Part II

Puzzle solving

(Redundant) representation of a configuration

- System comprises of:
  - 18 bead wheels
  - 11 blue beads
  - 12 yellow beads
  - 11 red beads

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
w1	anti-clock	y	b	b	b	r	r	r	r	r	r	r	b	r	b	b	r	b	y
w2	clockwise	y	b	y	y	y	y	y	y	y	y	y	y	r	b	b	r	r	b

Shifting w1 clockwise by 1

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
w1	anti-clock	b	b	b	r	r	r	r	r	r	r	b	r	b	b	r	b	y	y
w2	clockwise	b	b	y	y	y	y	y	y	y	y	y	y	b	r	b	r	r	b

Final blue/yellow/red configuration

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
w1	anti-clock	y	b	b	b	b	b	b	b	b	b	b	b	y	y	y	y	y	y
w2	clockwise	y	r	r	r	r	r	r	r	r	r	r	r	y	y	y	y	y	y

Generic configurations

- Original and specific puzzles:
  - 18 11/7/4 bead wheels
  - 12 6/4/2 yellow beads
  - 11 7/4/2 blue and red beads
- Generic puzzle:
  - w bead wheels
  - y yellow beads (y even because of vertical symmetry)
  - b = r = w - (y/2 + 1) blue and red beads

	0											r+1					w-1
w1	y	b	b	b	b	b	b	b	b	b	b	y	y	y	y	y	y
w2	y	r	r	r	r	r	r	r	r	r	r	y	y	y	y	y	y

nextConfigs method (1 of 2)

```
private LinkedList<String> nextConfigs(String config)
{ // 1
   String[] wheels = new String[2];
   wheels[0] = config.substring(0, wheel);
   wheels[1] = config.substring(wheel);
   LinkedList<String> result = new LinkedList<String>();
   for (int i = 0; i < 2; i++)
   {  // 2
         for (int j = 1; j < wheel; j++)
         { // 3
            char[] shiftedWheel = new char[wheel];
            for (int k = 0; k < wheel; k++)
                  shiftedWheel[k] = wheels[i].charAt((k+j) % wheel);
            String nextConfig;
```

nextConfigs method (2 of 2)

```
                String nextConfig;
                if (i == 0)
                    nextConfig = String.copyValueOf(shiftedWheel) +
                            shiftedWheel[0] +
                            wheels[1].substring(1, blueRed + 1) +
                            shiftedWheel[blueRed + 1] +
                            wheels[1].substring(blueRed + 2);
                else
                    nextConfig = shiftedWheel[0] +
                            wheels[0].substring(1, blueRed + 1) +
                            shiftedWheel[blueRed + 1] +
                            wheels[0].substring(blueRed + 2) +
                            String.copyValueOf(shiftedWheel);
                if (!config.equals(nextConfig) && !result.contains(nextConfig))
                    result.add(nextConfig);
            } // 3
        } // 2
    return result;
} // 1
```

On the disequality and inclusion test

- Consider next configurations of rbrb ryyb:

left shift	Wheel 1	both	Wheel 2	both
1	brbr	brbr byyr	yybr	ybrr yybr
2	rbrb	rbrb ryyb	ybry	ybry ybry
3	brbr	brbr byyr	bryy	bbry bryy

- Thus set of only 5 configurations
- 2(w-1) in worse-case as shown by rbbr ryyr:
  - bbrrbyyr, brrbbyyb, rrbbryyb, ybbryyrr, ybbyyrry, rbbyrryy

Depth-first (limited)

```
private LinkedList<String> depthFirst(String start, String dest, int depth)
{ // 1
    if (depth == 0) return null;
    else if (start.equals(dest))
    {
        LinkedList<String> route = new LinkedList<String>();
        route.add(start);
        return route;
    }
    else
```

Depth-first (limited) cont'

```
    else
    {
        LinkedList<String> result = nextConfigs(start);
        for (String nextConfig:result)
        {
            LinkedList<String>
                route = depthFirst(nextConfig, dest, depth - 1);
            if (route != null)
            {
                route.addFirst(start);
                return route;
            }
        }
        return null;
    }
} // 1
```

Iterative Deepening

```
public LinkedList<String> iterativeDeepening(String start, String dest)
{
    for (int depth = 1; true; depth++)
    {
        System.out.println(depth);
        LinkedList<String> route = depthFirst(start, dest, depth);
        if (route != null) return route;
    }
}
```

Sample runs

```
>java BeadFinder yrryybby ybbyyrry
1
2
3
```


```
4
[yrryybby, ryyrrbbr, byybbrrb, ybbyyrry]
>
>java BeadFinder ybbyyrry yrryybby
1
2
3
```


```
4
[ybbyyrry, byybbrrb, ryyrrbbr, yrryybby]
```

A problematically long run

```
>java BeadFinder rrbbryyb ybbyyrry
1
2
3
[rrbbryyb, rbbrryyr, ybbyyrry]
>
>java BeadFinder rbrbryyb ybbyyrry
1
2
…
14^C
```

(Time-out after 9 hours on 1.1GHz PC due to incompleteness *or* due to inefficiency?)

Brothers and sisters problems

- *n* missionaries and *n* cannibals are one side of a river;
- And so is a boat that holds *c* people
- Find the most time efficient way of moving everyone to the other side without leaving a group of missionaries on either side outnumbered by the cannibals

[*Machine Intelligence*, 3, 1968]

What is a configuration?

