LIEN TRAN
Math 107B - Spring 2002 - Final Project

# CAKE-CUTTING

## OBJECTIVE

The ability to fairly divide a cake among n-people is a great concern of individuals with siblings. Apparently, the problem of getting a fair share of cakes is more important than it sounds. This topic has moved beyond the kitchen and was introduced into the world of mathematics by Hugo Steinhaus on September 17, 1947. In this final, I will explore the methods in which a cake can be divided equally among n-people.

## SOURCES

1) www.mathworld.com
2) Cake-Cutting Algoriths: Be Fair If You Can
by Jack Robertson and William Webb

## WITH HELP FROM

Professor JOHN WAVRIK and STEFAN ERICKSON

## NOTE

This final contains two bit-map images that have been created by me.
So, please use a computer that supports bit-map image.
**The computers in the lab do not suppport bit-map images and overwrite the file leaving out the images.**
> **restart:**
>

In this section, I am exploring several possible algorithms that one can use to equally divide cakes.

## POSSIBLE ALGORITHMS FOR FAIR DIVISION OF CAKES

### CUT & CHOOSE algorithm (2 people)

Step1 - person1 cut the cake
Step2 - person2 chooses first

and the remaining piece goes to person1

**OUTCOMES (**This algorithm uses (n-1) cuts)

Person2 will always benefit despite the fact that the cake is or isn't divided equally. If person1 slips and cut one piece of the cake bigger than the other, person2 will benefit from choosing the bigger piece since person2 gets to choose first. Therefore, it is in the benefit of person1 to cut the cake as equally as he can.

DOESN'T CUT IT algorithm (3 people)

Step1 - person1 cuts a cake into two pieces so that
pieceA=1/3 of the cake and pieceB=2/3 of the cake

Step2 - person2 cuts pieceB into two equal pieces so that
pieceB1=1/3=pieceB2 of the cake

Step3 - person3 chooses one of three pieces (pieceA, pieceB1, or pieceB2)
then person1 chooses next
and the last piece goes to person2

**OUTCOMES (**This algorithm uses (n-1) cuts)

Person3 has the greatest advantage followed by person1 and person2, respectively. If person1 does not cut the cake into two pieces that are 1/3 and 2/3 of the cake, person3 will benefit from choosing the piece that is >1/3 and leaving the piece that is < 2/3 to be divided among person1 and peson2. In this case, person1 and person2 will not be able to get their fair share of the cake. In the second case, if person1 has cut the cake correctly but person2 does not divide the piece that is =2/3 of the cake into two equal pieces that are 1/3 each, then person3 will choose the piece that is >1/3, person1 will choose the piece that =1/3, and person2 is left with a piece that is <1/3. Therefore, person2 will always get the disadvantages unless both person1 and person2 cuts the cake evenly into three pieces that are =1/3 of the cake so that all three persons will be happy.

MOVING KNIFE algorithm (n people)

Step1 - person1 place the knife on the left end of the cake
and continuously moves the knife to the right
- any other player can say "STOP" when he believes that
the portion to the left of the knife is 1/n of the cake

and that piece is given to the player who said "STOP"
and the player is out
- if more than one players say "STOP,"
then the piece can go to any one of the players

Step2 to (n-1)
- Step1 is repeated for the remaining portion of the cake
by the remaining players

Stepn - Since there is only one player left, he gets the last piece

**OUTCOMES** (This algorithm uses (n-1) cuts)

This is not a good algorithm for fair division of cakes because there are three disadvantages in using this algorithm to divide cakes. One, the players may take advantage of the method and say "STOP" when the left portion is more than 1/n of the cake. Second, players who said "STOP" at the same time may argue over who gets the piece. Third, it is unfair for the last player because he is assigned the last piece of the cake despite its size. The last player will be the one to get the smallest portion because, when it comes to the last two players, the cutter has to keep the knife moving until the other player say "STOP." However, the other player does not have to say stop until he wants to and this gives him the authority to say stop when the remaining portion to the right of the cake is really small. Therefore, this algorithm does not have any "check and balance" system to maintain equality among the players.

