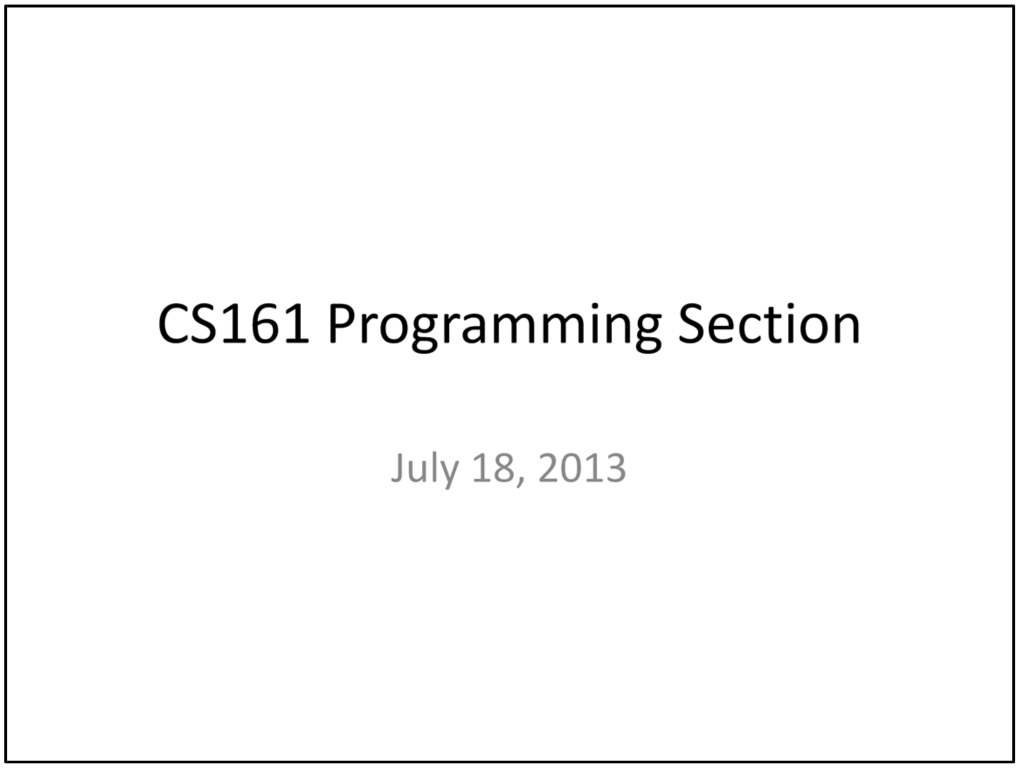
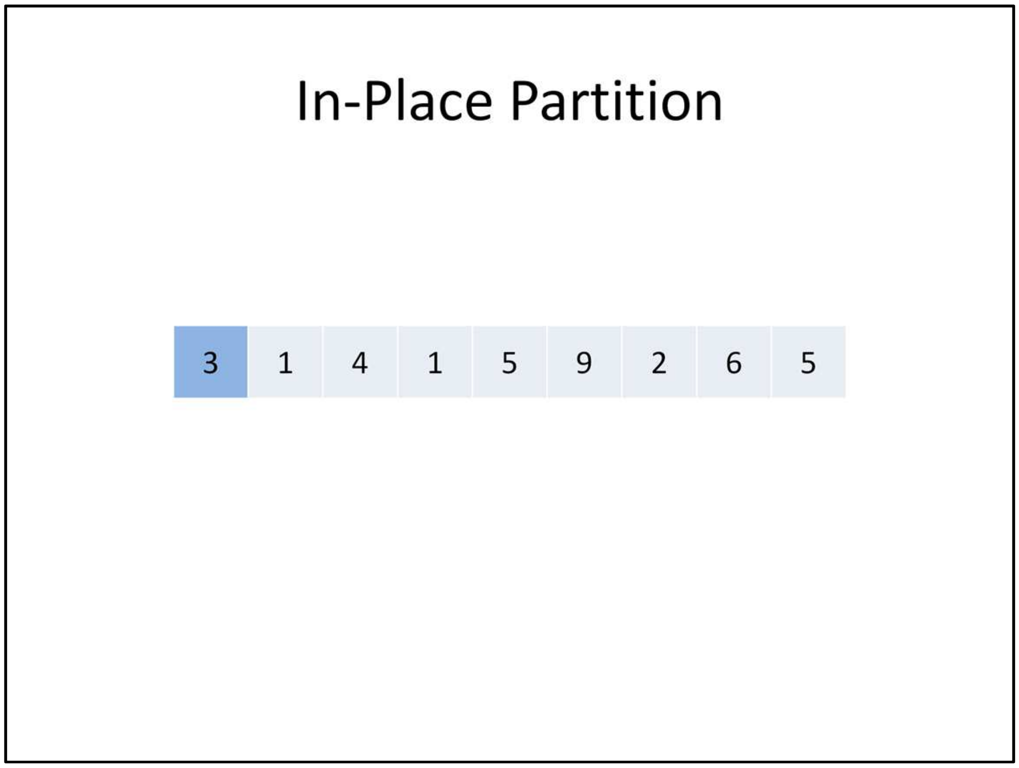


