# CpSc 2120: Algorithms and Data Structures

**Instructor:** Dr. Brian Dean — Fall 2020
**Webpage:** `http://www.cs.clemson.edu/~bcdean/` — MWF 9:05-9:55
**Handout 7:** Lab #5 — Flour 132

## 1 Bringing A Pancake to a Sword Fight

Here in CpSc 2120, we pride ourselves on showcasing applications in our assignments that are of direct relevance to modern computing systems in practice. In keeping with this underlying goal, this week's lab addresses the important question of the fate of a stack of pancakes caught in the middle of an epic sword fight.

Specifically, given $N$ straight cuts across a circular pancake that happen as collateral damage during the fight, we would like to compute, as quickly as possible, the number of individual pieces into which our pancake will be divided. We hope this cutting-edge application strikes a chord with our students in terms of motivating the utility of clever algorithms and data structures[1].

A sample input file with $N = 50,000$ cuts is given in the lab directory:

`/group/course/cpsc212/f20/lab05/cuts.txt`

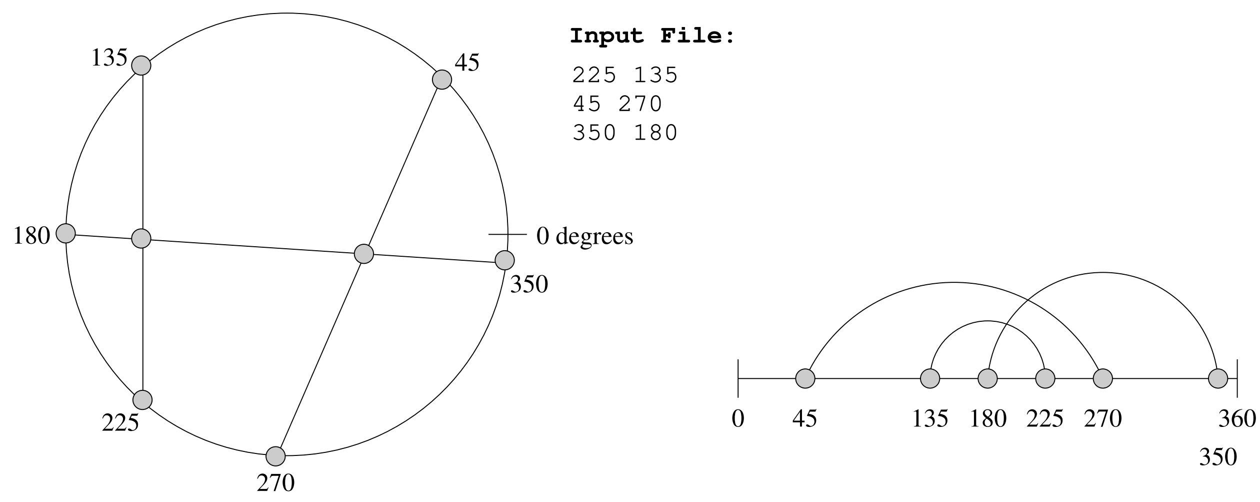
Each cut is specified by two doubles, giving the pair of angles at which the cut crosses the circle. Hence, the input to your program will consist of $N$ lines, each containing two doubles. Each angle is between 0 and 360 degrees, as shown in Figure 1. A convenient way to represent each cut is as a `pair<double, double>` object, with the `.first` and `.second` fields representing the two angles.

## 2 Goal One: An Easy $O(n^2)$ Solution as a Warm-Up

Let's make the simplifying assumption that every intersection point between two cuts is an intersection point between *only* those two cuts, not three or more. In this case, one can show as a consequence of Euler's formula (Euler is pronounced "Oiler") that the number of pieces into which the circular pancake is cut will be $1+N+P$, if $P$ is the total number of intersection points inside the circle (we'll go over the math behind this in lab). Thus, counting pieces essentially boils down to counting pairwise intersections between cuts. Your task for the first part of the lab is to determine $P$ in the straightforward way, by comparing all pairs of cuts, counting those that intersect inside the circle. This will run in $O(N^2)$ time.

Testing for intersection of two cuts within the circular pancake is easy to do if we "straighten out" the circle into a line segment, as shown in to the right in Figure 1, by breaking the circle at 0 degrees and pulling it straight. Each cut now turns into an interval, characterized by angles at

[1]The instructor is quite proud of the multiple puns in this sentence; you are encouraged to laugh at them.



