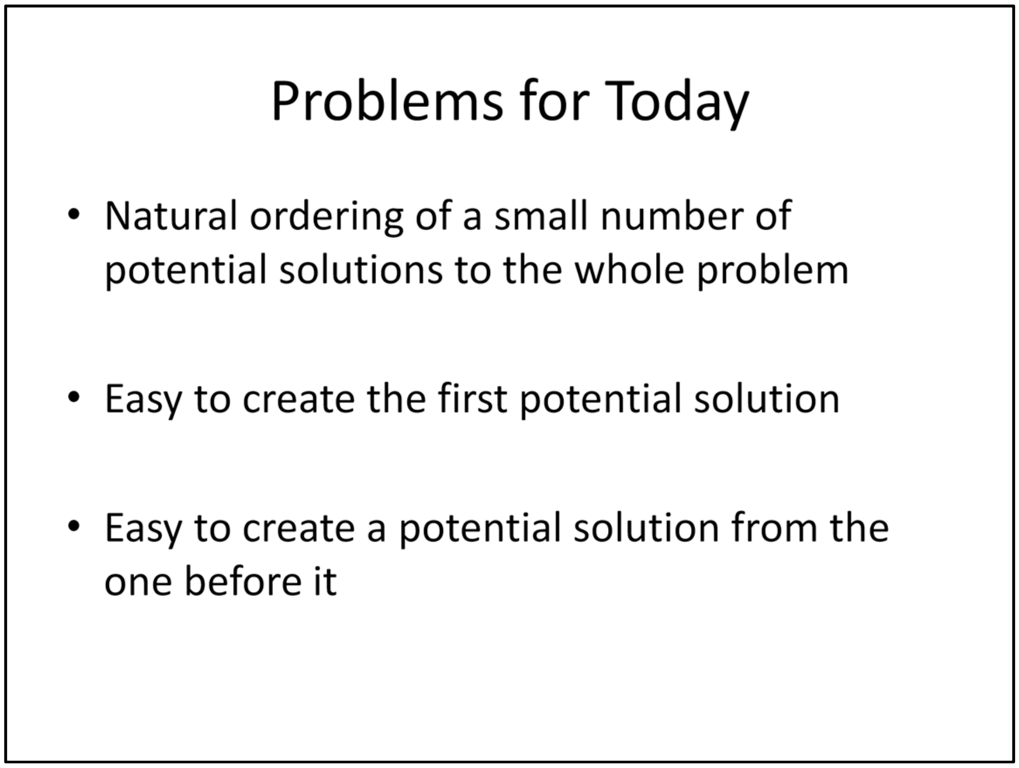
Hi everyone. Today we're going to cover three problems, which is a little ambitious, but as we learned in lecture, greedy algorithms may be hard to come up with and hard to justify, but they tend to be very easy to code.



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

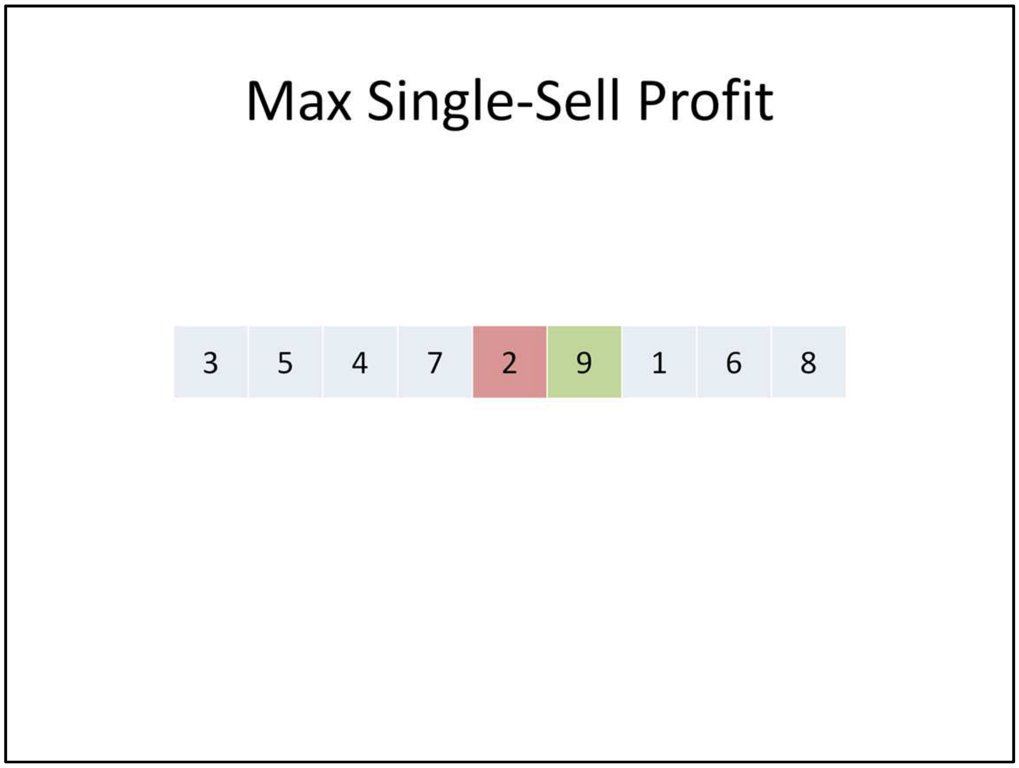
Strictly speaking, the three problems we're gonna cover today aren't actually greedy. However, they share a lot of ideas, proof techniques, and coding techniques with greedy problems. So what makes a problem greedy? The important thing is that we're able to break the problem down into a bunch of smaller 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.



Problems 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

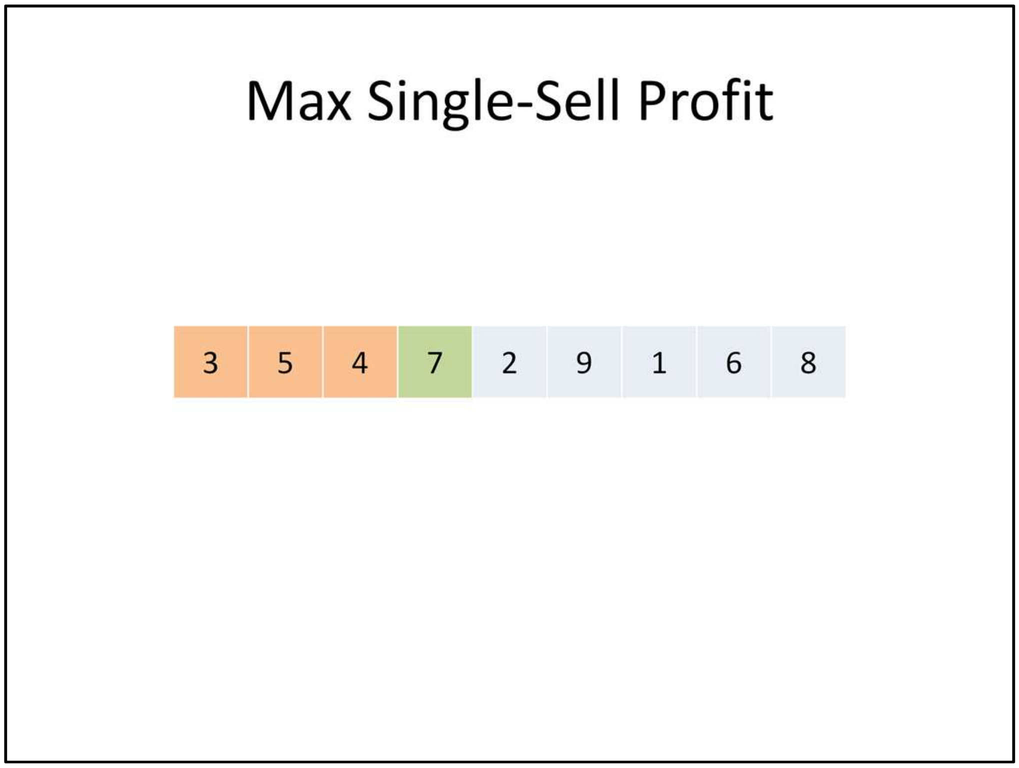
The problems for today are going to be a slight variation on that. In these problems, we're going to be able to identify a small number of potential solutions to the whole problem that we can order in some way. Then we're going to find an efficient way to start from one potential solution and iterate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