TRIMMING algorithm (n people)

Step1 - person1 cuts a piece, 1/n of the cake

Step2 - this piece is passed successively to
person2, person3,...,person(n-1)
- anyone who thinks the piece is bigger than 1/n trims
so that it is 1/n according to the trimmer.

Step3 - person(n) takes the trimmed piece
if he thinks it is at least 1/n of the cake
- otherwise the trimmed piece goes to the last trimmer

Step4 - repeast Step1 to Step3 on the remaining portion of cake
with n replaced by (n-1)
- repeat until only one player left

**OUTCOMES** (This algorithm uses indefinite cuts)

This is not a sufficient way of fairly dividing cakes into n-pieces. By trimming the cake, the players will end up with small/little pieces or even crumbs as their share of the cake. Also, this method can take a really long time to finish because the players may not agree to each of the portions until these portions become little pieces of crumbs.
>

Now that we know how to divide cakes equally among n-people. Lets explore the ways in which we could divide cakes equally among two families with unequal number of members using the Ramsey Partitions. The purpose of the Ramsey Partition is to divide a cake into k1/n and k2/n where n=k1+k2. In other words, the cake has to be divided into a ratio of k1:k2 for familyA:familyB.

RAMSEY PARTITIONS

**********

RAMSEY PARTITION algorithm

Step1 - With the ratio k1:k2 given,

choose any Ramsey partition for the ratio k1:k2

Step2 - Ask either player to cut the cake

in the ratios given in the Ramsey partition

Step3 - Ask the non-cutter to indicate which pieces

are acceptable

Step4 - Assign the non-cutter pieces chosen from

the acceptable ones whose values sum to

the non-cutter's portion and the cutter

takes the remaining pieces

**********

## Below is an EXAMPLE of a RAMSEY PARTITION:

Let familyA=k1=8 and familyB=k2=5 thus, the entire cake is 8+5=13. For the table below, the letters (A or B) to the right of a number indicates that this piece has been chosen by family A or B. Also, the number inside the { } indicates pieces that are selected from the previous steps.

Starting with a cake of 13/13 , the Ramsey Partition below requires 5-cuts/steps.

(1) FamilyB makes the first cut to divide the cake into two pieces: 8/13 and 5/13. Then familyA gets to pick.

If familyA thinks that one of the pieces is worth at least 8/13 of the cake, then familyA will pick that piece and the 5/13 goes to familyB. The division process stops here because, now, familyA has {8/13} and familyB has {5/13}.

However, if familyA does not think that any of the pieces is at least 8/13 of the cake, then familyA will take the piece that is at least 5/13. Now, familyA has {5/13} and the piece that is 8/13 is divided in step(2).

(2) FamilyA cuts the remaining piece into two pieces: 5/13 and 3/13. Then familyB gets to pick.

If familyB thinks that one of the pieces is worth at least 5/13 of the cake, then familyB will pick that piece and the 3/13 goes to familyA. The division process stops here because, now, familyA has {5/13, 3/13} and familyB has {5/13}.

However, if familyB does not think that any of the pieces is at least 5/13 of the cake, then familyB will take the piece that is at least 3/13. Now, familyA has {5/13}, familyB has {3/13} and the piece that is 5/13 is divided in step(3).

(3) FamilyB cuts the remaining piece into two pieces: 3/13 and 2/13. Then familyA gets to pick.

If familyA thinks that one of the pieces is worth at least 3/13 of the cake, then familyA will pick that piece and the 2/13 goes to familyB. The division process stops here because, now, familyA has {5/13, 3/13} and familyB has {3/13, 2/13}.

However, if familyA does not think that any of the pieces is at least 3/13 of the cake, then famil A will take the piece that is at least 2/13. Now, familyA has {5/13, 2/13}, familyB has {3/13} and the piece that is 3/13 is divided in step(4).

