6G6Z1109: Software Agents and Optimisation

Term 2, Lecture 7:

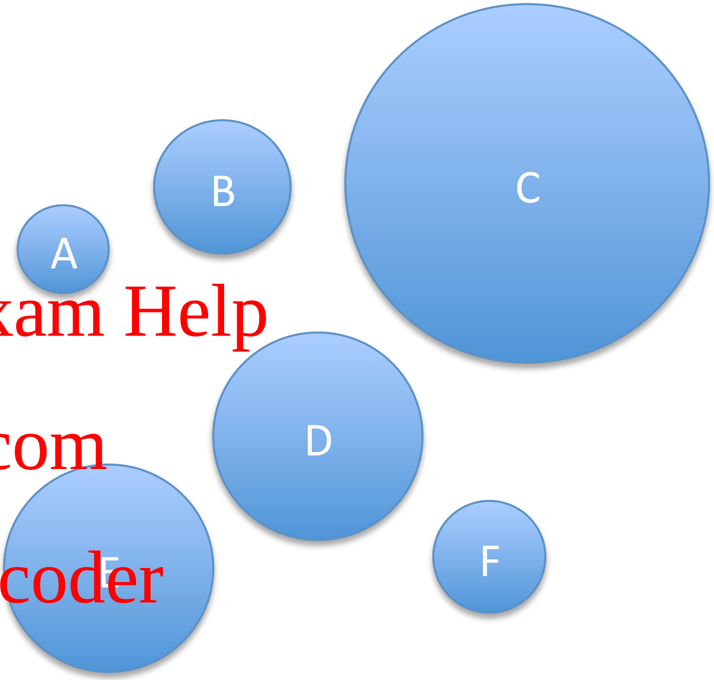
Local Search

Local search

- For many problems, we can improve the quality of genetic search by somehow *embedding* knowledge of the problem into the algorithm
- Benefits:
  - – Reduces the probability of “illegal” solutions
  - – Simplifies the representation scheme
  - – Speeds up search/reduces size of solution space

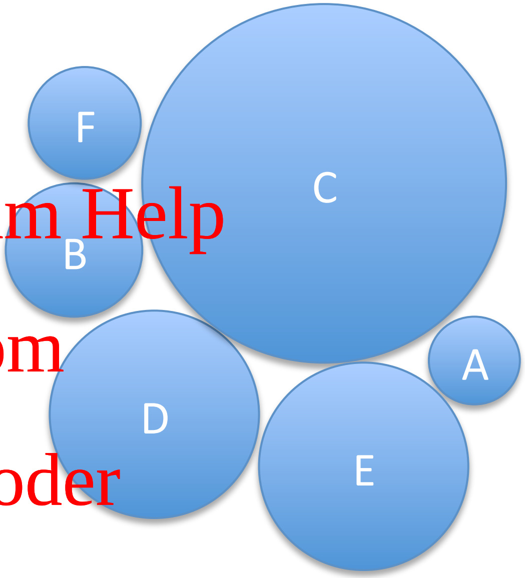
An example

- Imagine we wish to pack a collection of *non-overlapping* circles with specified radii into the *smallest possible containing radius* (i.e., the assignment)
- How might we *represent* a solution to the problem in a GA?



One idea

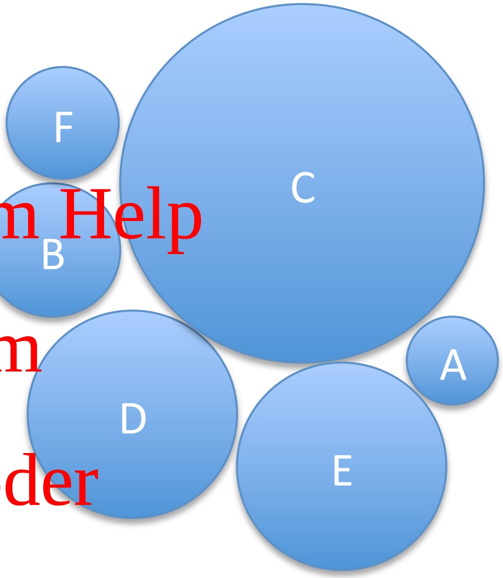
- Represent a solution as a set of x,y *coordinate pairs*, one for each circle's centre point



A	B	C	D	E	F
x,y	x,y	x,y	x,y	x,y	x,y

One idea

- Represent a solution as a set of x,y *coordinate pairs*, one for each circle's centre point

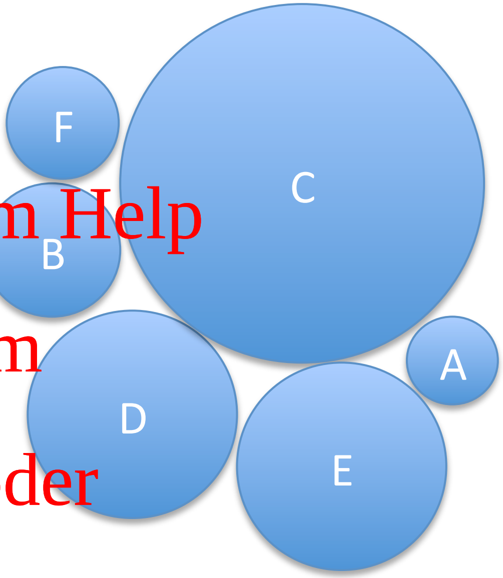


A	B	C	D	E	F
x,y	x,y	x,y	x,y	x,y	x,y

Problems with this approach?

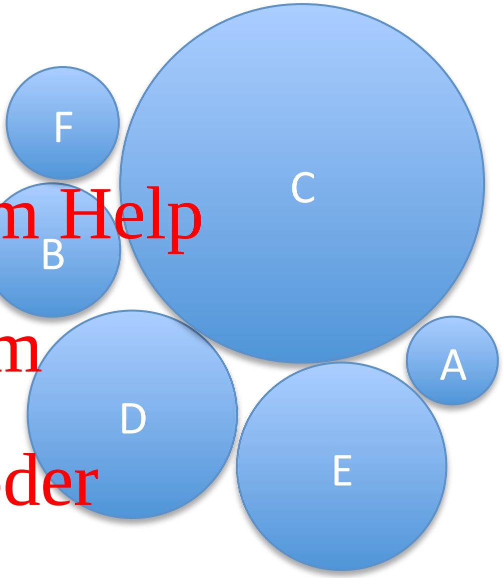
First problem

- For a (e.g.) 30x30 discrete space, there are 30x30=90 possible locations for each circle
- 90x90x90x90 = 90^n possibilities of placement for n circles
- n=10, 90^n=34,867,844,010,000,000,000
- Problem 1: *size of search space* for even modest probem instances