Max Single-Sell Profit

3	5	4	7	2	9	1	6	8

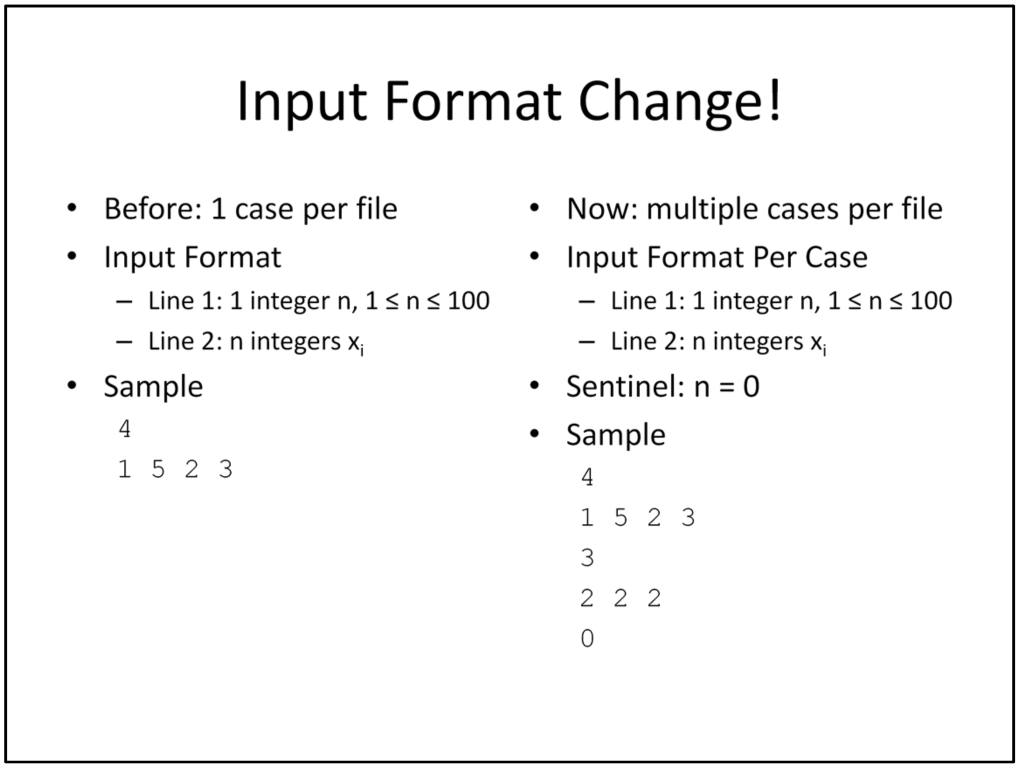
As a warmup, let's revisit the maximum single-sell profit problem that Keith discussed in the divide and conquer lectures. As a reminder, we can buy once, and then sell once afterward, and the profit we make is the price when we sell, minus the price when we buy. In this case, the maximum profit we can make is 7, and here's one way we can do it. We already know how to solve this problem in linear time using divide and conquer, but that solution ended up using logarithmic extra space due to the recursion. Let's take a look at another approach that will end up using less space.



Max Single-Sell Profit

3	5	4	7	2	9	1	6	8

First, let's notice that there are n possible different sell times, one for each price. Let's consider the profit we get if we commit to selling at a particular time, say when the price is 7 here. Given that we committed to this sell time, what's the best buy time for us? Well, it corresponds to the minimum element in the subarray up until the 7. Our profit is then 7 minus the minimum element we've seen up until the 7. So we can imagine this as n potential solutions, one for each of the possible sell times. Here, we'll define the solution to be both the sell time and the corresponding buy time. There's a natural ordering on these solutions, namely the chronological order. Now which of these solutions is easy to compute? (The first.) Now, given one solution, say the one corresponding to selling at 7, can we quickly compute the one corresponding to selling at 2? (Yes; all we need to do is to update the minimum price so far.) This means that we can walk over all the possible sell times and keep track of the best profit so far. When we're done, we report the best profit over all n solutions we considered. Note that computing the first solution took constant time, and moving to the next solution always took constant time, so we only used a linear amount of time. Even better, we only needed to track the minimum so far and the best profit so far, so we use a constant amount of extra space.



Input Format Change!

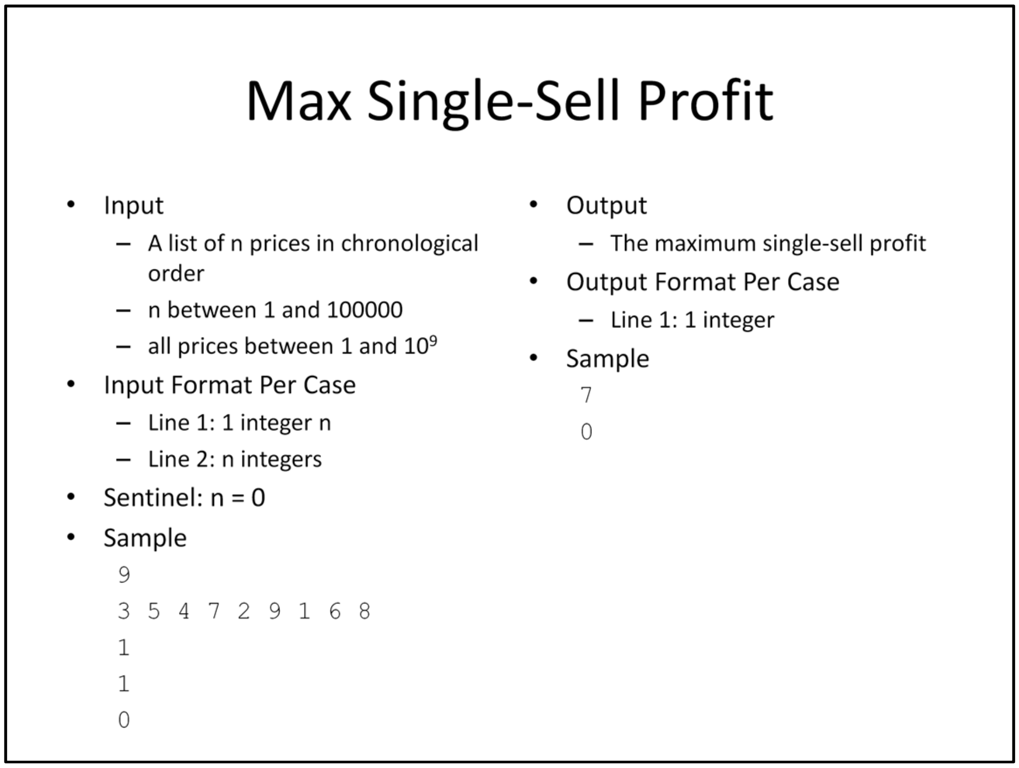
- Before: 1 case per file
- Input Format
  - Line 1: 1 integer n, $1 \le n \le 100$
  - Line 2: n integers x_i
- Sample

```
4
1 5 2 3
```

- Now: multiple cases per file
- Input Format Per Case
  - Line 1: 1 integer n, $1 \le n \le 100$
  - Line 2: n integers x_i
- Sentinel: n = 0
- Sample

```
4
1 5 2 3
3
2 2 2
0
```

Now, part of the reason I'm including a warmup problem is we're making a change to the input format. In previous sections we've only had one instance of the problem per data file, which made it somewhat unwieldy to test your program against all the data. This time the data has been grouped into far fewer files. Each problem is going to have one small input file and one large input file, and each file will have multiple test cases. This slide gives you an example of how the test cases will be represented; they'll always be terminated with sentinel values. This means you can wrap your algorithm's code around a big while loop that breaks when it reaches the sentinel case.