CS161 Programming Section

July 18, 2013

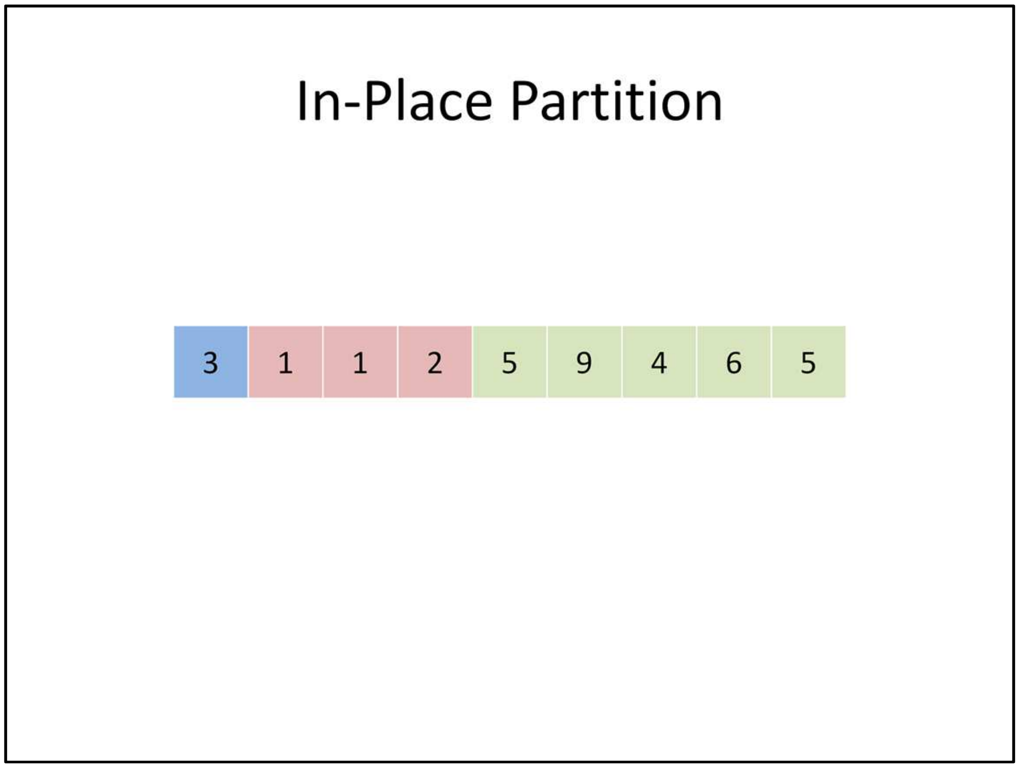
Hi everyone. Today we'll be covering randomized algorithms, implementing one Las Vegas algorithm and one Monte Carlo algorithm. The problems I've picked out for today don't require much code, so there's a good chance we'll finish early. I promise this has nothing to do with the fact that I'm behind on grading. None whatsoever. Nope.



In-Place Partition

3	1	4	1	5	9	2	6	5

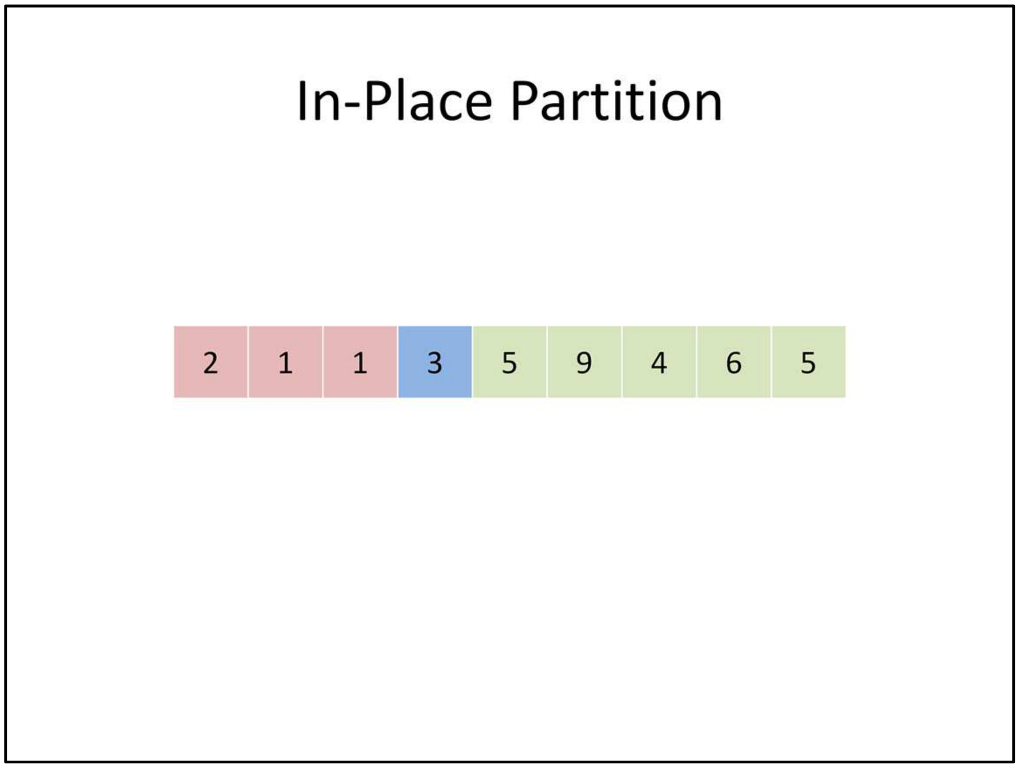
Anyhow, for our Las Vegas algorithm, we're pretty much obligated to implement Quicksort. Before we start coding, though, let's take a bit of time to figure out why Quicksort beats the other n log n sorting algorithms in practice. One advantage Quicksort has over Mergesort, for example, is that Quicksort can be done in place. This means we don't have to copy pieces of our array around like we had to in order to make the merge step of Mergesort work. All of the complexity of making Quicksort run in place is found in the partition step, so let's go over how to perform an in-place partition. Here, let's suppose that we want to partition using the first element, 3, as our pivot. Now, I'd like to refer you all to Keith's Problem Set Advice handout, which covered how to partition an array in place based on a predicate. We're going to use that algorithm here. Our predicate will be true if the array element is bigger than the pivot, and false otherwise.