First problem

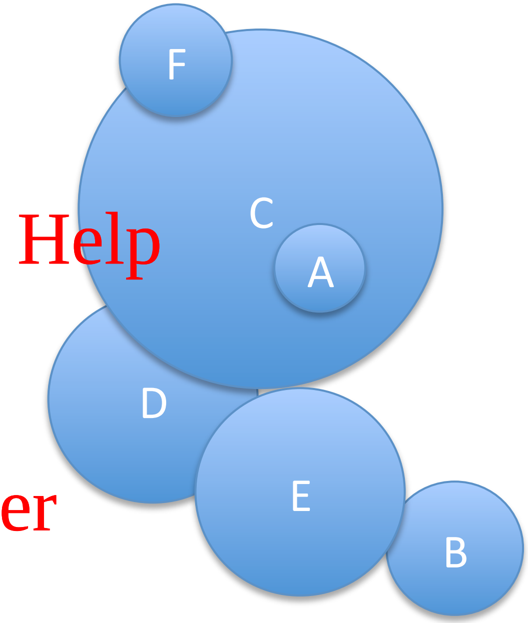
- For a (e.g.) 30x30 discrete space, there are 30x30=90 possible locations for each circle
- 90x90x90x90 = 90^n possibilities of placement for n circles
- n=10, 90^n=34,867,844,010,000,000,000
- Problem 1: *size of search space* for even modest probem instances



Next problem?

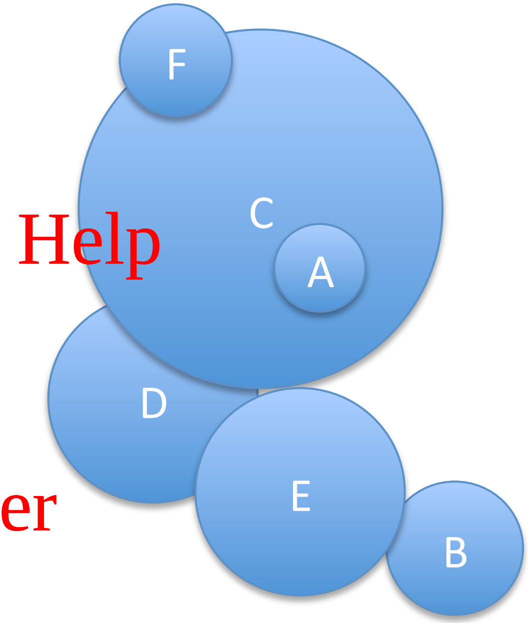
Second problem

- Nothing in this encoding prevents circles from *overlapping*
- *Illegal* solutions are actually *much* (MUCH) more likely than legal ones, as legal solutions require *all* circles to be disjoint
- Problem 2: *illegal* solutions are the norm



Second problem

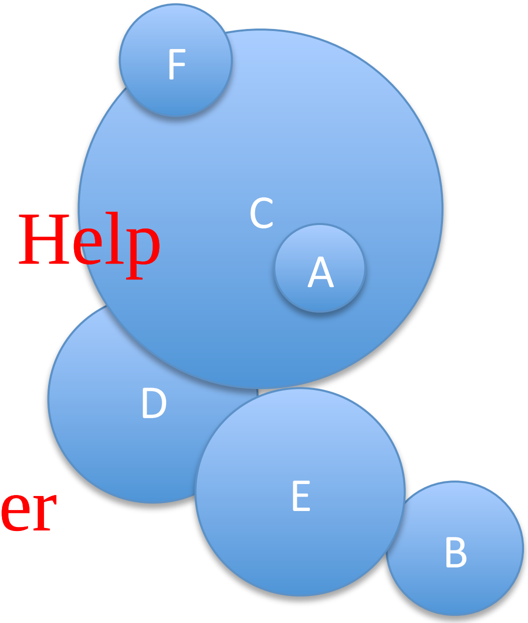
- Nothing in this encoding prevents circles from overlapping
- *Illegal* solutions are actually *much* (MUCH) more likely than legal ones, as legal solutions require *all* circles to be disjoint
- Problem 2: *illegal* solutions are the norm



Which type of encoding would address both of these problems?

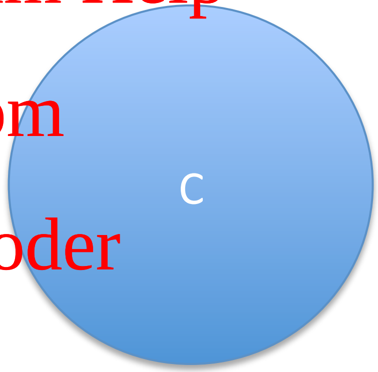
Order-based encoding

- Recall that an *order-based* encoding specifies a sequence of "things"/moves/etc
- We can use this type of encoding to specify the *order in which circles are placed*



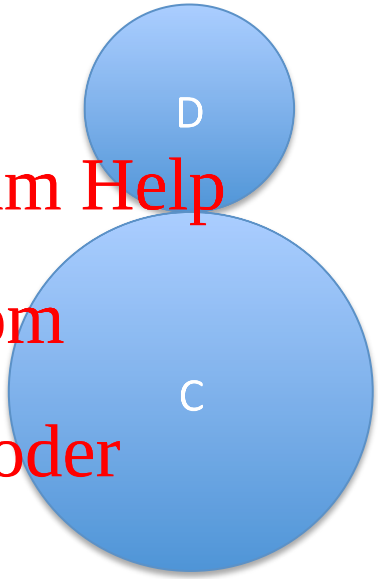
Order-based encoding: example

- If we assume that the first circle is placed in the centre of the space, then an ordering CDAEFB is shown to the right



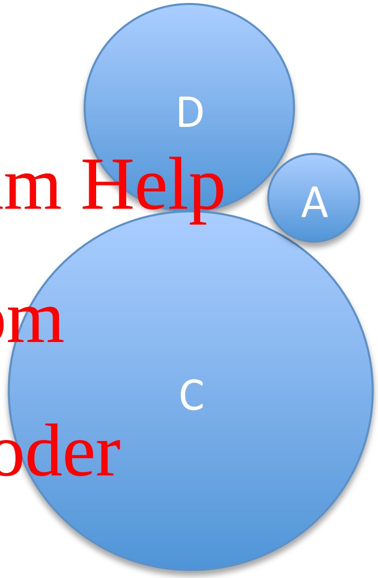
Order-based encoding: example

- If we assume that the first circle is placed in the centre of the space, then an ordering CDAEFB is shown to the right



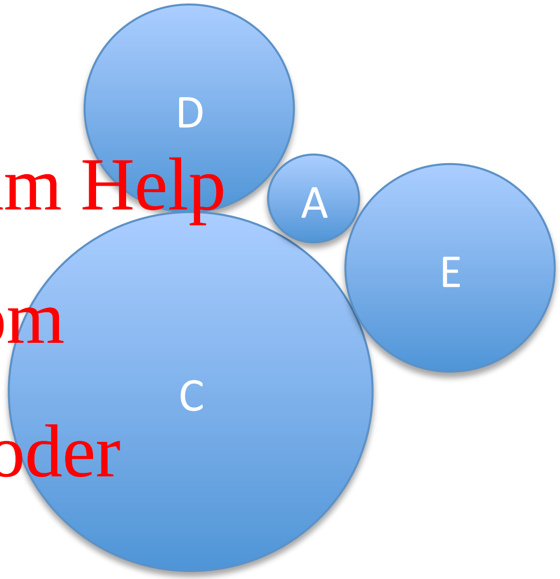
Order-based encoding: example

- If we assume that the first circle is placed in the centre of the space, then an ordering CDAEFB is shown to the right