- Start configuration ⟨n,n,true⟩
  - n missionaries, n cannibals and 1 boat on the initial side
- End configurations ⟨0,0,true⟩ and ⟨0,0,false⟩
  - 0 missionaries and 0 cannibals on initial side
- Consider next configurations for ⟨2,2,true⟩ for the n = 4 and c = 2 instance:
  - ⟨0,2,false⟩ √ cc ~~~ mmmmcc
  - ⟨1,1,false⟩ √ mc ~~~ mmmccc
  - ⟨2,0,false⟩ × mm ~~~ mmcccc

Triple class (1 of 2)

```
public class Triple
{
   private int miss, cann;
   private boolean boat;

   Triple(int miss, int cann, boolean boat)
   {
         this.miss = miss;
         this.cann = cann;
         this.boat = boat;
   }

   public int getMiss()
   {
         return miss;
   }
   // other accessor methods
```

Triple class (2 of 2)

```
   public boolean isValid(int n)
   {
      if (miss == n) return true;                  // case 1
      if (miss == 0) return true;                  // case 2
//       if (miss >= cann && (n - miss) >= (n - cann)) return true;
//       if (miss >= cann && (-miss) >= (-cann)) return true;
//       if (miss >= cann && cann >= miss) return true;
      if (miss == cann) return true;               // case 3
      return false;
   }

   public String toString()
   {
         return "(" + miss + ", " + cann + ", " + boat + ")";
   }

}
```

nextConfig() (1 of 2)

```
public LinkedList<Triple> nextConfigs(Triple config)
{
        LinkedList<Triple> result = new LinkedList<Triple>();
        for (int moveMiss = 0; moveMiss <= c; moveMiss++)
           for (int moveCann = 0; moveCann <= c - moveMiss; moveCann++)
                 if (config.getBoat())
                 {
                    int newMiss = config.getMiss() - moveMiss;
                    int newCann = config.getCann() - moveCann;
                    if (newMiss < 0) ;
                    else if (newCann < 0) ;
                    else
                    {
                       Triple triple = new Triple(newMiss, newCann, false);
                       if (triple.isValid(n)) result.add(triple);
                    }
                 }
                 else
```

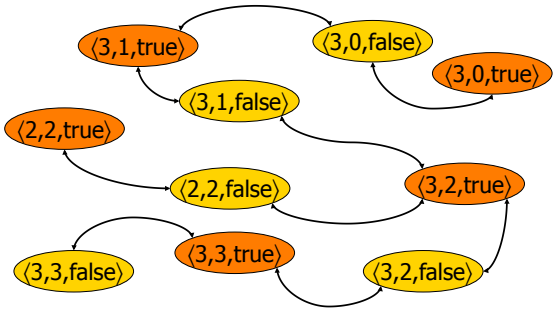
nextConfig() (2 of 2)

```
            else
            {
               int newMiss = config.getMiss() + moveMiss;
               int newCann = config.getCann() + moveCann;
               if (newMiss > n) ;
               else if (newCann > n) ;
               else
               {
                  Triple triple = new Triple(newMiss, newCann, true);
                  if (triple.isValid(n)) result.add(triple);
               }
            }
      return result;
} // 1
```

Sample runs with iterative deepening

```
>java BoatFinder 5 3
1
...
8
[(5, 5, true), (4, 4, false), (4, 4, true),
(3, 3, false), (3, 3, true), (0, 3, false),
(0, 3, true), (0, 0, false)]
>
>java BoatFinder 3 1
1
2
...
1043 ^C
```

Debugging n = 3 and c = 1



Counting configurations

- Consider n = 3 and initially ignore the boat:
- Cases 1 and 2 both give 4 = n+1 configurations
- Case 3 gives 2 = n-1 additional configurations
- Total of 2(2(n+1) + (n-1)) = 2(3n+1) configurations considering boat positions

mmm			ccc
mmm	c		cc
mmm	cc		c
mmm	ccc		
		mmm	ccc
	c	mmm	cc
	cc	mmm	c
	ccc	mmm	
		mmm	ccc
m	c	mm	cc
mm	cc	m	c
mmm	ccc		

depthLimitedIterativeDeepening method

```
public LinkedList<Triple> depthLimitedIterativeDeepening()
{
      for (int depth = 1; depth < 6*n+2; depth++)
      {
         System.out.println(depth);
         LinkedList<Triple> route = depthFirst(new Triple(n, n, true), depth);
         if (route != null) return route;
      }
      return null;
}
```

Sample runs with depth limited iterative deepening

```
>java BoatFinder 5 3
1
...
8
[(5, 5, true), (4, 4, false), (4, 4, true),
(3, 3, false), (3, 3, true), (0, 3, false),
(0, 3, true), (0, 0, false)]
>
>java BoatFinder 3 1
1
...
19
null
```

Generic concepts

	Route planning	*Bead puzzle*	*Missionaries and cannibals*	*8-puzzle*
state	town	configuration of 2 wheels	who is on what bank	?
operator	road graph	move graph	move graph	?
goal state(s)	destination town	unique target configuration	all people on other side of river	?
branching factor	~4	2(w-1)	~3	~4
solution	journey	series of wheel moves	series of boat and cargo moves	?