In-Place Partition

3	1	1	2	5	9	4	6	5

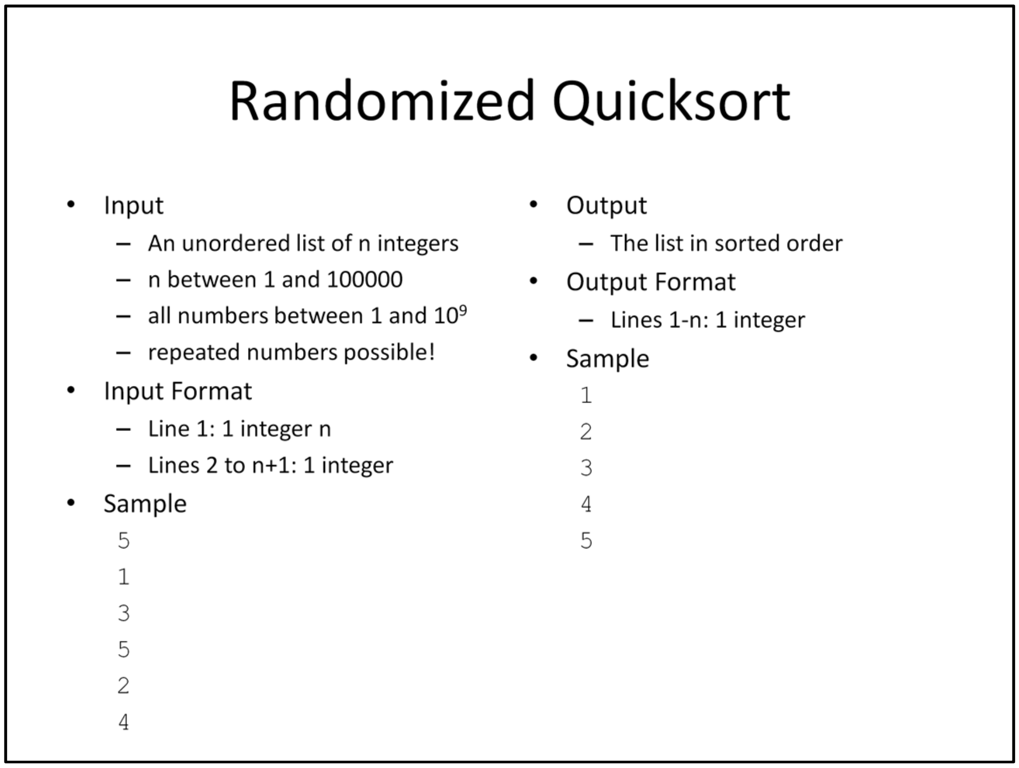
Once we're done using that algorithm, our array will look something like this. The first element will be our pivot, followed by a block of elements that are all NO larger than our pivot, followed by all the elements that ARE larger than our pivot. Then all we need to do to put our pivot in the right place is to swap it with the last element of the red block,



In-Place Partition

2	1	1	3	5	9	4	6	5

like so. Now, what if we want to choose a pivot besides the first element in the array? Well, before we run this partitioning algorithm, we can simply swap the pivot we want with the first element in the array.



Randomized Quicksort

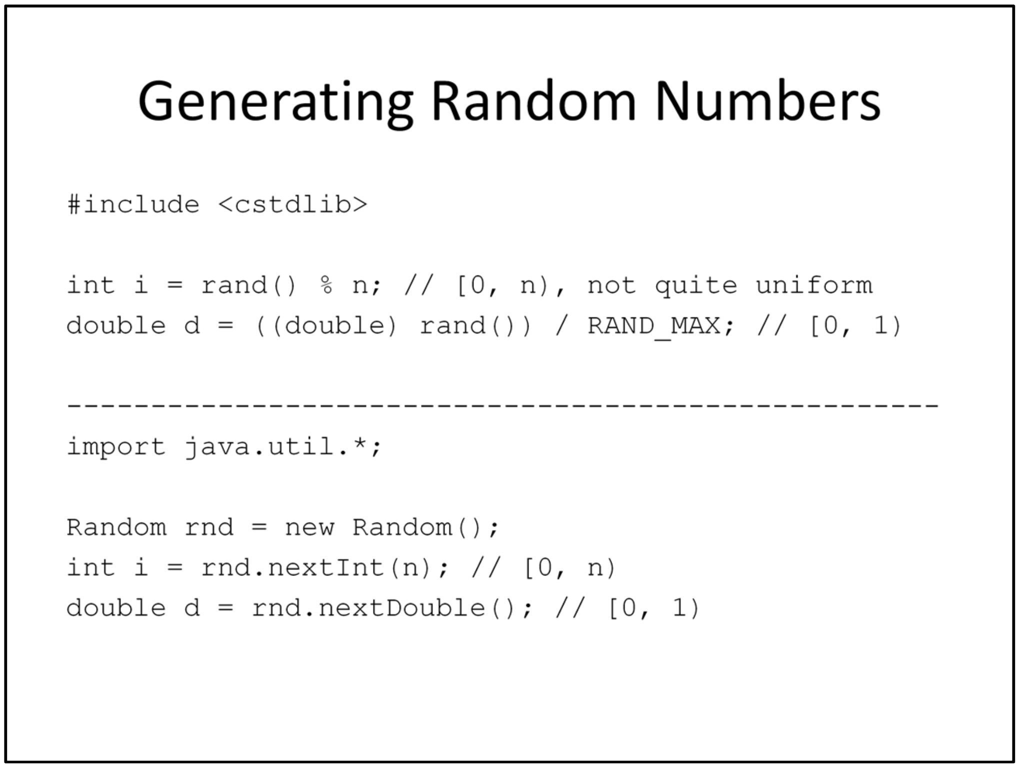
- Input
  - An unordered list of n integers
  - n between 1 and 100000
  - all numbers between 1 and 10^9
  - repeated numbers possible!
- Input Format
  - Line 1: 1 integer n
  - Lines 2 to n+1: 1 integer
- Sample