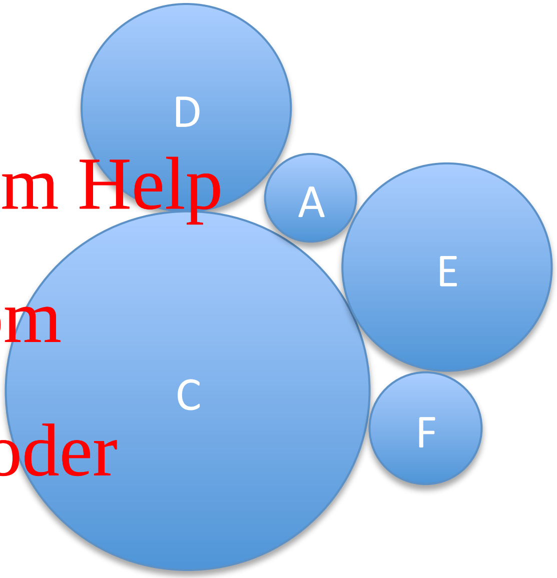
Order-based encoding: example

- If we assume that the first circle is placed in the centre of the space, then an ordering CDAEFB is shown to the right



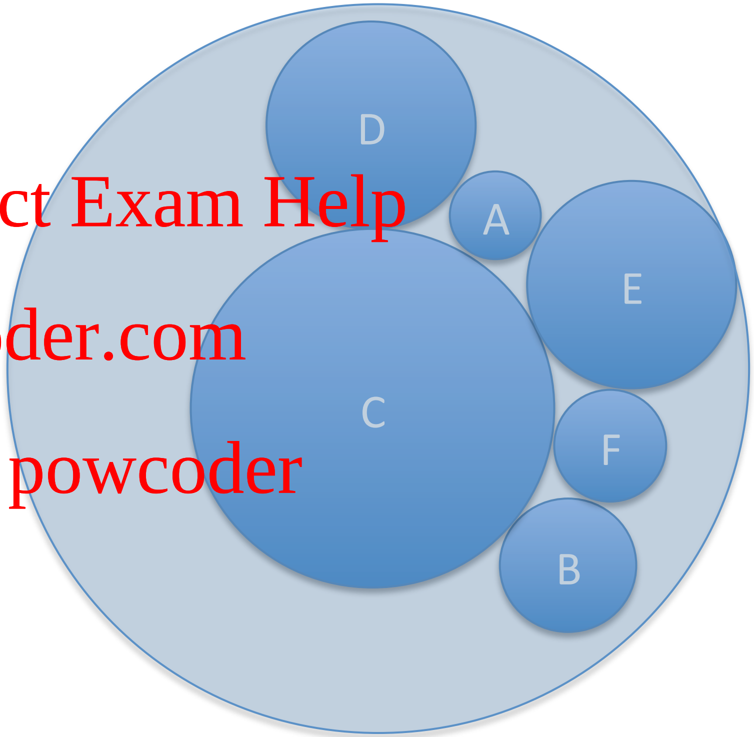
Order-based encoding: example

- If we assume that the first circle is placed in the centre of the space, then an ordering CDAEFB is shown to the right



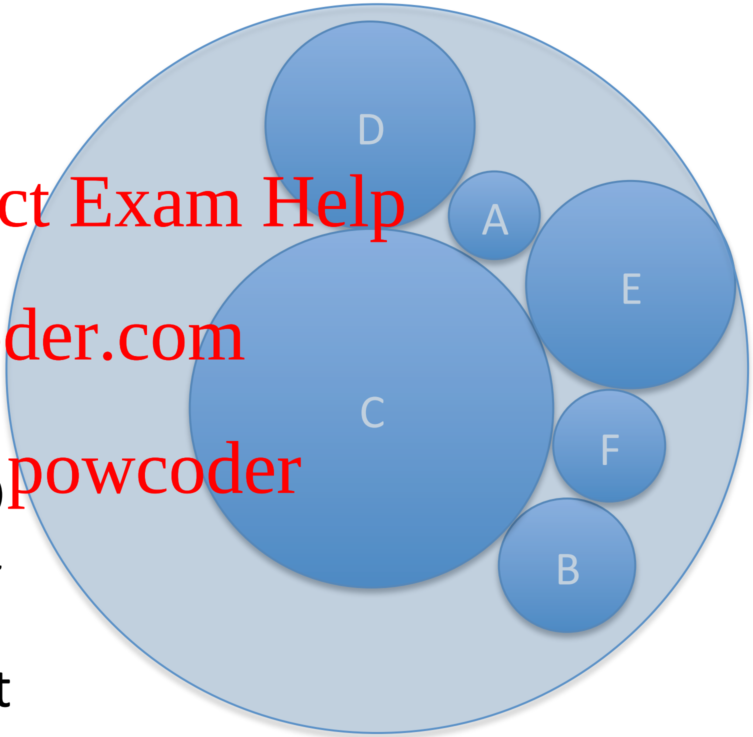
Order-based encoding: example

- If we assume that the first circle is placed in the centre of the space, then an ordering CDAEFB is shown to the right



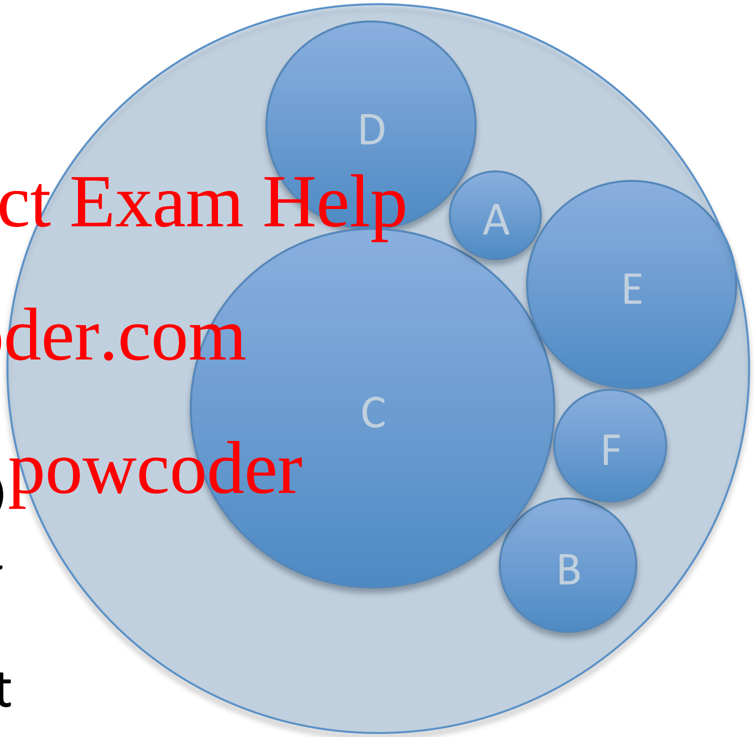
Order-based encoding: benefits

- If we get the placement algorithm right, then this encoding *automatically* prevents illegal solutions
- Number of possible solutions/ permutations of n circles = n!
- 10!=3,628,800
- *Significantly* less than 90^{10} $(90^{10}$=34,867,844,010,000,000,000$)$
- Placement ordering is *independent* of size of 2D space in which circles are placed (unlike direct placement encoding)