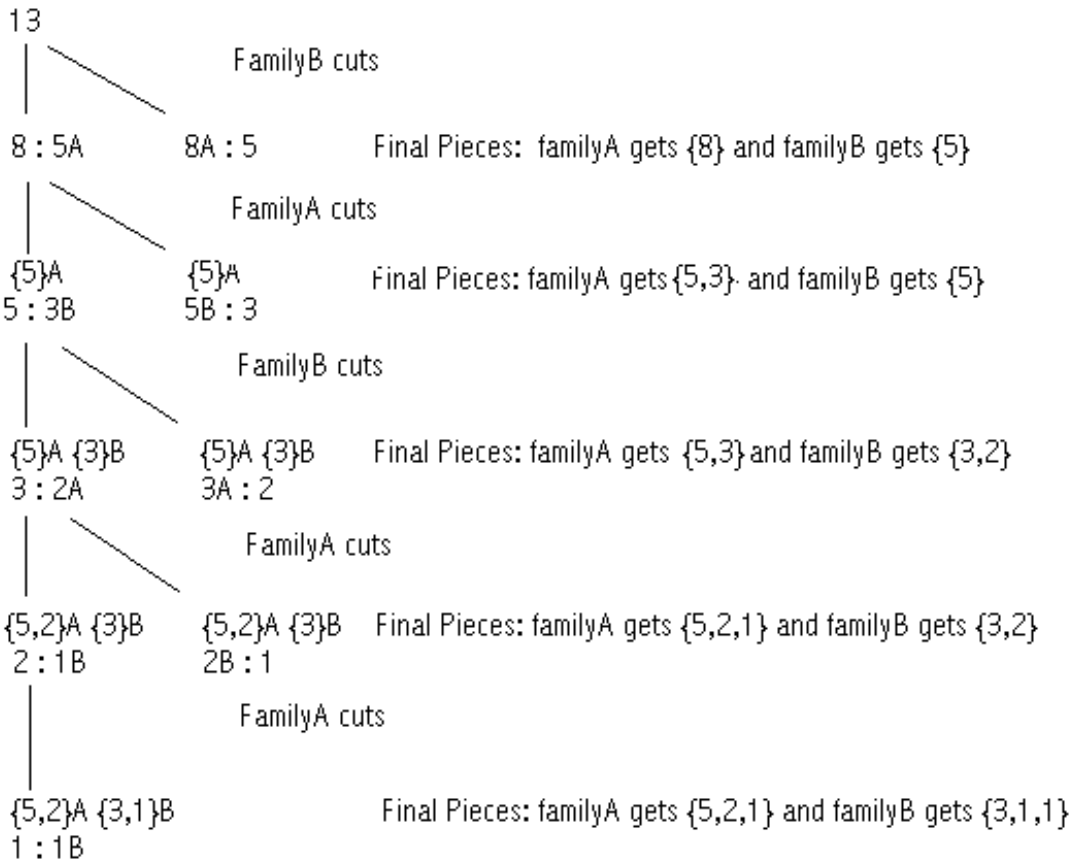
(4) FamilyA cuts the remaining piece into two pieces: 2/13 and 1/13. Then familyB gets to pick.

If familyB thinks that one of the pieces is worth at least 2/13 of the cake, then familyB will pick that piece and the 1/13 goes to familyB. The division process stops here because, now, familyA has {5/13, 2/13, 1/13} and familyB has {3/13, 2/13}.

However, if familyB does not think that any of the pieces is at least 2/13 of the cake, then familyB will take the piece that is at least 1/13. Now, familyA has {5/13, 2/13}, familyB has {3/13, 1/13} and the piece that is 2/13 is divided in step(5).

(5) FamilyB cuts the remaining piece into two pieces: 1/13 and 1/13. Then familyA gets to pick.

If familyA thinks that one of the pieces is worth at least 1/13 of the cake and the remaining piece goes to familyB. Now, familyA has {5/13, 2/13, 1/13} and familyB has {3/13,1/13,1/13}.




---

The branches that are to the right are immediate steps to the Ramsey Partitions, and they are not considered as a complete Ramsey Partition. However, the branch that goes straight down on the left is one of the many Ramsey Partitions for the ratio of k1=8:5=k2. In other words, the set of {5,3,2,1,1,1} is a Ramsey Partition, where familyA gets {5,2,1} and familyB gets {3,1,1}.