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

- Output
  - The list in sorted order
- Output Format
  - Lines 1-n: 1 integer
- Sample

```
1
2
3
4
5
```

With this in-place partition algorithm under your belt, implementing the rest of Quicksort should be easy. Here's the problem spec slide. Notice that we ARE going to have to deal with repeated numbers.



Generating Random Numbers

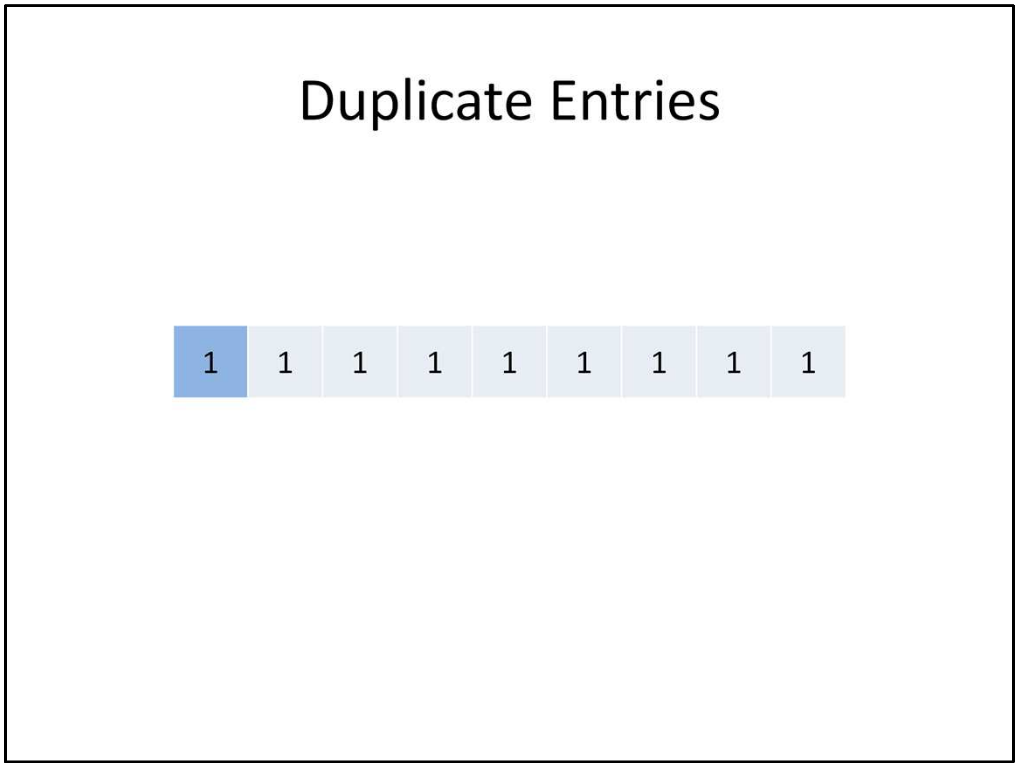
```
#include <cstdlib>

int i = rand() % n; // [0, n), not quite uniform
double d = ((double) rand()) / RAND_MAX; // [0, 1)

----------------------------------------------------
import java.util.*;

Random rnd = new Random();
int i = rnd.nextInt(n); // [0, n)
double d = rnd.nextDouble(); // [0, 1)
```