Order-based encoding: benefits

- If we get the placement algorithm right, then this encoding *automatically* prevents illegal solutions
- Number of possible solutions/ permutations of n circles = n!
- 10!=3,628,800
- *Significantly* less than 90^{10} $(90^{10}$=34,867,844,010,000,000,000$)$
- Placement ordering is *independent* of size of 2D space in which circles are placed (unlike direct placement encoding)

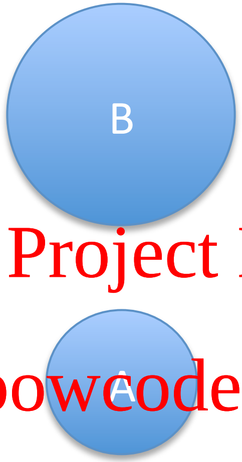


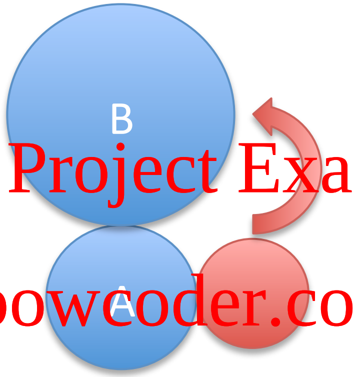
This is our "local search". For each circle in turn, we "search locally" for the best place in which to put it…

Circle placement

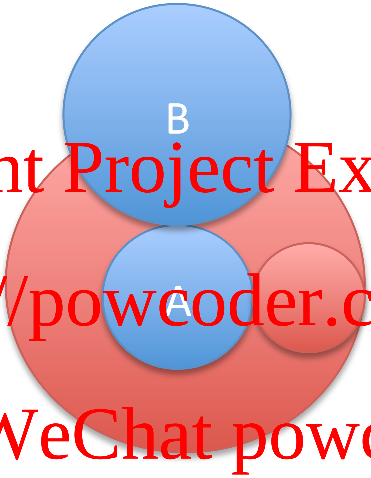
- Assume a set of circles, with specified radii
- We place the first circle in the centre of the space
- Where do we place the *next* circle?

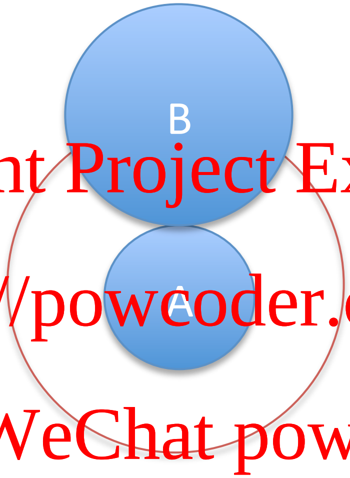
A


B
A



The summed radii of the circles defines a “shell” of valid placement points for B….


B
A

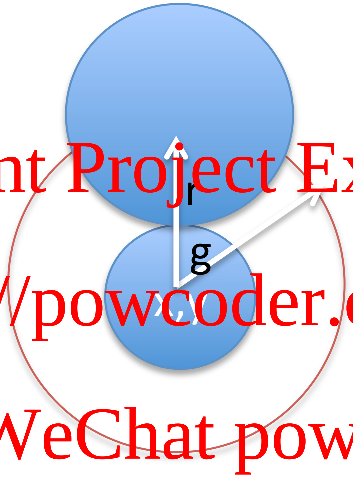

B
A

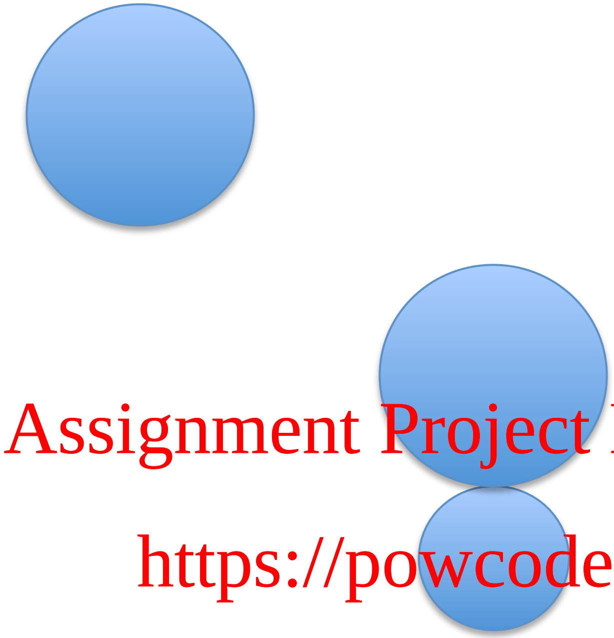
Given a point x,y, a radius r, and an angle, g, we can find the location of the point on the circle using:

px=x + cos(a) * r
py=y + sin(a) * r

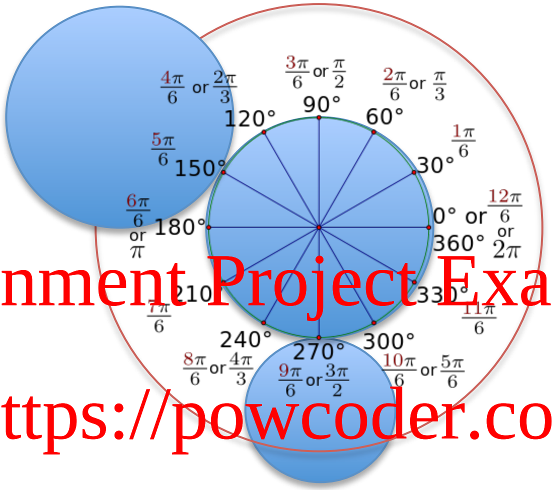
NB: all angles must be in *radians*

NB: r must be equal to the radius of a *plus* the radius of b

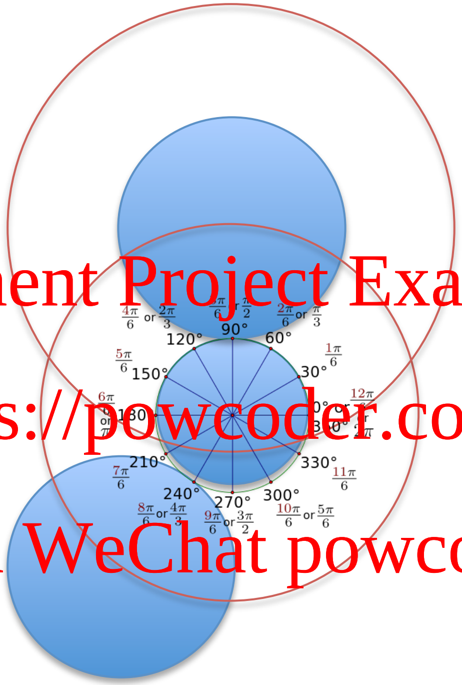




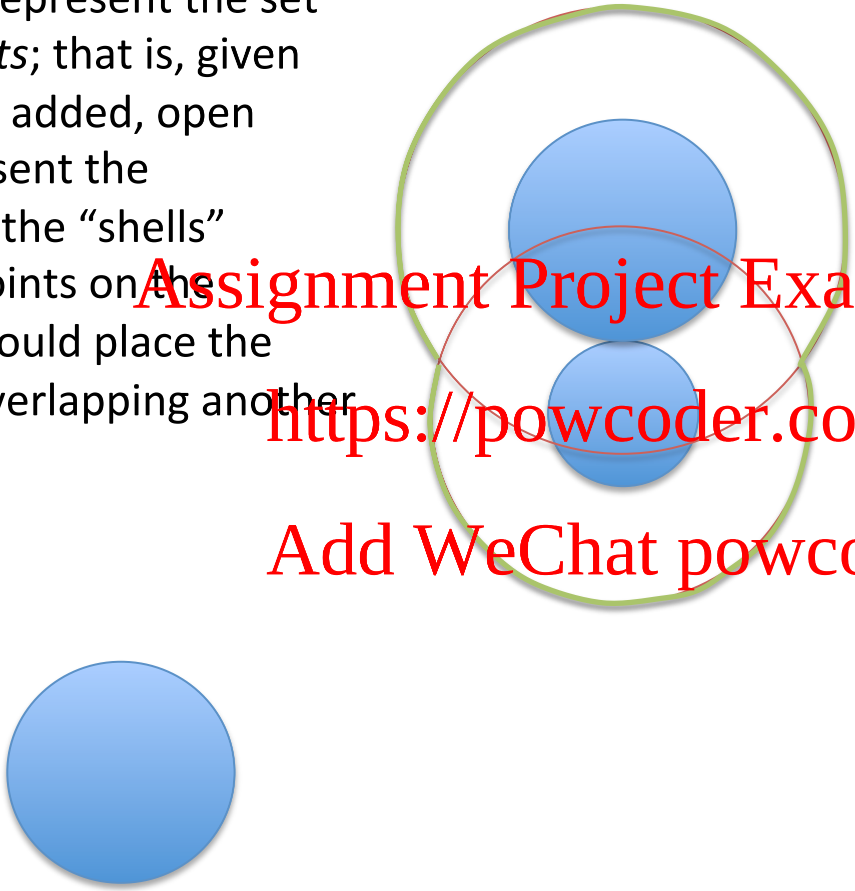
When adding a *new* circle, we start by calculating the set of “shells” that must exist for each *existing* circle…



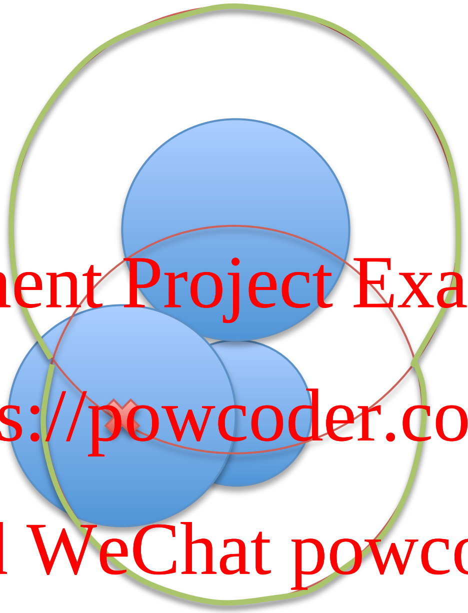
$\frac{3\pi}{6}$ or $\frac{\pi}{2}$
90°
$\frac{2\pi}{6}$ or $\frac{\pi}{3}$
60°
$\frac{4\pi}{6}$ or $\frac{2\pi}{3}$
120°
$\frac{1\pi}{6}$
30°
$\frac{5\pi}{6}$
150°
$\frac{6\pi}{6}$ or π
180°
0° or 360°
$\frac{12\pi}{6}$ or 2π
210°
$\frac{7\pi}{6}$
330°
$\frac{11\pi}{6}$
240°
$\frac{8\pi}{6}$ or $\frac{4\pi}{3}$
270°
$\frac{9\pi}{6}$ or $\frac{3\pi}{2}$
300°
$\frac{10\pi}{6}$ or $\frac{5\pi}{6}$


$\frac{4\pi}{6}$ or $\frac{2\pi}{3}$
120°
$\frac{3\pi}{6}$ or $\frac{\pi}{2}$
90°
$\frac{2\pi}{6}$ or $\frac{\pi}{3}$
60°
$\frac{1\pi}{6}$
30°
$\frac{5\pi}{6}$
150°
$\frac{6\pi}{6}$ or π
180°
$\frac{12\pi}{6}$ or 2π
0° or 360°
210°
$\frac{7\pi}{6}$
330°
$\frac{11\pi}{6}$
240°
$\frac{8\pi}{6}$ or $\frac{4\pi}{3}$
270°
$\frac{9\pi}{6}$ or $\frac{3\pi}{2}$
300°
$\frac{10\pi}{6}$ or $\frac{5\pi}{6}$

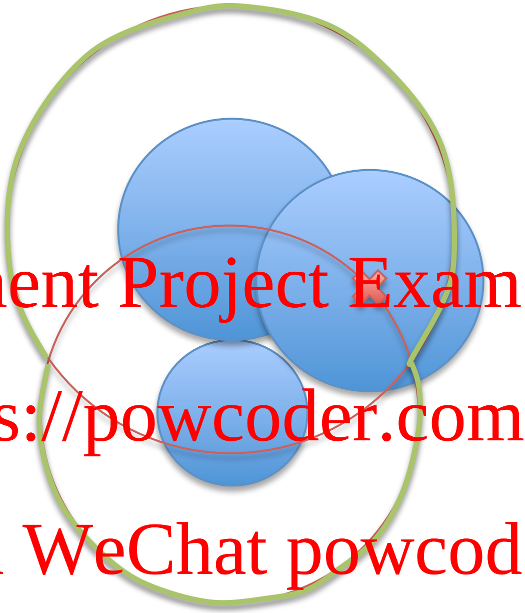
Green lines represent the set of *open points*; that is, given a circle to be added, open points represent the sum of all of the “shells” minus the points on the shells that would place the new circle overlapping another circle