**Input File:**

```
225 135
45 270
350 180
```

Figure 1: Example input with $N = 3$ cuts, resulting in $P = 2$ intersection points inside the circle. These cuts divide the circle into a total of 6 pieces.

which it begins and ends. For any pair of intervals, there are only three possible ways they can be related:

- They can "nest". For example, the interval $(135, 225)$ is nested completely within the interval $(45, 270)$ in the figure.
- They can be completely disjoint. No examples of this appear in the figure, but intervals $(1, 10)$ and $(20, 30)$ would be disjoint, since they do not overlap at all.
- The can "cross". In the figure, the $(180, 350)$ interval crosses both others. Pairs of intervals that cross correspond to pairs of cuts that make an intersection point inside the circle.

Your goal for this first part of lab is to write a program that reads the input file, counts the number of pairs of cuts that cross (i.e., determine the value of $P$), and prints out both $P$ as well as the resulting number of pieces, $1 + N + P$.

For the input file provided, the correct answer for $P$ is 415,199,866. Note that it may take your program a while to reach this number, since it is counting intersections one at a time.

## 3 Goal Two: Implementing `get_rank`

We now work towards an $O(N \log N)$ solution for our problem. To start with, make a copy of the balanced binary search tree code from lab 4, change the data type of keys stored in the tree from `int` to `double`, and add the (recursive, of course) implementation of one new function:

```
int get_rank(Node *root, double x)
{
   // returns the number of nodes in the tree with keys < x.
}
```

The value of $x$ does not need to exist as a key in the tree, although your code should still work fine even it does. Your `get_rank` function should run in $O(\log n)$ time on a balanced tree.

## 4 Goal Three: An $O(N \log N)$ Solution

To get started here, let's read in our input file into a vector of `pair<double, double>`'s, where each cut is actually represented twice — once as (angle1, angle2), once as (angle2, angle1):

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Make a simpler alias for the pair<double, double> type
typedef pair<double,double> cut;

int main(void)
{
   vector<cut> V;
   double angle1, angle2;
   while (cin >> angle1 >> angle2) {
      V.push_back(make_pair(angle1, angle2));
      V.push_back(make_pair(angle2, angle1));
   }
   sort(V.begin(), V.end());

   // Rest of solution to follow...
}
```

After reading in our input, we call the built-in `sort` command to sort our vector of cuts. The sort function runs in $O(N \log N)$ time (the fact that our vector has $2N$ entries doesn't matter here, since it just changes the constant factor in the $O(N \log N)$). Conveniently, `sort` knows how to sort pairs of things — it does so by sorting on the `.first` elements of each pair, only using the `.second` elements to break ties between elements that have the same `.first` values. In our case, all the `.first` values can be assumed to be distinct, so we are essentially just sorting all the pairs on their `.first` values.

The rest of our algorithm now scans through the entries in $V$ one by one, counting up intersections along the way in a careful fashion assisted by a balanced binary search tree $T$ that keeps track of all the "active" intervals at any given point in time (an interval is "active" if we've scanned its starting point but not yet its ending point). By putting each cut in our vector twice, we visit it once when its `.first` is the earlier endpoint on the line segment, and later when its `.first` is the later endpoint on the line segment. For example, the figure contains an interval $(180, 350)$. We think of visiting the 180 endpoint when we process the pair $(180, 350)$ in $V$, and we think of visiting the 350 endpoint when we later process the pair $(350, 180)$ in $V$. When we process the starting endpoint of an interval, we add the interval to our BST. When we process the ending endpoint, we remove it from our BST. Therefore, as we scan through $V$, at any given time our BST will represent only the "active" intervals. When we represent an interval in the BST, we store its ending endpoint. For example, the interval $(180, 350)$ would be represented by 350 in the BST.

Using the `get_rank` function in an appropriate way, finish the implementation described above so that it computes the value of $P$ during the scan through the vector $V$. Your algorithm should run in $O(N \log N)$ total time since it makes $N$ inserts, $N$ removals, and $N$ calls to `get_rank`, and all of these individual operations take $O(\log N)$ time each. We will go over the full details of the algorithm in lab.

Your code should print out $P$ as well as the number of pieces: $1 + N + P$. To check that you are getting the right output, these values should match the numbers printed by your slower program.

## 5 Grading

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Monday, October 5. No late submissions will be accepted.