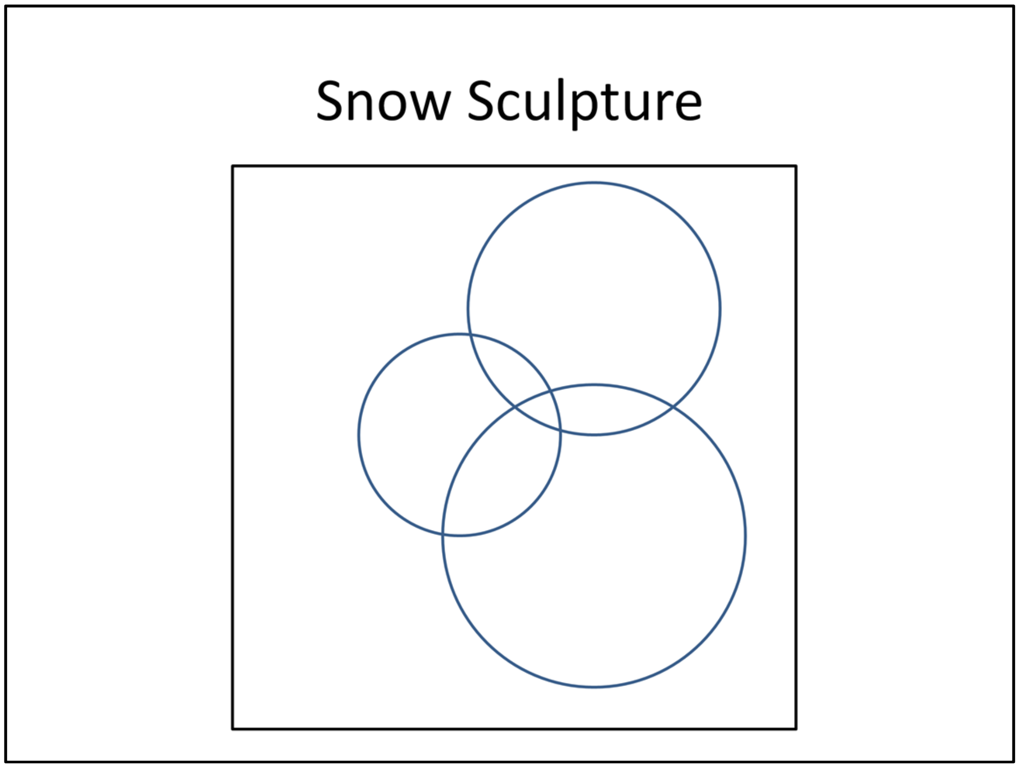
Now, you're going to be choosing your pivots at random. I've included this slide as a quick reference for how to generate a random integer between 0 and n, as well as how to generate a random floating point number between 0 and 1. As usual, C++ is on top, and Java is on the bottom. You might notice that the top is actually C code. There IS a random library in C++'s STL, but it's far more than what we'll need for today.



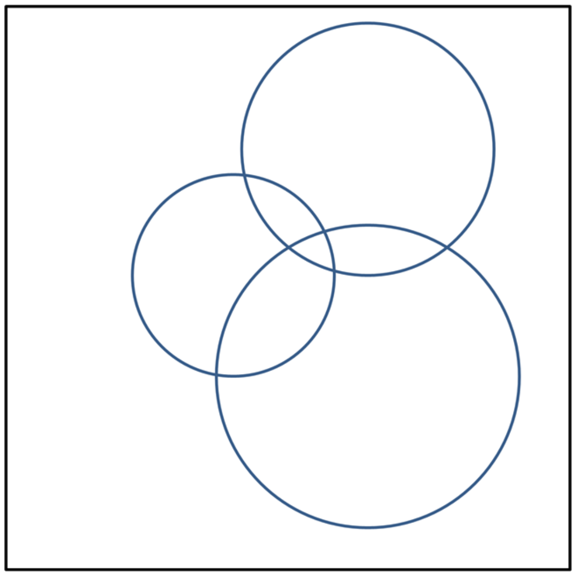
Duplicate Entries

1	1	1	1	1	1	1	1	1

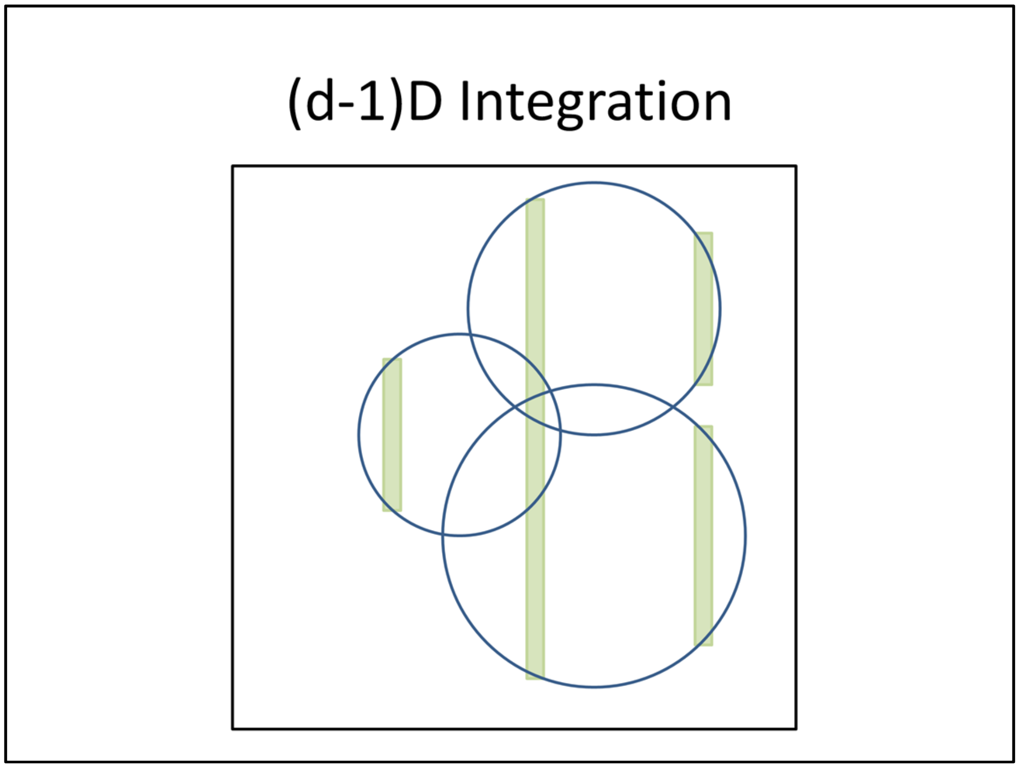
Now, the test data I've provided you with for Quicksort includes a test case that looks something like this. This case is actually kind of annoying for Quicksort to handle, because no matter what pivot you choose, you'll always end up with a bad split. That is, unless you modify the predicate you use in your partition subroutine. What you want to do in this case is whenever you run into entries that are identical to your pivot, you want to send roughly half of those entries to the left and the other half to the right. I'll let you figure out how to do this on your own. One trick that you might be tempted to try is to make a coin flip every time you come across an entry that matches your pivot. Keep in mind, though, that generating random numbers is expensive, so see if you can come up with something less computationally intensive.



