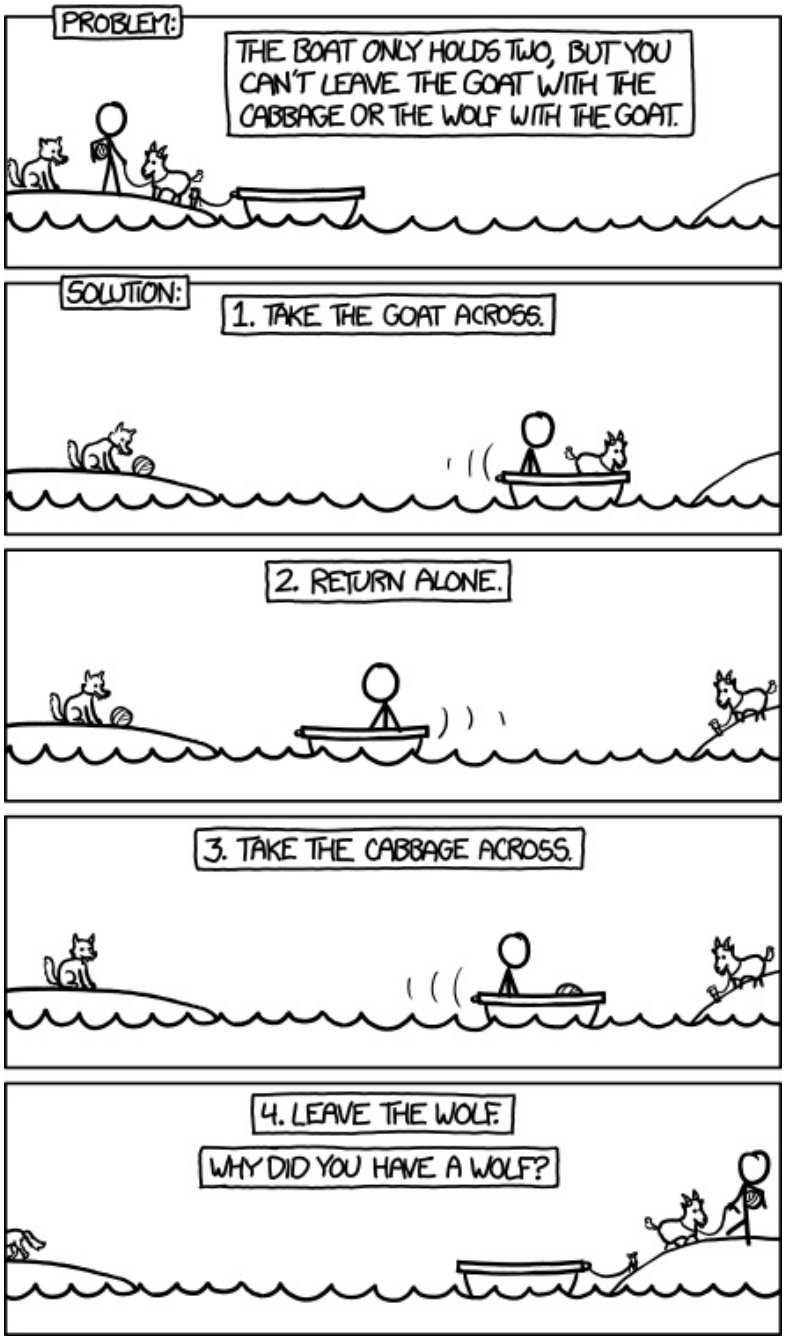
1 FWGC Puzzle using Graph Solution

A farmer wants to get his goat, wolf, and cabbage to the other side of the river. His boat isn't very big and it can only carry him and either the goat, the wolf, or the cabbage. Now...if he leaves the goat alone with the cabbage, the goat will gobble up the cabbage. If he leaves the wolf alone with the goat, the wolf will gobble up the goat. When the farmer is present, the goat and cabbage are safe from being eaten up by their predators.