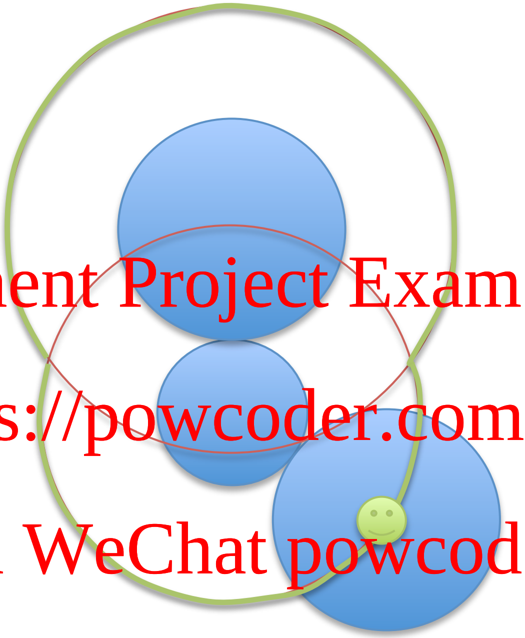
Green lines represent the set of *open points*; that is, given a circle to be added, open points represent the sum of all of the "shells" minus the points on the shells that would place the new circle overlapping another circle



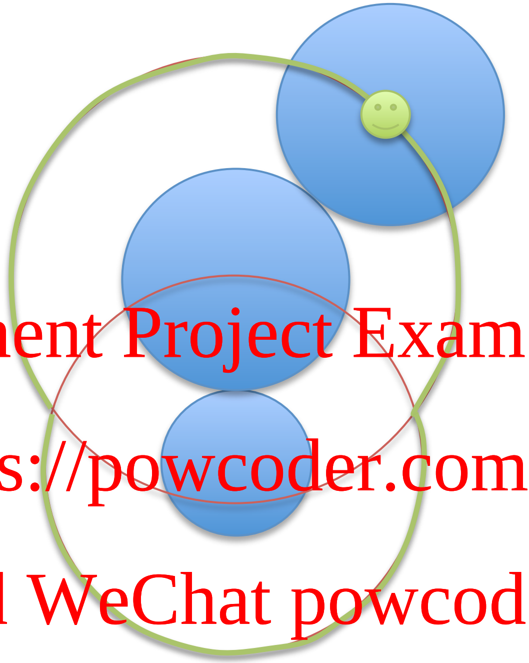
Green lines represent the set of *open points*; that is, given a circle to be added, open points represent the sum of all of the "shells" minus the points on the shells that would place the new circle overlapping another circle



Green lines represent the set of *open points*; that is, given a circle to be added, open points represent the sum of all of the “shells” minus the points on the shells that would place the new circle overlapping another circle

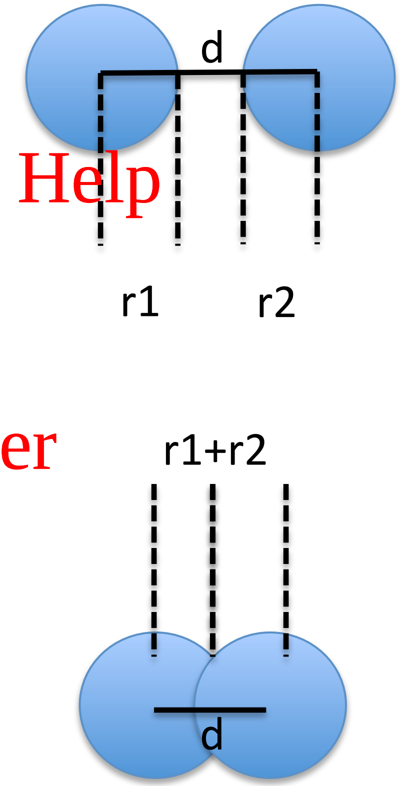


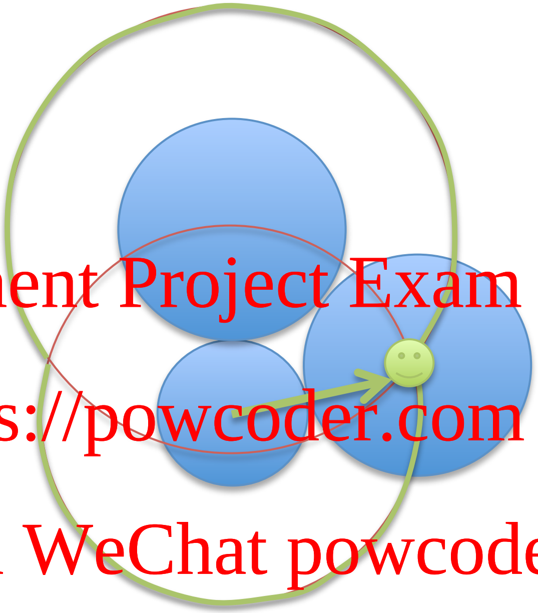
Green lines represent the set of *open points*; that is, given a circle to be added, open points represent the sum of all of the “shells” minus the points on the shells that would place the new circle overlapping another circle



How do we check to see if two circles overlap?