Snow Sculpture

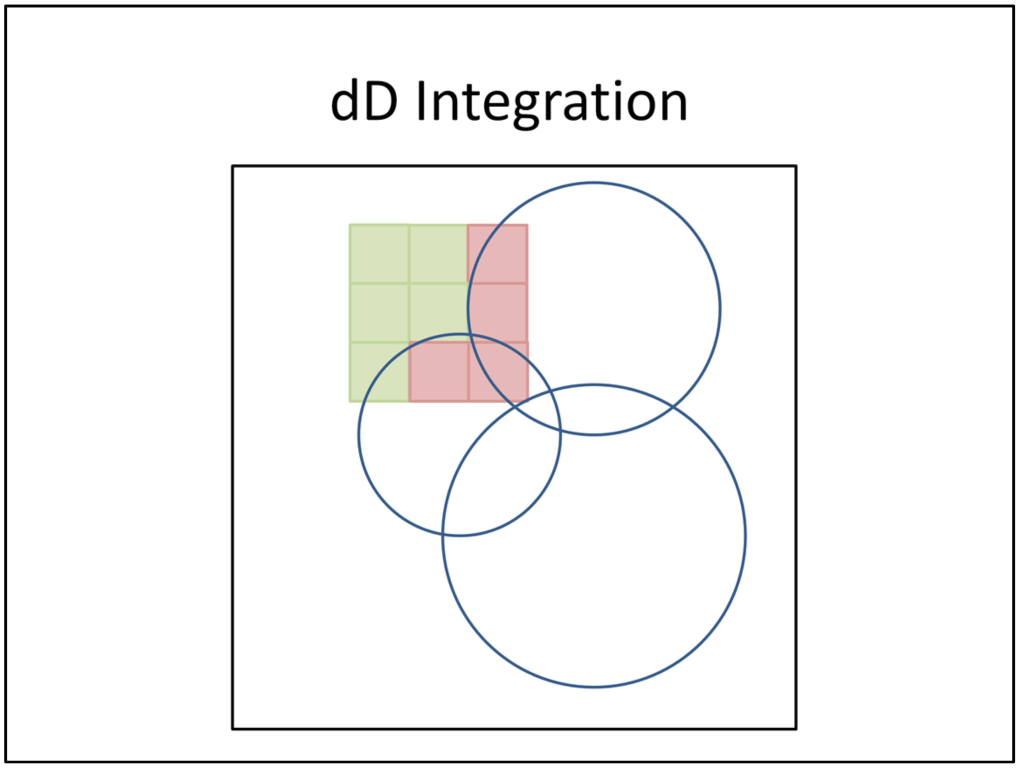


Now for our Monte Carlo problem. This problem is actually a three-dimension problem, but since that's hard to make slides for, all of these slides are going to be in two dimensions. In this problem, we can imagine making a snow sculpture out of a bunch of spherical snowballs, potentially overlapping, and we want to know how much snow is in the sculpture. In other words, we want to compute the volume of the union of a bunch of spheres. Now, how would we do this? You're welcome to try to work out a closed-form solution to this. It turns out it's already a pain with 2 spheres, and it only gets worse from there.