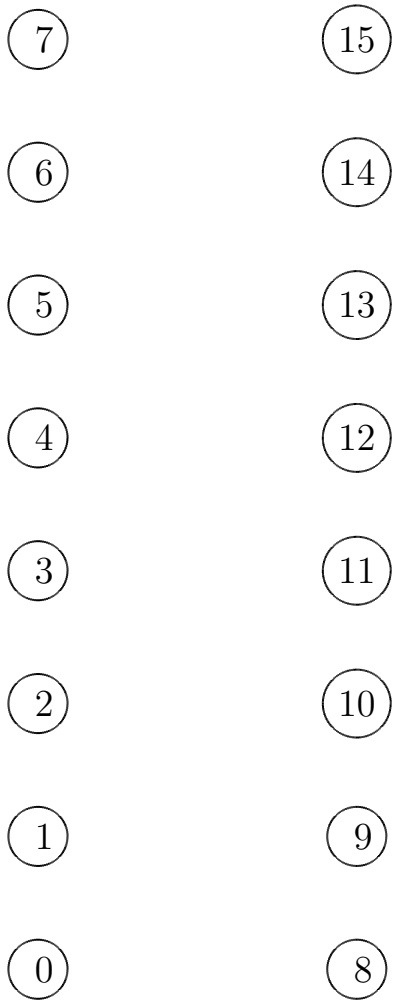
How can the farmer get everything safely to the other side of the river?



http://xkcd.com/1134/

1.1 States of Farmer, Wolf, Goat, Cabbage Problem

	Farmer	Wolf	Goat	Cabbage	Safe
0	south	south	south	south	Yes
1	south	south	south	north	Yes
2	south	south	north	south	Yes
3	south	south	north	north	No
4	south	north	south	south	Yes
5	south	north	south	north	Yes
6	south	north	north	south	No
7	south	north	north	north	No
8	north	south	south	south	No
9	north	south	south	north	No
10	north	south	north	south	Yes
11	north	south	north	north	Yes
12	north	north	south	south	No
13	north	north	south	north	Yes
14	north	north	north	south	Yes
15	north	north	north	north	Yes


7
6
5
4
3
2
1
0
15
14
13
12
11
10
9
8
South
North

1.2 C++ Solution of Farmer, Wolf, Goat, Cabbage Problem

```
/*  FWGC.cpp

    A breadth-first solution to the Farmer, Wolf,
    Goat and Cabbage problem.
 */

#include <iostream>
#include <iomanip>

#include "queueL.h"

using namespace std;

const unsigned int FARMER_MASK  = 0x08;
const unsigned int WOLF_MASK    = 0x04;
const unsigned int GOAT_MASK    = 0x02;
const unsigned int CABBAGE_MASK = 0x01;

    //  prototypes
int Safe(    int location );
int farmer(  int location );
int wolf(    int location );
int goat(    int location );
int cabbage( int location );

void ShowLocation( int onNorthSide );
void ShowStates();
void PrintMoveList( int route[], int nRoutes );
```

```
    //  inline functions

inline int farmer( int location )
{  return 0 != (location & FARMER_MASK); }

inline int wolf( int location )
{  return 0 != (location & WOLF_MASK); }

inline int goat( int location )
{  return 0 != (location & GOAT_MASK); }

inline int cabbage( int location )
{  return 0 != (location & CABBAGE_MASK); }
```

```
/*  Safe -- Test if location is safe.
    Return true if situation is safe.
 */

int Safe( int location )
{
       //  goat eats cabbage
    if( (goat(location) == cabbage(location)) &&
        (goat(location) != farmer(location))     )
    {
        return 0;
    }

       //  wolf eats goat
    if( (goat(location) == wolf(location)) &&
        (goat(location) != farmer(location))     )
    {
        return 0;
    }

       //  any other situation is safe
    return 1;
}
```

```
int main()
{
    Queue moves;
    const int MAX_STATES = 16;
    int route[MAX_STATES] = { -1 };

       //  Initialize route
    for( int i = 0 ; i < MAX_STATES ; i++ )
        route[i] = -1;
```

```
   //  all start on South side of river
moves.Insert( 0 );

const int MAX_PASSES = 100;
int iPass = 0;
while( !moves.IsEmpty() && iPass < MAX_PASSES ) {
    cout << "iPass: " << iPass << endl;
    cout << "    move queue:" << endl;
    moves.Print();

       //  get current location
    int location = moves.Delete();
    cout << "    location: " << location << endl;

    for( int iMove = 1 ; iMove <= 8 ; iMove <<= 1 )
    //for( int iMove = 1 ; iMove <= 8 ; iMove *= 2 )
    {
           //  farmer always moves
        int newLocation = location ^ (FARMER_MASK | iMove);

        if( Safe(newLocation) &&
            (route[newLocation] == -1) ) {
            route[newLocation] = location;
            moves.Insert( newLocation );
        }

        iPass++;
    }
}
```

```
      //  Display route
    cout << "\nPath (in reverse):" << endl;
    for( int location = MAX_STATES-1 ;
         location > 0 ;
         location = route[location] )
        cout << " " << location;

    cout << endl;

    PrintMoveList( route, MAX_STATES-1 );

      //  Show all possible states
    ShowStates();

    cout << "\nDone!" << endl;

    return 0;
}
```

```
/*  Display move list in human readable form  */

void PrintMoveList( int route[], int nRoutes )
{
    cout << endl;
    cout << "Farmer  |  Wolf   |  Goat   |  Cabbage |" << endl;
    cout << "--------+---------+---------+----------|" << endl;

    for( int location = nRoutes ;
         location > 0 ;
         location = route[location] )
    {
         ShowLocation( farmer(location)  );
         ShowLocation( wolf(location)    );
         ShowLocation( goat(location)    );
         ShowLocation( cabbage(location) );

         cout << endl;
    }

    cout << " south   |  south   |  south   |  south   |" << endl;
}

void ShowLocation( int onNorthSide )
{
    if( onNorthSide )
        cout << " north   | ";
    else
        cout << " south   | ";
}
```

```
void  ShowStates()
{
    cout << endl;
    cout << "    | Farmer  |  Wolf   |  Goat   |  Cabbage |  Safe  |
    cout << "----+---------+---------+---------+----------+--------|

    for( int i = 0 ; i < 16 ; i++ )
    {
        cout << setw(3) << i << " | ";
        ShowLocation( farmer(i) );
        ShowLocation( wolf(i) );
        ShowLocation( goat(i) );
        ShowLocation( cabbage(i) );

        ShowLocation( Safe(i) );

        cout << endl;
    }
}
```