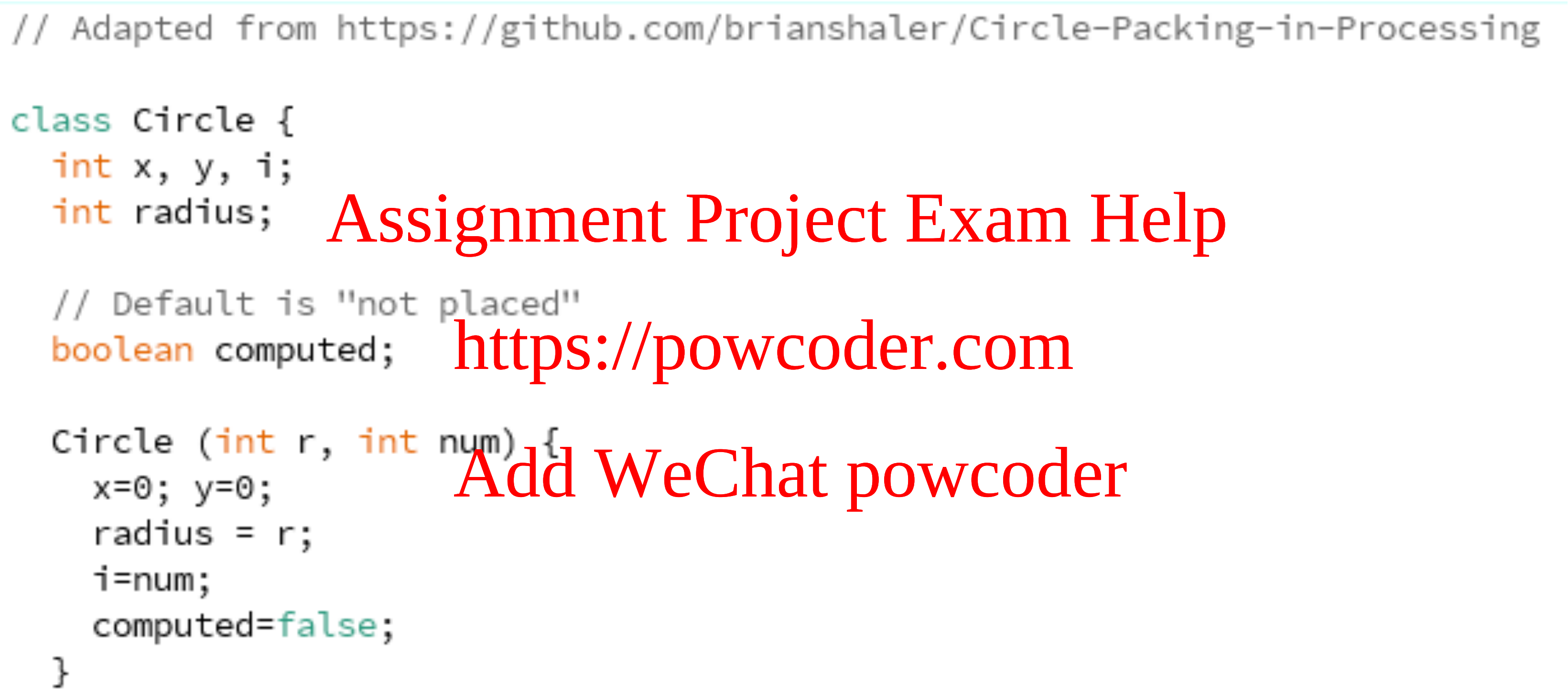
If the distance, d, between their centre points is between the sum and the difference of their radii, then they overlap





To place the circle, we simply find the open point closest to the *centre* of the space…

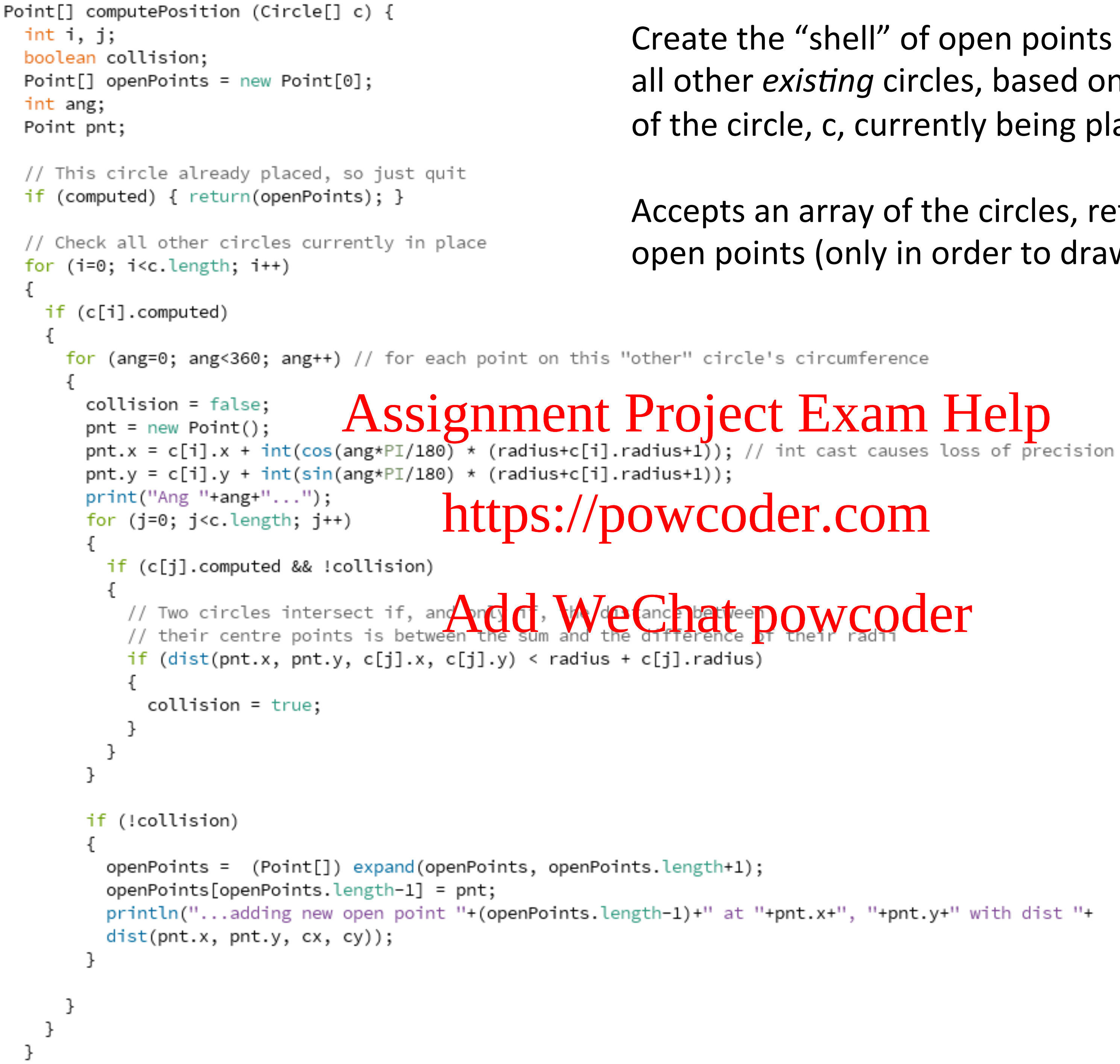
Code!