Max Single-Sell Profit

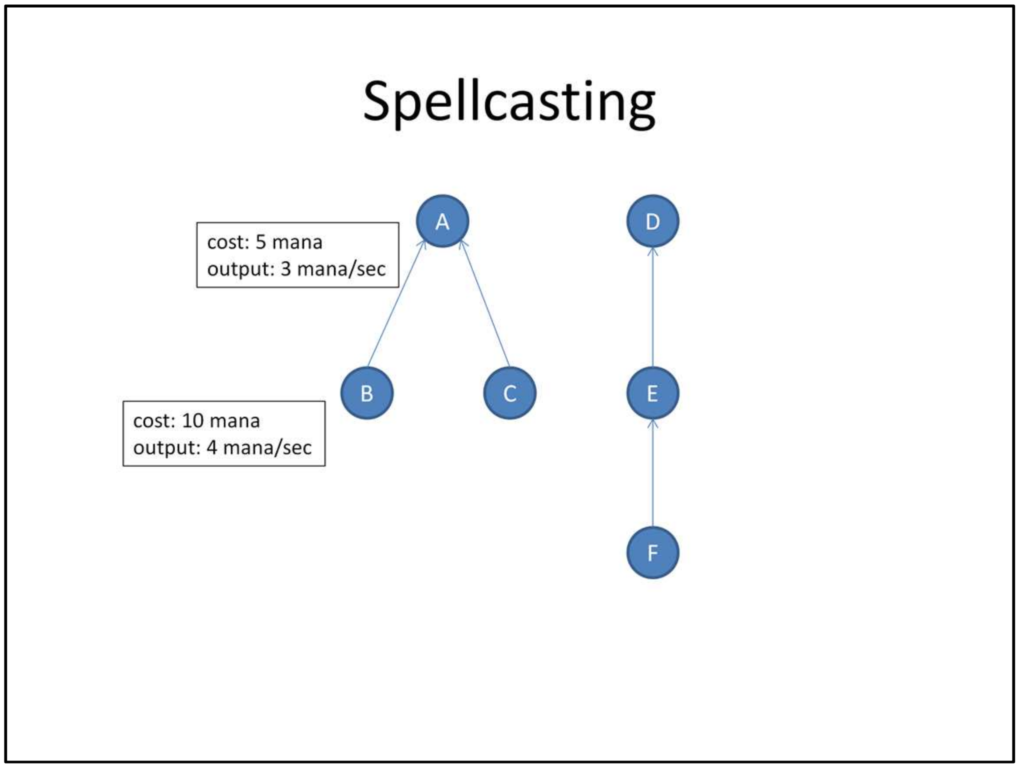
- Input
  - A list of n prices in chronological order
  - n between 1 and 100000
  - all prices between 1 and 10^9
- Input Format Per Case
  - Line 1: 1 integer n
  - Line 2: n integers
- Sentinel: n = 0
- Sample

```
9
3 5 4 7 2 9 1 6 8
1
1
0
```

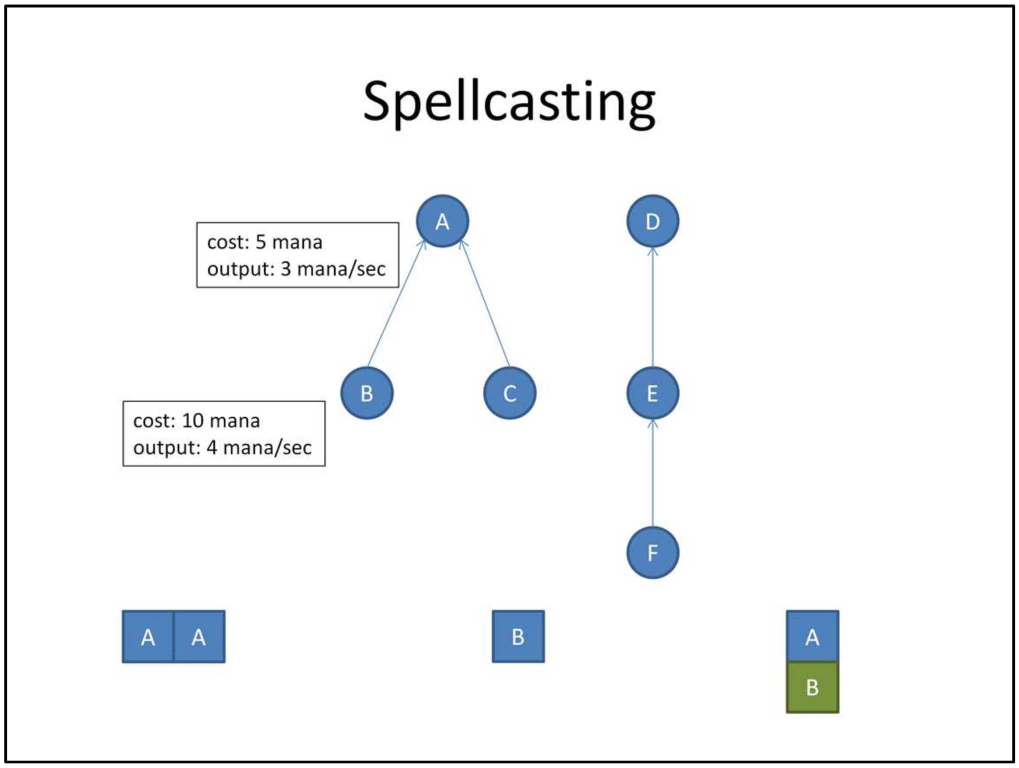
- Output
  - The maximum single-sell profit
- Output Format Per Case
  - Line 1: 1 integer
- Sample

```
7
0
```

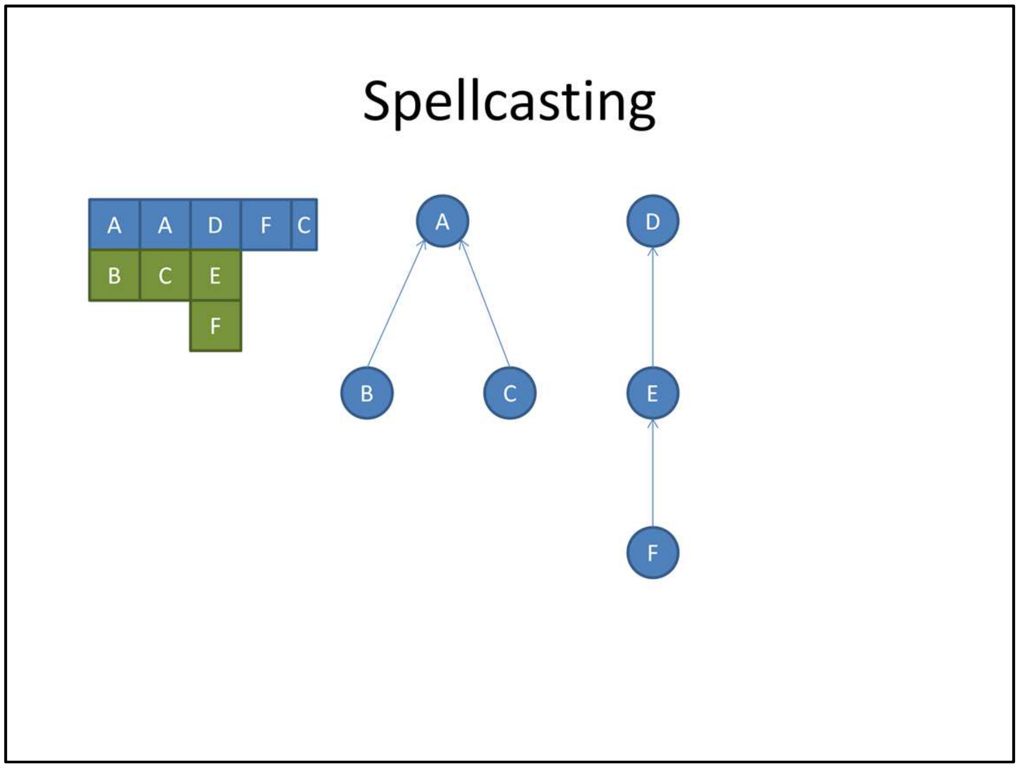
Now here's the specification slide for the maximum single-sell profit problem.



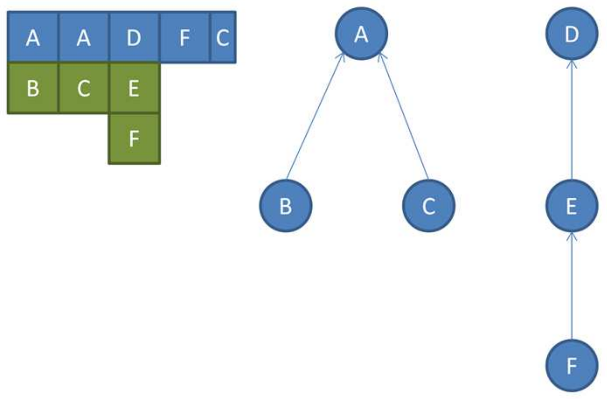
All right, on to the first real problem for today. There was a preview text file of this problem posted on the course website, but we'll go over it in detail now, so don't worry if you haven't seen it. You might want to open it up for reference, though. Anyhow, this problem is a little more whimsical in its flavor than we've done in the past; we're going to imagine that we're wizards casting a spell, and we do that by spending mana to produce elements that output mana continuously. We want to produce enough of various elements to meet a certain level of mana production as fast as possible. If you're morally opposed to solving problems that are based on fantasy, replace the word "spell" with "portfolio", "element" with "stock", and "mana" with "money" and the ideas still apply. But personally, finance and I don't get along, so...spellcasting. In any case, these elements each cost a certain amount of mana per unit and produce a certain amount of mana per unit per second.