## PROPERTIES OF RAMSEY PARTITIONS

The example above is a Ramsey Partition because it satisfies 2 properties:

**(1)** A partition is Ramsey for the ratio k1:k2 if and only if the sums do not skip over either k1 or k2 when summing terms in the order they appear, leaving out whichever terms you wish.

Applying this property to the set {5,3,2,1,1,1}, it is a Ramsey Partition because 5=5 and 5+3=8 so k2=5 and k1=8 are not skipped.

**(2)** Let k1 and k2 be positive integers. A partition of the integer k1+k2 is a Ramsey Partition in the ratio k1:k2 if and only if, for any subset of terms in the partition, **there are** parts which sums to k2 in the complementary subset if **there are not** parts which sums to k1 in that subset.

Using these two properties, my goal is to produce a program that generates a table of all possible Ramsey Partitions for a given k1:k2

```
> with(combinat): with(ListTools):
Warning, the protected name Chi has been redefined and unprotected

Warning, the assigned name Group now has a global binding
```

### What is the purpose of this procedure?

The purpose of this procedure is to generate the sum of the elements in the list. Notice that Maple displays the numbers in a list from the smallest to the largest, respectively. Therefore, this procedure adds the numbers from right to left (or from last to first).

```
> sums := proc(L)

        # defining local variables
        local S, i, j;

        # initializing the set S
        S := {};

        # this for-loop goes from 1 to the number of
elements in the list L
        for i from 1 to nops(L)

          # this do-loop add the sequence of values from 1
to the number of elements in
          # the list L and combining it to the set S
          do S := S union {add(L[j],j=i..nops(L))}; end do;
```

```
        # returning the set S
        return S;

        end proc:
```

Here is a simple test to see if my procedure works. Apparently it does because it returns a list of the total starting at 4=4, 4+3=7, 4+3+2=9,...

```
> sums([1,1,2,3,4]);
```

{4, 7, 9, 10, 11}

What is the purpose of this procedure?

The purpose of this procedure is to call "sums" and store the result in a list to test whether or not the k1 and k2 are in my list of sums. If so, then return true.

```
> test := proc(k1,k2,L)

        # defining local variables
        local S, maxSum, mn, mx;

        # calling sums to add the elements in list L and
storing the result in S
        S := sums(L);

        # calling the function max to test the maximum
elements in S
        maxSum := max(seq(S[i],i=1..nops(S)));

        # returns the minimum of k1 and k2
        mn := min(k1,k2);

        # returns the maximum of k1 and k2
        mx := max(k1,k2);

        # if this test is true, then this mean that maxSum
will not skip k1 or k2
        # because it cannot skip something that it does not
reach
        if maxSum < mn then return true;

        # if this test is true, then we want to test if k1
and k2 are in S
        elif maxSum >= mx then return member(k1,S) and
member(k2,S);

        # otherwise, we just want to test if the minimum of
k1 and k2 are in S
```

```
        else member(mn,S);

        end if;

        end proc:
```

Here is a simple test to see if my procedure works. Apparently it does because it returns true when the numbers k1 and k2 are in the list of sums and false when they are not.

For this case, 4 and 7 are in the list of sums which contains {4,7,910,11}

```
> test(4,7,[1,1,2,3,4]);
```

```
                                    true
```

For this case, 5 and 6 are not in the list of sums which contains {4,7,9,10,11}

```
> test(5,6,[1,1,2,3,4]);
```

```
                                    false
```

## What does this procedure do?

The purpose of this procedure is to test for property(2) by sub-partitioning the list. This gives the sublists and their complements.

```
> SubPartitions := proc(Part)

                # defining local variables
                local SS, Subsets, i;

                # this creates a set from 1 to the number
of elements in Part
                SS := {$1..nops(Part)};

                # returns a set of all the subsets of SS
                Subsets := powerset(SS);

                # maps x to a list that contains x and a
list that contains everything else
                # this gives a list that contains a
sublist and its complements
                map(x->[convert(x,list),convert(SS minus
x, list)], Subsets);
                subs({seq(i=Part[i],i=1..nops(Part))},%);

                end proc:
```