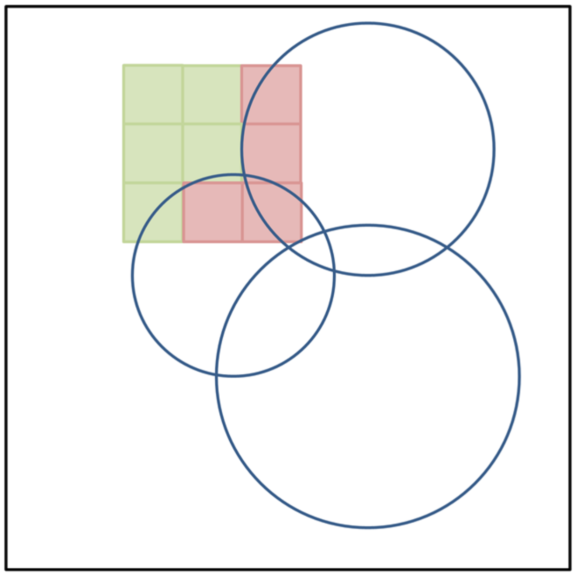


(d-1)D Integration

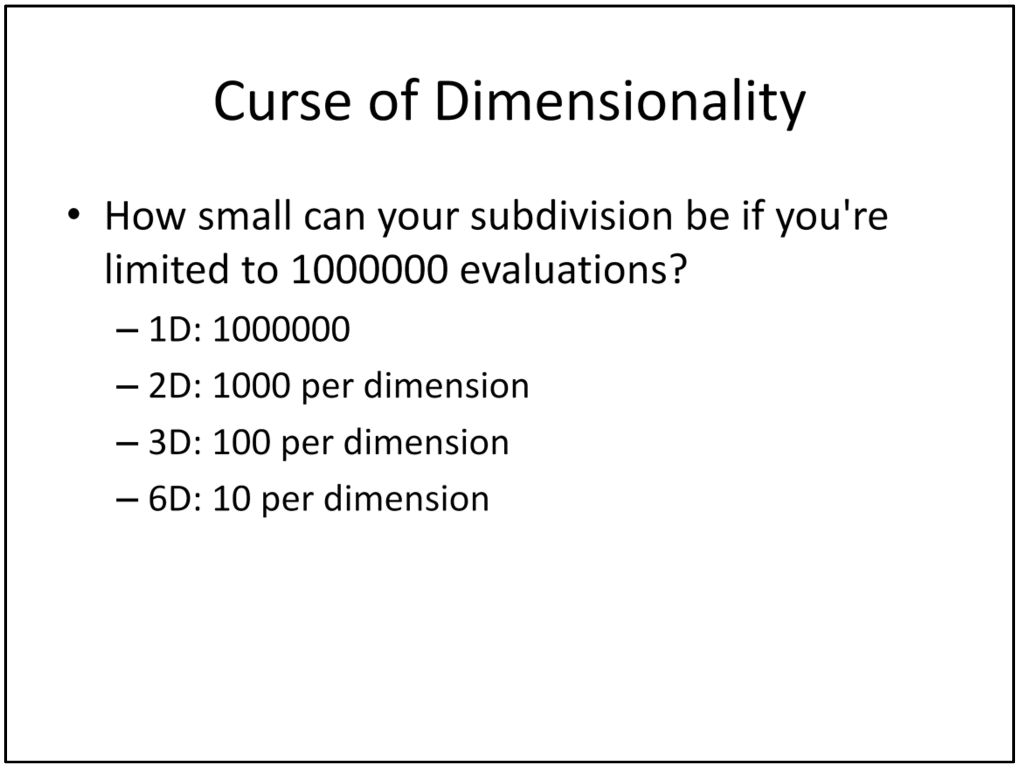
Well, let's see if we can approximate it. We could try to compute a Riemann sum. But it turns out this is still a hairy problem, because we need to figure out where each of the spheres begins and ends in each slice we make.



dD Integration



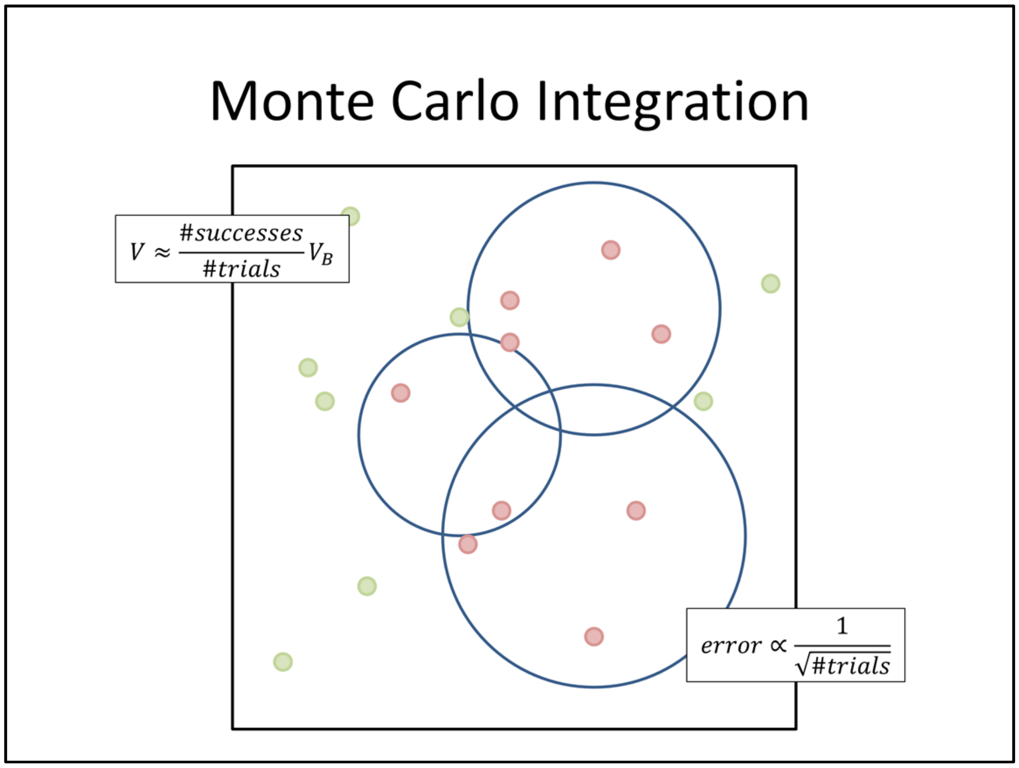
So let's try going up a dimension and integrating there. One thing we can do is we can divide our space into a bunch of cells, and then test the center of each cell to see whether it's inside our sculpture. Then we just add up the volumes of the cells whose centers are inside our sculpture. That gets rid of a lot of the nasty math, since testing whether a point is in the sculpture is equivalent to testing whether it's in any of the spheres. This is something we could implement, but before we commit to it, we should stop and think about how accurate this answer is going to be. If we wanted to only test a million points, we'd need to split our volume into a 100x100x100 grid. The issue is that while a million points sounds like a lot, our grid ends up being pretty coarse because we have to split them across 3 dimensions.