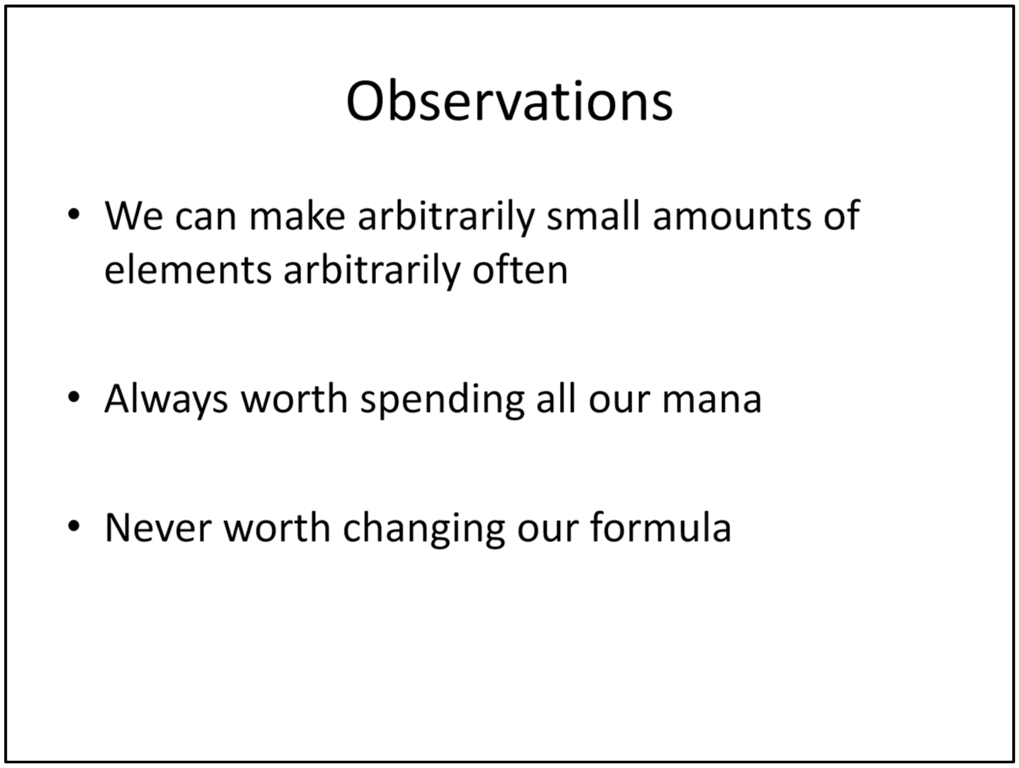
For example, if I had 10 mana, I could produce 2 units of element A, and then I would have an output of 6 mana per second. Alternatively, I could produce 1 unit of element B, and then I would have an output of 4 mana per second. I have one more option, and that comes from these arrows I've drawn between these elements. There can be up to one arrow coming from an element, and that points to an element that supports it. If we have a certain quantity of a supporting element, then we can produce up to that quantity of the supported element at half cost. This means I could produce 1 unit of A for 5 mana, and then use that to support 1 unit of B for half of 10, which is 5 mana. In this case, I would get an output of 7 mana per second. Notice that production happens instantaneously; as soon as I produce 1 unit of A, I can immediately use it to support the unit of B. The mana output also starts instantaneously; after 1 second has elapsed, I will have 7 mana that I can use to produce more elements. In fact, I don't have to wait a full second; I could wait five sevenths of a second, and I'd have the 5 mana I needed for another unit of A. There's one last detail to this problem: We also don't have to purchase elements in discrete units. This means with that 5 mana we just got, we could produce half a unit of A and half a unit of B if we so desired.



Spellcasting



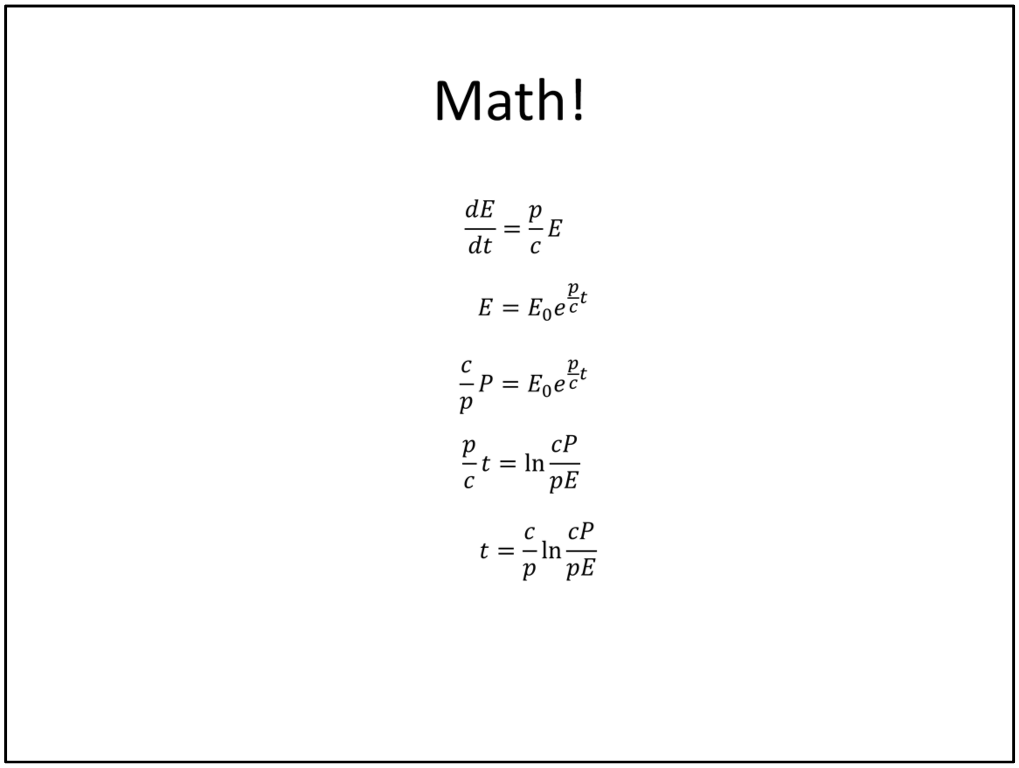
So after a certain amount of time has passed, the elements we've produced so far might look something like this, where we produced the blue elements at full cost and the green elements at half cost. Notice that if we produce element D first, we can use it to support element E, and then use the E we produced to support element F. Also notice that we can split our A between supporting elements B and C, though if we get too much of element C for A to support, then we have to pay the full cost again.
Notice that this problem allows us to make far more decisions than we have ever had available to us in problems we've discussed so far. At every point in time, we could produce any combination of elements in any proportion. That gives us an uncountably infinite number of potential solutions. Let's see if we can whittle that down to a more manageable set.



Observations

- We can make arbitrarily small amounts of elements arbitrarily often
- Always worth spending all our mana
- Never worth changing our formula

Interestingly enough, the fact that this is a continuous problem actually makes it easier. Because we can make arbitrarily small amounts of elements arbitrarily often, we never have a reason to save up mana. Instead, we should just spend it as soon as it comes in. So let's suppose that at a given point in time, we're producing elements in a specific proportion. We'll call these proportions a formula. A formula has a cost per unit output which depends on the proportions of elements in it. Notice that if at a given time a formula is optimal, it will always be optimal, because all formulas are usable at all times, and the cost per unit output doesn't depend on the amount of mana OR mana production on hand.