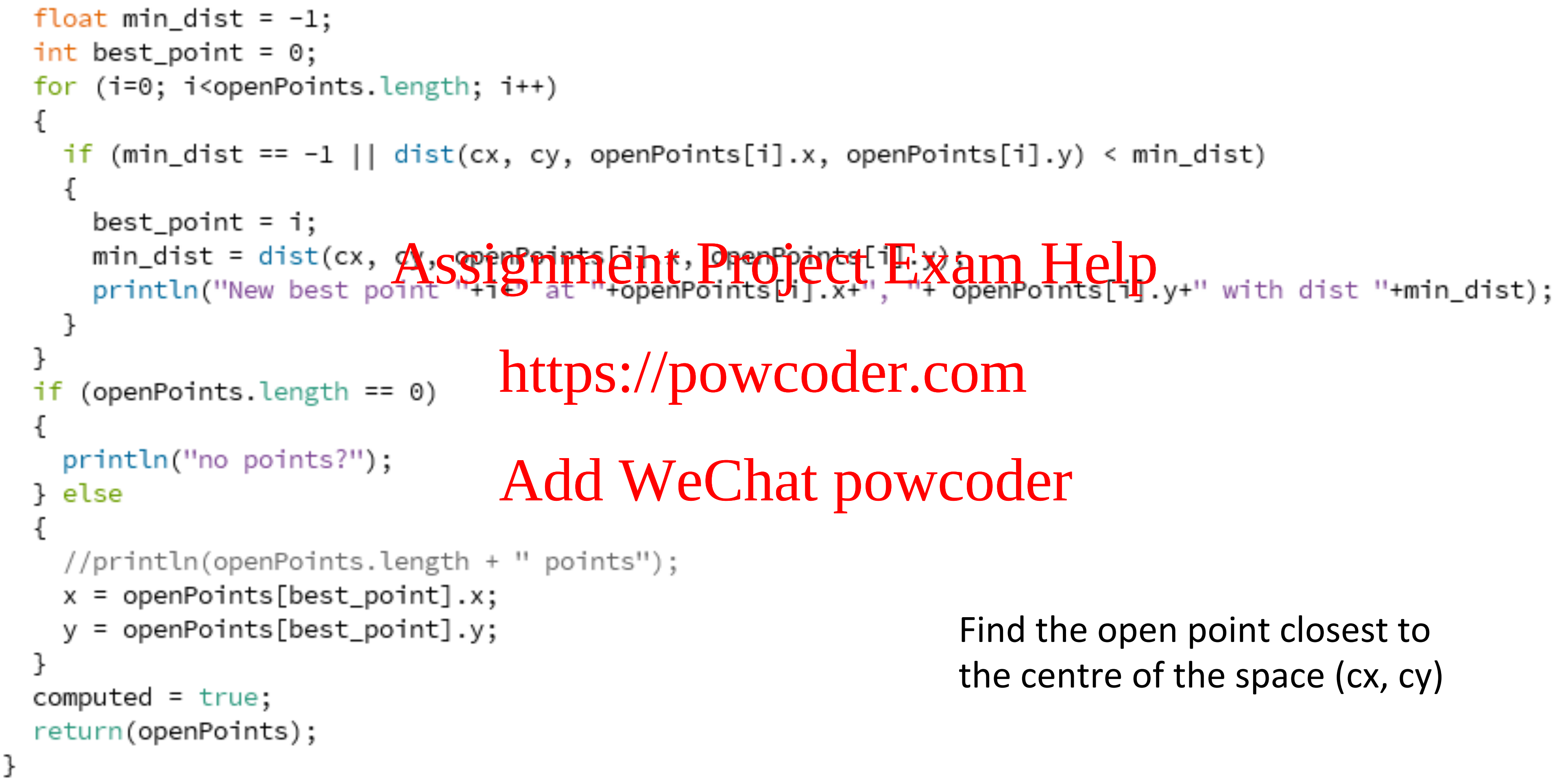
```
// Adapted from https://github.com/brianshaler/Circle-Packing-in-Processing

class Circle {
  int x, y, i;
  int radius;

  // Default is "not placed"
  boolean computed;

  Circle (int r, int num) {
    x=0; y=0;
    radius = r;
    i=num;
    computed=false;
  }
```

Create the “shell” of open points around all other *existing* circles, based on the radius of the circle, c, currently being placed

Accepts an array of the circles, returns the list of open points (only in order to draw them)



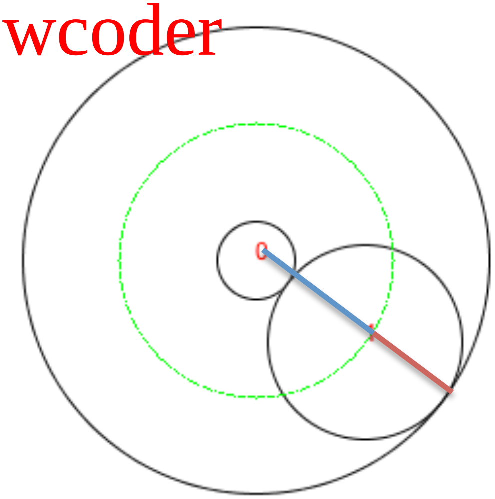
```
Point[] computePosition (Circle[] c) {
  int i, j;
  boolean collision;
  Point[] openPoints = new Point[0];
  int ang;
  Point pnt;

  // This circle already placed, so just quit
  if (computed) { return(openPoints); }

  // Check all other circles currently in place
  for (i=0; i<c.length; i++)
  {
    if (c[i].computed)
    {
      for (ang=0; ang<360; ang++) // for each point on this "other" circle's circumference
      {
        collision = false;
        pnt = new Point();
        pnt.x = c[i].x + int(cos(ang*PI/180) * (radius+c[i].radius+1)); // int cast causes loss of precision
        pnt.y = c[i].y + int(sin(ang*PI/180) * (radius+c[i].radius+1));
        print("Ang "+ang+"...");
        for (j=0; j<c.length; j++)
        {
          if (c[j].computed && !collision)
          {
            // Two circles intersect if, and only if, the distance between
            // their centre points is between the sum and the difference of their radii
            if (dist(pnt.x, pnt.y, c[j].x, c[j].y) < radius + c[j].radius)
            {
              collision = true;
            }
          }
        }

        if (!collision)
        {
          openPoints =  (Point[]) expand(openPoints, openPoints.length+1);
          openPoints[openPoints.length-1] = pnt;
          println("...adding new open point "+(openPoints.length-1)+" at "+pnt.x+", "+pnt.y+" with dist "+
          dist(pnt.x, pnt.y, cx, cy));
        }

      }
    }
  }
```



```
  float min_dist = -1;
  int best_point = 0;
  for (i=0; i<openPoints.length; i++)
  {
    if (min_dist == -1 || dist(cx, cy, openPoints[i].x, openPoints[i].y) < min_dist)
    {
      best_point = i;
      min_dist = dist(cx, cy, openPoints[i].x, openPoints[i].y);
      println("New best point "+i+" at "+openPoints[i].x+", "+ openPoints[i].y+" with dist "+min_dist);
    }
  }
  if (openPoints.length == 0)
  {
    println("no points?");
  } else
  {
    //println(openPoints.length + " points");
    x = openPoints[best_point].x;
    y = openPoints[best_point].y;
  }
  computed = true;
  return(openPoints);
}
```

Find the open point closest to the centre of the space (cx, cy)

```
float computeBoundary ()
{
  // Find bounding circle for circles
  int i;
  float outer_limit=0;
  int furthest=0;
  float distance=0;

  for (i=0; i<numCircles; i++)
  {
    if (Circles[i].computed)
    {
      distance=dist(cx, cy, Circles[i].x, Circles[i].y)+Circles[i].radius;
      if (distance >= outer_limit)
      {
        outer_limit=distance;
        furthest=i;
      }

    }
  }
  return(outer_limit);
}
```



Next lecture

- Next week: Comparative analysis of algorithms, more help with the assignment
- This week's lab: Start to implement assignment solution!