Here is a simple test to see if my procedure works. Apparently it

does because it returns (the sublist, its complement).

```
> SubPartitions([1,1,2,3,4]);
  {[[], [1, 1, 2, 3, 4]], [[1], [1, 2, 3, 4]], [[3], [1, 1, 2, 4]],
[[1, 1, 2, 3, 4], []],

       [[1, 2, 3, 4], [1]], [[2, 3, 4], [1, 1]], [[3, 4], [1, 1, 2]],
[[1, 3, 4], [1, 2]],

       [[1, 1, 3, 4], [2]], [[4], [1, 1, 2, 3]], [[1, 4], [1, 2, 3]],
[[1, 1, 4], [2, 3]],

       [[2, 4], [1, 1, 3]], [[1, 2, 4], [1, 3]], [[1, 1, 2, 4], [3]],
[[1, 1], [2, 3, 4]],

       [[2], [1, 1, 3, 4]], [[1, 2], [1, 3, 4]], [[1, 1, 2], [3, 4]],
[[1, 3], [1, 2, 4]],

       [[1, 1, 3], [2, 4]], [[2, 3], [1, 1, 4]], [[1, 2, 3], [1, 4]],
[[1, 1, 2, 3], [4]]}
```

**What does this procedure do?**

The purpose of this procedure is to eliminate the partitions that does not satisfy the property(1) of the Ramsey Partition. First, it calculates all of the possible partitions of the given k1 and k2. Then, it calls the test function to see check for property(1).

```
> ramsey1 := proc(k1,k2)

          # defining loval variables
          local part, newPart, j;

          # returns all of the partition of k1+k2
          part := partition(k1+k2);

          # selecting the partitions from part which
satisfy the function test
          newPart := select(L->test(k1,k2,L),part);

          # returning the reversed version of newPart
          return Reverse(newPart);

          end proc:
```

Here is a simple test to see if my procedure works.

```
> R := ramsey1(3,5);
  R := [[1, 2, 2, 3], [1, 1, 1, 2, 3], [1, 1, 1, 1, 1, 3], [1, 1, 1, 1,
1, 1, 2],

       [1, 1, 1, 1, 1, 1, 1, 1]]
```

Choosing the 4th list inside list R, I use the SubPartitions to generate sublists and their complements.

```
> S := SubPartitions(R[4]);
  S := {[[], [1, 1, 1, 1, 1, 1, 2]], [[1], [1, 1, 1, 1, 1, 2]], [[2],
[1, 1, 1, 1, 1, 1]],

       [[1, 1, 1, 1, 1], [1, 2]], [[1, 1, 1, 1], [1, 1, 2]], [[1, 1,
1], [1, 1, 1, 2]],

       [[1, 1], [1, 1, 1, 1, 2]], [[1, 1, 1, 1, 1, 1, 2], []], [[1, 1,
1, 1, 1, 2], [1]],

       [[1, 1, 1, 1, 2], [1, 1]], [[1, 1, 1, 2], [1, 1, 1]], [[1, 1,
2], [1, 1, 1, 1]],

       [[1, 2], [1, 1, 1, 1, 1]], [[1, 1, 1, 1, 1, 1], [2]]}
```

**What does this procedure do?**

The purpose of this procedure is to check for property(2) by calling test to check every sublist.

```
> ramsey2 := proc(k1,k2,L)

           # defining local variables
           local subParts, i;

           # calling SubParition on L and setting it to
subParts
           subParts := SubPartitions(L);

           # this for-loop goes from 1 to the number of
elements in subParts
           for i from 1 to nops(subParts)

             do
               # if test returns false, then this means
that the sublist does not
               # satisfies property(2) of the Ramsey
Partitions so we return false
               if (test(k1,k2,subParts[i][1]) = false)
               then return false; end if;
             end do;

           # else we return true
           return true;

           end proc:
```

Here is a simple test to see if my procedure works.

```
> ramsey2(3,5,R[4]);
                                                  true
```

Knowing that the three procedures above works, it is time to put

it all together.