Math!

$$\frac{dE}{dt} = \frac{p}{c}E$$

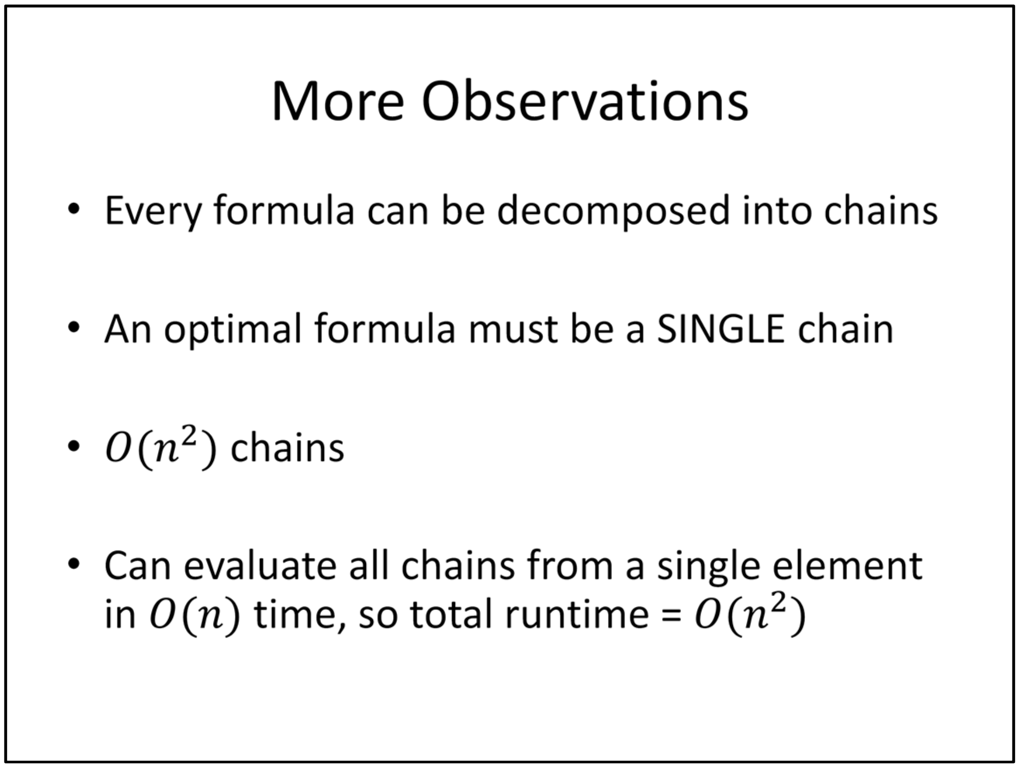
$$E = E_0 e^{\frac{p}{c}t}$$

$$\frac{c}{p}P = E_0 e^{\frac{p}{c}t}$$

$$\frac{p}{c}t = \ln\frac{cP}{pE}$$

$$t = \frac{c}{p}\ln\frac{cP}{pE}$$

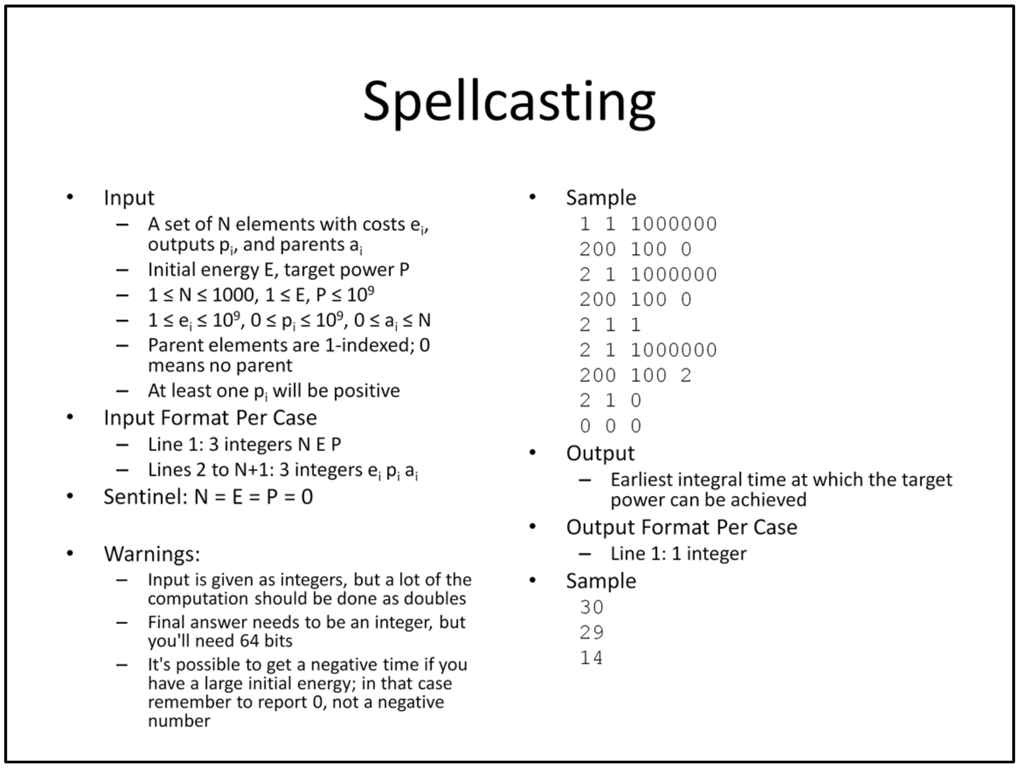
So now we've whittled down the decisions we have to make. We now just need to find the optimal formula, and use that formula until we reach our target output. Now, how fast do we reach that output? Well, let's suppose our formula costs lowercase c mana to produce lowercase p mana per second. Then we can solve a differential equation, which is NOT a prerequisite for this course, so you can just trust me on that step, to get an equation that tells us the amount of mana we have at any given time, provided we started with capital E sub 0 mana. We can then solve for the point at which we reach our target mana per second capital P, which confirms what we said earlier about the formula: We want to find the formula that minimizes the cost per unit output.



More Observations

- Every formula can be decomposed into chains
- An optimal formula must be a SINGLE chain
- $O(n^2)$ chains
- Can evaluate all chains from a single element in $O(n)$ time, so total runtime = $O(n^2)$

Unfortunately, there are still an infinite number of possible formulas, so we need to make some more observations. To help with that, let's define a chain of elements to be a list of elements that we produce in equal proportions, each of which is supported by the one that follows it in the list. For example, in our configuration earlier, getting equal amounts of B and A would be a chain, as would getting C in isolation, as well as getting equal amounts of F, E, and D. So a chain is a specific type of formula. We notice that every formula can be expressed as the sum of chains. The chains can be there in differing proportions, and they can overlap, but they're all chains. Now, let's suppose we decompose a formula that way. Each chain has its own cost per unit output. There has to be some chain with the minimum cost per unit output. Now let's look at any other chain; because all formulas are always available, we could take all the mana we invested into this chain and instead invest it into the minimum cost chain, which would give us a new formula that's at least as good as what we had before, but with one fewer chain in its decomposition. If we repeat this process, we can argue that there must be a minimum cost formula that consists of a single chain, namely, the minimum cost chain. OK, NOW we have managed to get down to a finite number of potential solutions, using an informal version of what you'll see called the Exchange Argument tomorrow in lecture. Notice that every chain starts at an element and ends at an element, so there are order n^2 of them. At this point, we can see that there's a cubic solution, namely to check all n^2 chains, each of which can have up to n elements. But we can take advantage of an ordering among the chains to do even better. Notice that if we start with a chain of one element and then add supporting elements one at a time, we can evaluate the cost per unit output of all chains that start at a give element in order n time. This means that the total runtime of checking all the chains to find the minimum cost per unit output is quadratic.