Curse of Dimensionality

- How small can your subdivision be if you're limited to 1000000 evaluations?
  - 1D: 1000000
  - 2D: 1000 per dimension
  - 3D: 100 per dimension
  - 6D: 10 per dimension

This issue comes up a lot in geometric problems and is known as the curse of dimensionality. This curse says that as the dimension of your problem goes up, either your runtime explodes, or your accuracy gets shot. Now the question is, can we get around this curse of dimensionality for this particular problem?

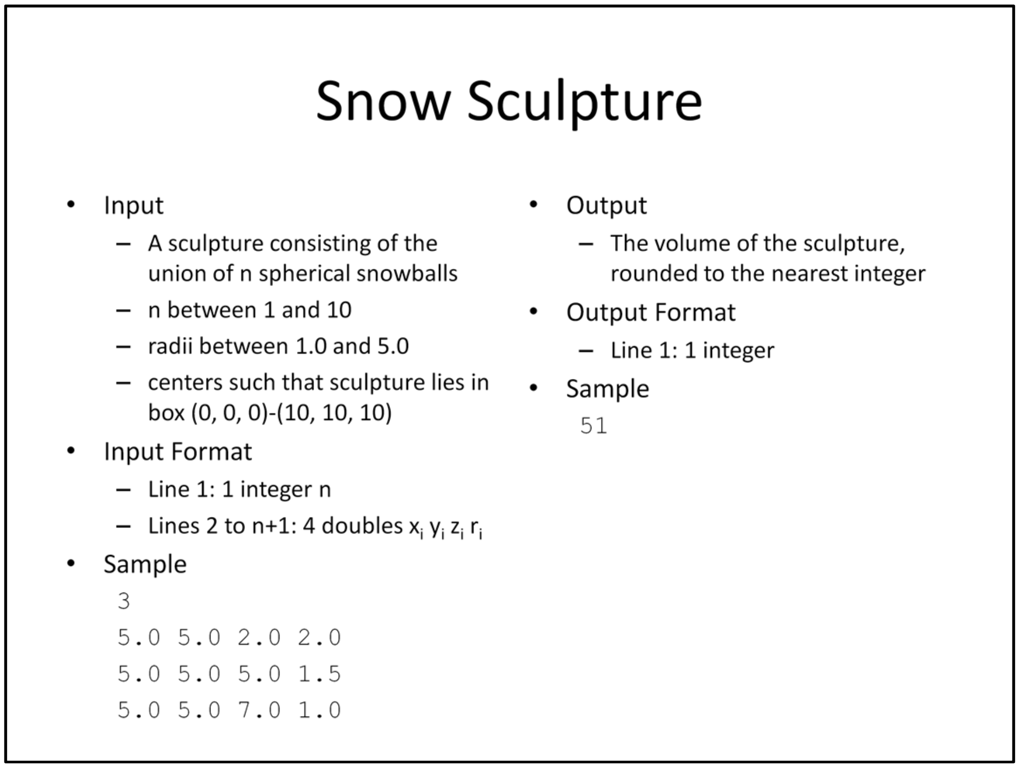


Monte Carlo Integration

$$V \approx \frac{\#successes}{\#trials} V_B$$

$$error \propto \frac{1}{\sqrt{\#trials}}$$

It turns out there is, at least to some extent. There's another way of integrating using a Monte Carlo randomization scheme. Choose a point at random inside a bounding box of the volume you care about. Then check whether it actually lies inside the sculpture. Imagine a random variable equal to 0 if you land outside the sculpture, and 1 if you land inside it. Notice that the ratio of the volume you care about to the volume of the bounding box is exactly the expectation of this random variable. What this means is you can approximate the volume of the sculpture by just running this test over and over again, and averaging the results. If you apply a lot of math that's beyond the scope of this class, you'll be able to see that the relative error of this approximation is inversely proportional to the square root of the number of trials. Notice that this is independent of the dimensionality of the problem.



Snow Sculpture

- Input
  - A sculpture consisting of the union of n spherical snowballs
  - n between 1 and 10
  - radii between 1.0 and 5.0
  - centers such that sculpture lies in box (0, 0, 0)-(10, 10, 10)
- Input Format
  - Line 1: 1 integer n
  - Lines 2 to n+1: 4 doubles x_i y_i z_i r_i
- Sample

```
3
5.0 5.0 2.0 2.0
5.0 5.0 5.0 1.5
5.0 5.0 7.0 1.0
```

- Output
  - The volume of the sculpture, rounded to the nearest integer
- Output Format
  - Line 1: 1 integer
- Sample

```
51
```

So here's the specification for the snow sculpture problem. Here, we've guaranteed that the sculpture will live inside a particular 10x10x10 box, which you can use as your bounding box for your Monte Carlo integration. This problem is a little different from the other problems we've had before in that you're not guaranteed to match the output files we gave you even if your algorithm is implemented correctly. You're going to need to experiment with different numbers of trials to see how many trials you need in order to reliably match the output files. It may help to print out the unrounded volume estimates while you're experimenting to see how much variation there is in your answer.