**What does this procedure do?**

The purpose of this procedure is to display the all of the partitions that satisfy the property(1) and property(2) of the Ramsey Partitions.

```
> finallyDONE := proc(k1,k2)

                # defining local variables
                local R, L, i;

                # initializing the list L
                L := [];

                # calling the ramsey1 on k1 and k2 to test
for property(1)
                R := ramsey1(k1,k2);

                # this for-loop goes from 1 to the number of
elements in R
                for i from 1 to nops(R)

                  do
                    # if ramsey2 returns true for k1 and k2,
                    # then reverse the elements in R[i] and
place in L
                    if ramsey2(k1,k2,R[i])
                    then L := [op(L), Reverse(R[i])]; end
if;
                  end do;

                # return the list L
                return L;

                end proc:
```

Calling the procedure above for k1=8:5=k2, the following Ramsey Partitions are displayed.

```
> f := finallyDONE(8,5);
  f := [[5, 3, 2, 1, 1, 1], [5, 3, 1, 1, 1, 1, 1], [5, 2, 1, 1, 1, 1,
1, 1],

        [5, 1, 1, 1, 1, 1, 1, 1, 1], [4, 1, 1, 1, 1, 1, 1, 1, 1, 1],
[3, 2, 2, 1, 1, 1, 1, 1, 1],

        [3, 2, 1, 1, 1, 1, 1, 1, 1, 1], [3, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1],

        [2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1], [2, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
1, 1, 1],

       [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

However, this is not what I want. So I use a for-loop to create a table that displays the partitions on seperate lines and YOILA, this is what I get.

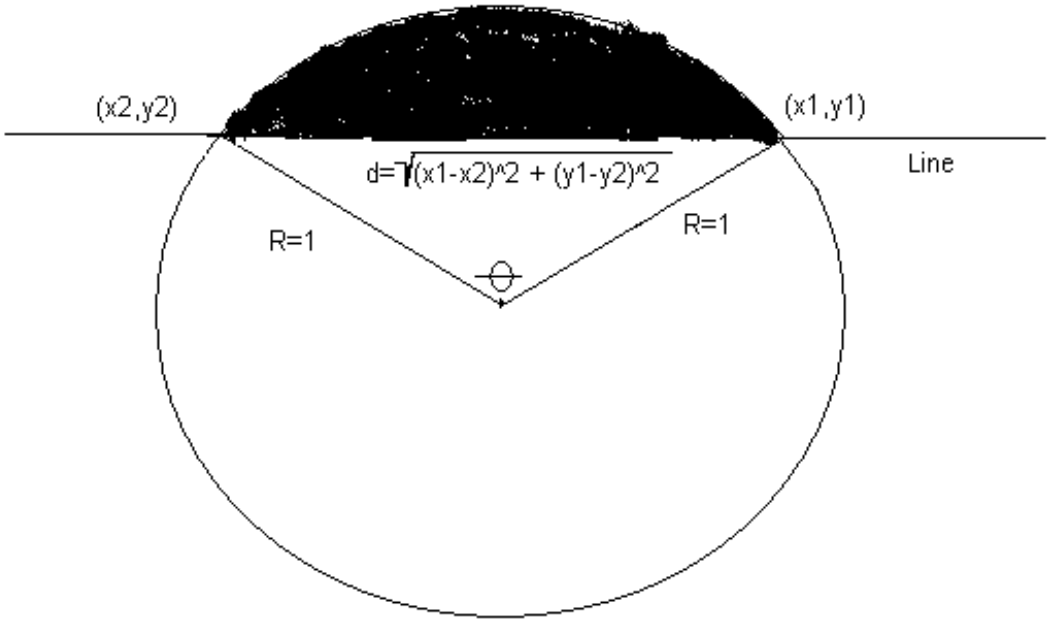
```
> for i from 1 to nops(f)
   do printf("%d.   %a\n", i, f[i]) end do;
1.   [5, 3, 2, 1, 1, 1]
2.   [5, 3, 1, 1, 1, 1, 1]
3.   [5, 2, 1, 1, 1, 1, 1, 1]
4.   [5, 1, 1, 1, 1, 1, 1, 1, 1]
5.   [4, 1, 1, 1, 1, 1, 1, 1, 1, 1]
6.   [3, 2, 2, 1, 1, 1, 1, 1, 1]
7.   [3, 2, 1, 1, 1, 1, 1, 1, 1, 1]
8.   [3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
9.   [2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1]
10.   [2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
11.   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
>
```

One may ask for the reason why we bother to write a program that generates all of the Ramsey Partitions when it is a lot easier to use the partition of [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ... (n-times)] to divide the cakes into n-pieces. The answer is that the Ramsey Partitions provide all of the possible ways to divide the cake. So that individuals who are concern with the least number of cuts that are required to divide the cake can look at the different partitions and compare. Notice that in the example above, the first partition of [5,3,2,1,1,1] only requires 5-cuts whereas the last partition of [1,1,1,1,1,1,1,1,1,1,1,1,1] requires 12-cuts.

Now, that we are familiar with some of the algorithms to fairly divide cakes, I want to write a program to calculate the area of the piece.

## EXPLORATION SECTION

Assume that the cake is a unit circle that is being divide by a line L. For example, in the diagram below, we want to calculate the area of the shaded region.




To do this we need:
(areaOfShadedRegion) + (areaOfTriangle) = (theta/2Pi) ( Pi*R^2), which implies that

areaOfShadedRedion = (theta/2Pi) ( Pi*R^2) - (areaOfTriangle), note that R=1 for unit circle

areaOfShadedRedion = (theta/2Pi) ( Pi*1) - (areaOfTriangle), notice that Pi cancels out

areaOfShadedRedion = (theta/2) - (areaOfTriangle)

and so:
(1st)
We need to find theta, which is (vector1 . vector2) / (||vector1|| . ||vector2||) = cos(theta)
Note that vector1 and vector2 are taken from the coordinates between the line and the circle.
(2nd)
Now, calculate areaOfTriangle = (1/2)*d*R*sin(theta)
(3rd)
After we know theta and the areaOfTriangle, we can substitute the solution back into the equation in red to solve for the areaOfShadedRegion.

### What is the purpose of this procedure?

The purpose of this procedure is to find the coordinates of intersection betwee nthe lines and the unit circle.