Spellcasting

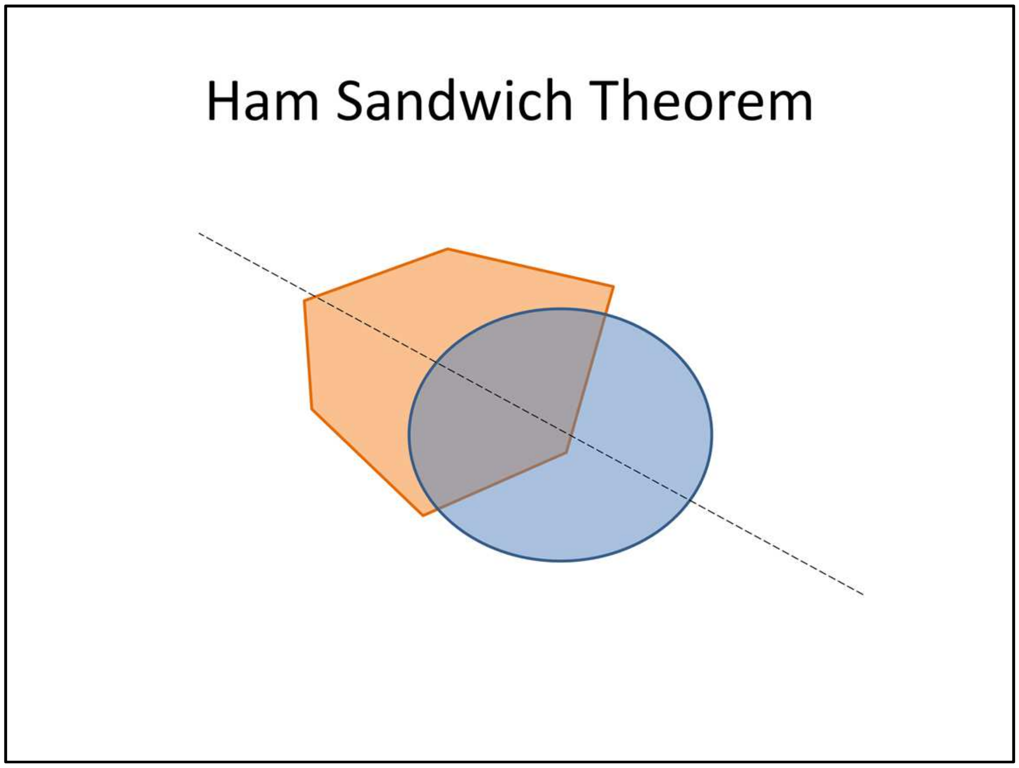
- Input
  - A set of N elements with costs e_i, outputs p_i, and parents a_i
  - Initial energy E, target power P
  - $1 \le N \le 1000$, $1 \le E, P \le 10^9$
  - $1 \le e_i \le 10^9$, $0 \le p_i \le 10^9$, $0 \le a_i \le N$
  - Parent elements are 1-indexed; 0 means no parent
  - At least one p_i will be positive
- Input Format Per Case
  - Line 1: 3 integers N E P
  - Lines 2 to N+1: 3 integers e_i p_i a_i
- Sentinel: N = E = P = 0
- Warnings:
  - Input is given as integers, but a lot of the computation should be done as doubles
  - Final answer needs to be an integer, but you'll need 64 bits
  - It's possible to get a negative time if you have a large initial energy; in that case remember to report 0, not a negative number
- Sample

```
1 1 1000000
200 100 0
2 1 1000000
200 100 0
2 1 1
2 1 1000000
200 100 2
2 1 0
0 0 0
```

- Output
  - Earliest integral time at which the target power can be achieved
- Output Format Per Case
  - Line 1: 1 integer
- Sample

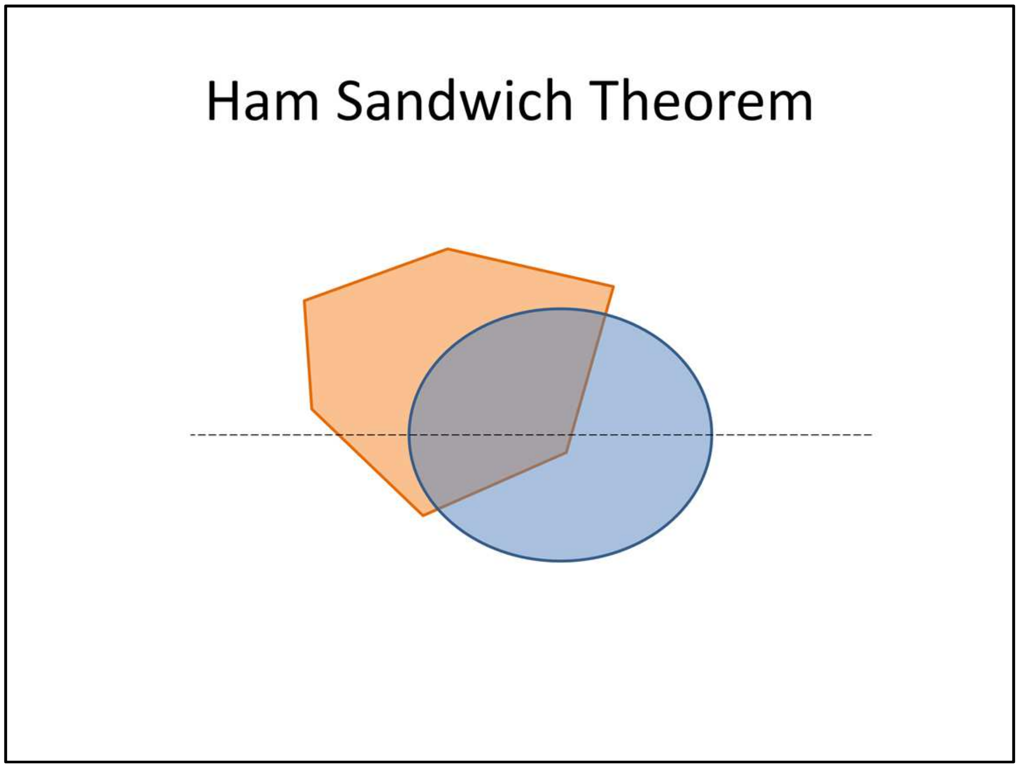
```
30
29
14
```

This specification slide is a little denser than what we've had before, which is why the handout printed this out larger than usual. Don't be intimidated by the length, though; the algorithm itself is really simple. I've included a couple of warnings about some edge cases that might cause problems.



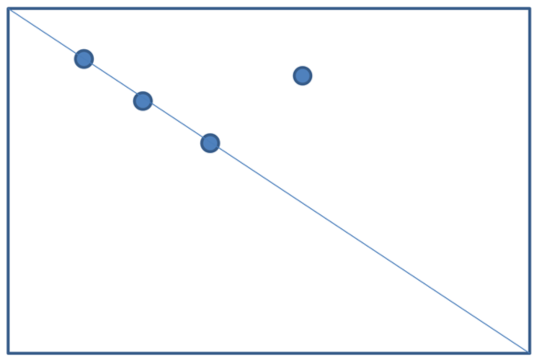
Ham Sandwich Theorem

All right, time to move on to the last problem of the day, 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?


Ham Sandwich Theorem

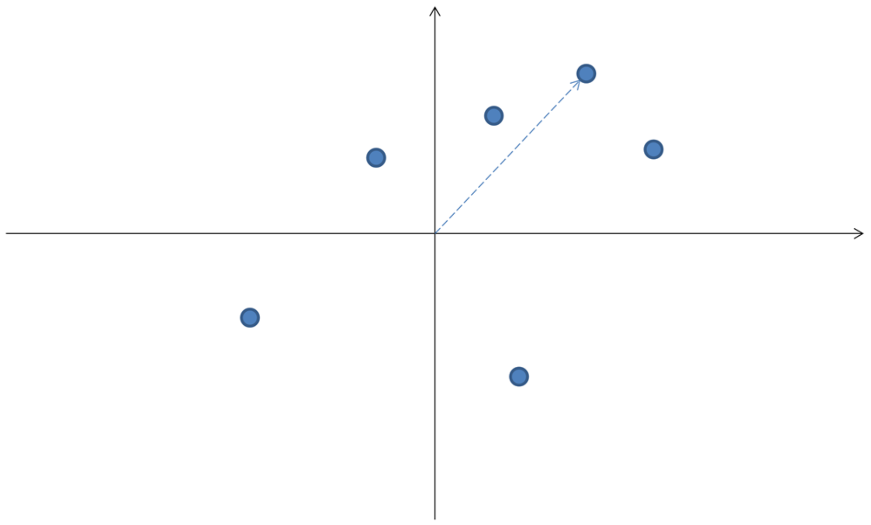
Well, let's imagine we cut one of the two objects in half along a specific direction. If we cut the other object in half, we're done; otherwise, more than half is on one side and less than half is on the other side. Now let's vary the direction continuously; we can imagine effectively rotating this line 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 two 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.

Partition

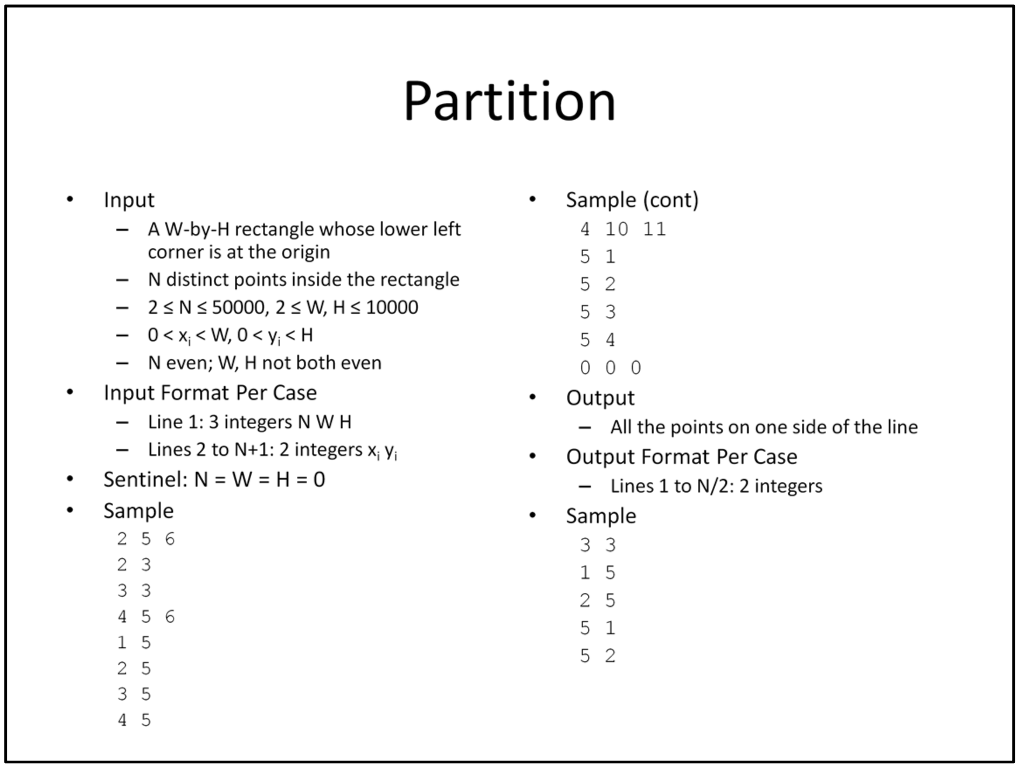


Now let's move on to the problem that we want to solve. We have a rectangle, and we have an even number of points inside the rectangle. We want to partition the rectangle AND the point set in half with a single straight line. Notice that we can choose to which half to assign each point that lies exactly 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 just like before, 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 just like with the last problem, let's see if we can find an ordering of these lines that we can use to speed up our evaluation.

Angular Sort



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.



Partition

- Input
  - A W-by-H rectangle whose lower left corner is at the origin
  - N distinct points inside the rectangle
  - $2 \le N \le 50000$, $2 \le W, H \le 10000$
  - $0 < x_i < W$, $0 < y_i < H$
  - N even; W, H not both even
- Input Format Per Case
  - Line 1: 3 integers N W H
  - Lines 2 to N+1: 2 integers x_i y_i
- Sentinel: N = W = H = 0
- Sample

```
2 5 6
2 3
3 3
4 5 6
1 5
2 5
3 5
4 5
```

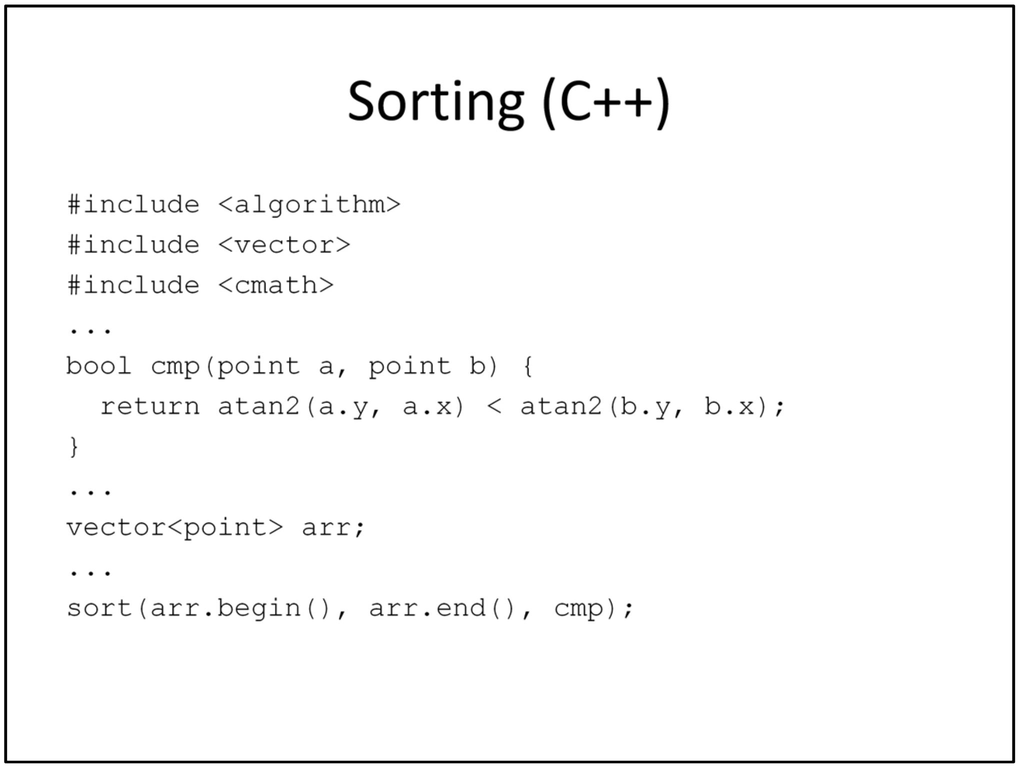
- Sample (cont)

```
4 10 11
5 1
5 2
5 3
5 4
0 0 0
```

- Output
  - All the points on one side of the line
- Output Format Per Case
  - Lines 1 to N/2: 2 integers
- Sample

```
3 3
1 5
2 5
5 1
5 2
```

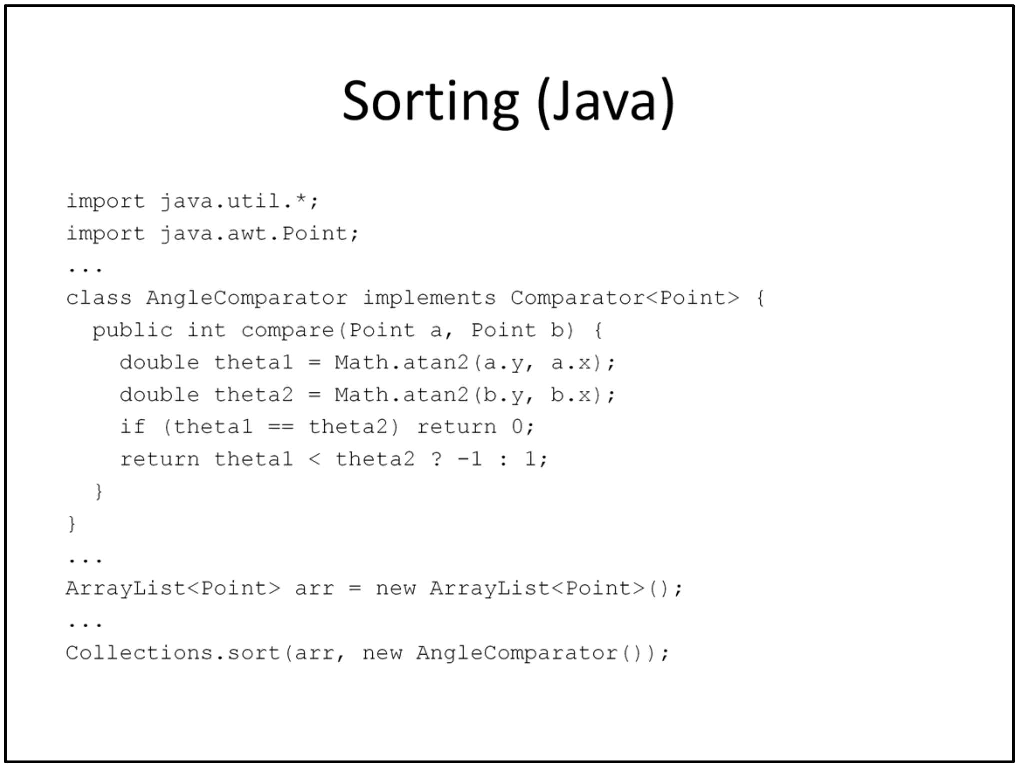
So now, here's the specification slide for the point set partition problem. Notice that the rectangle as given is NOT centered at the origin. You may find it convenient to translate all the input so that the center of the rectangle is the origin. Also notice that the problem bounds guarantee that there is no point in the center of the rectangle.