```
> findingXY := proc(line)

              # defining local variables
              local myCircle, answer, x1, x2, y1, y2;

              # defining a unit circle
              myCircle := x^2 + y^2 = 1;

              # solving a linear system for values of x and
y
              answer := solve({myCircle,line},{x,y});

              #x1 := answer[1][1];
              #y1 := answer[1][2];
              #x2 := answer[2][1];
              #y2 := answer[2][2];

              end proc:
```

In this example, the procedure returns {y,x} , {y,x}

```
> findingXY(y=x/3-1);
                          {y = -1, x = 0}, {y = -4/5, x = 3/5}
```

In this example, the procedure returns {x,y} , {y,x}

```
> findingXY(y=2*x/3+1);
                                 -12
                          {x = ---, y = 5/13}, {x = 0, y = 1}
                                 13
>
>
>
```

Since the position of x and y changes depending on the equation of the line, I could not figure out how to obtain the values of x and y from the procedure in order to calculate for d = [(x1-x2)^2+(y1-y2)^2]^(1/2). So I do not know how to proceed.

## CLOSING THOUGHTS

I started this project with the goal of exploring ways in which I can divide cakes and other desserts into pieces with equal area using only vertical cuts. However, the more I research the topic of cake-cutting, the more my project changes. My findings consist more of experiments rather than the actual programming

so it is difficult to place together a final project with programs. There are so many facts to consider. Thanks to the help provided by Stefan in explaining the algorithm of Ramsey Partitions, I have decided to concentrate my final project on writing a program for this algorithm which will display a table of all the Ramsey Partitions. However, the program for Ramsey Partitions that I have written works for small k1 and k2 and will become inefficient for large k1 and k2.