Sorting (C++)

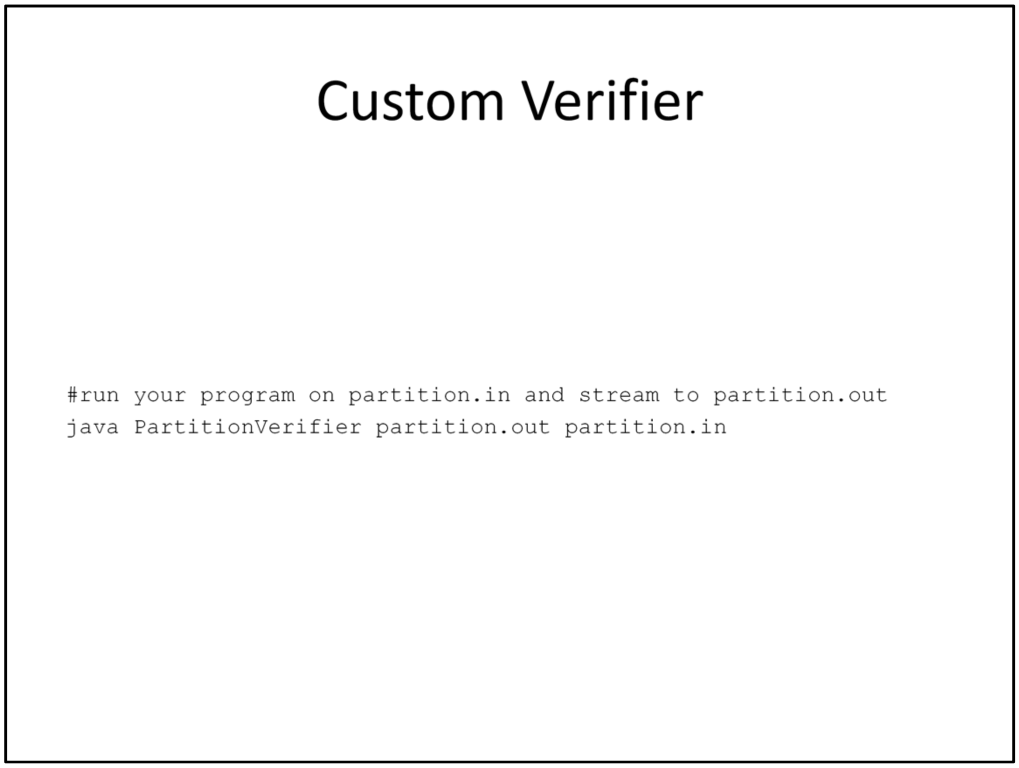
```
#include <algorithm>
#include <vector>
#include <cmath>
...
bool cmp(point a, point b) {
  return atan2(a.y, a.x) < atan2(b.y, b.x);
}
...
vector<point> arr;
...
sort(arr.begin(), arr.end(), cmp);
```

Now, last week you folks paid your due by coding up QuickSort from scratch. From now on, you're more than welcome to use your language libraries for sorting. Here's a sample of how you would sort your points by angle in C++.


Sorting (Java)

```
import java.util.*;
import java.awt.Point;
...
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;
  }
}
...
ArrayList<Point> arr = new ArrayList<Point>();
...
Collections.sort(arr, new AngleComparator());
```

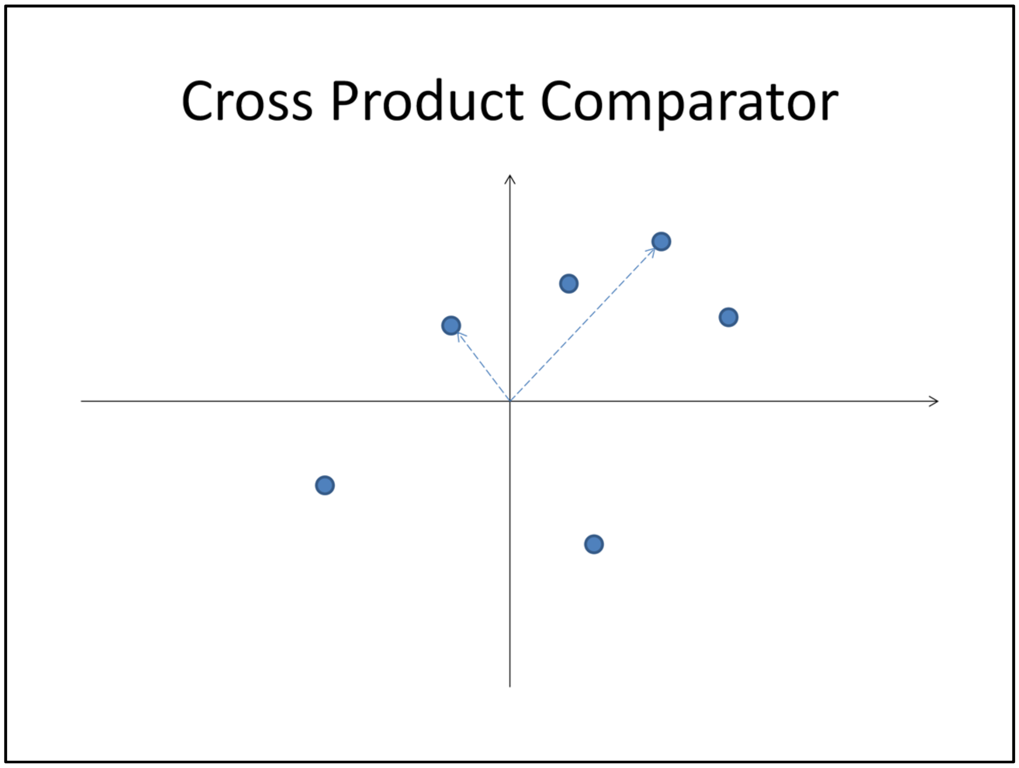
And here's how you would do so in Java. I didn't manage to get these slides onto the handout, so I've written them up on the board behind me.



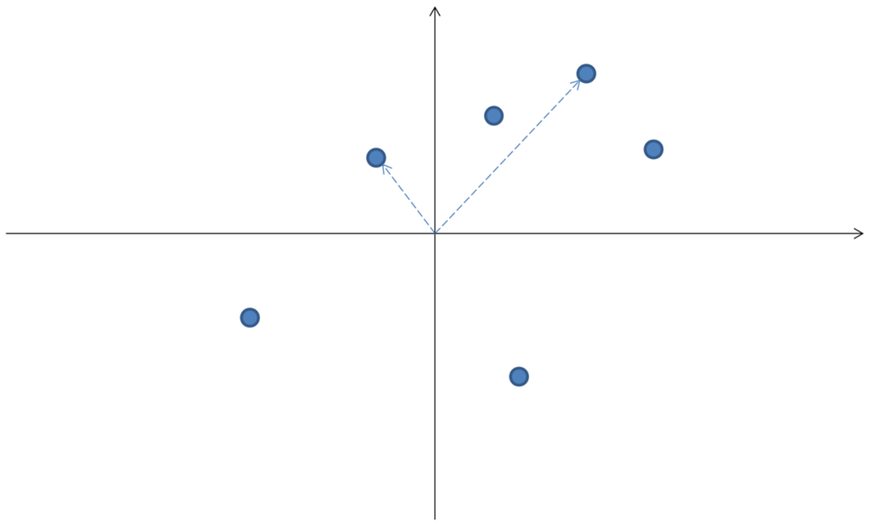
Custom Verifier

```
#run your program on partition.in and stream to partition.out
java PartitionVerifier partition.out partition.in
```

One last note before I let you guys go off and code. This problem is a little different from the ones we've covered previously in that there are multiple correct solutions. That means you can't just check your output against a premade output file to see whether it's correct. Instead, I've written up a verifier program where you pass in the input file and your program's output file and it tells you whether your output is correct. To use it, just invoke this command in the week5 directory, which has all the data as well as this verifier program.



Cross Product Comparator



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.