Output:

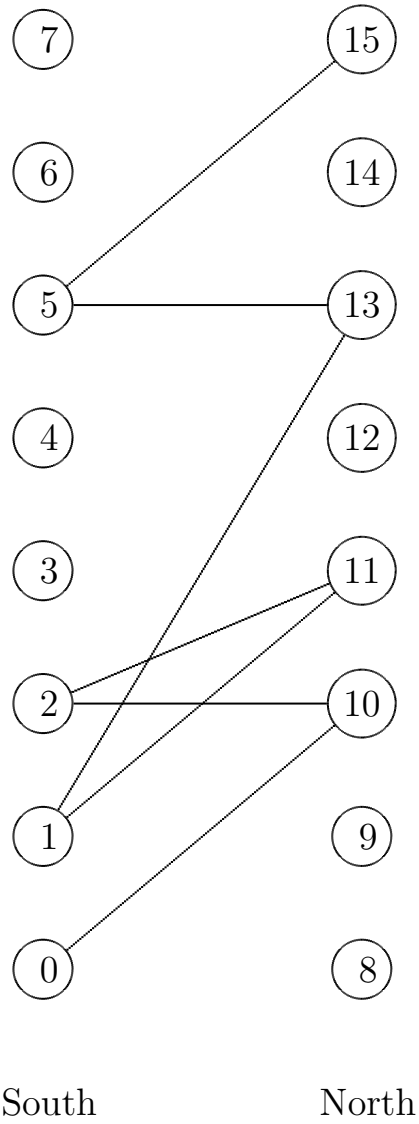
```
iPass: 0
0
iPass: 4
10
iPass: 8
0
2
iPass: 12
2
iPass: 16
11
14
iPass: 20
14
1
iPass: 24
1
4
iPass: 28
4
13
iPass: 32
13
iPass: 36
5
iPass: 40
15

Path:
 15 5 13 1 11 2 10
Done!
```

1.3 Solution

```
Path (in reverse):
 15 5 13 1 11 2 10

Farmer   |  Wolf    |  Goat    |  Cabbage |
---------+----------+----------+----------|
 north   |  north   |  north   |  north   |
 south   |  north   |  south   |  north   |
 north   |  north   |  south   |  north   |
 south   |  south   |  south   |  north   |
 north   |  south   |  north   |  north   |
 south   |  south   |  north   |  south   |
 north   |  south   |  north   |  south   |
 south   |  south   |  south   |  south   |
```


7
6
5
4
3
2
1
0
15
14
13
12
11
10
9
8
South
North

1.4 Scheme Solution of FWGC Problem

```
;  Farmer wolf goat cabbage problem

(define (make-state f w g c) (list f w g c))

(define (nth n aList)
  (cond ((empty? aList) empty)
        ((= 0 n) (first aList))
        (else (nth (sub1 n) (rest aList)))))

(define (farmer-side state)
  (nth 0 state))

(define (wolf-side state)
  (nth 1 state))

(define (goat-side state)
  (nth 2 state))

(define (cabbage-side state)
  (nth 3 state))
```

```
(define (safe state)
  (cond ((and (symbol=? (goat-side state) (wolf-side state))
              (not (symbol=? (farmer-side state) (wolf-side state)))
              empty)
         ((and (symbol=? (goat-side state) (cabbage-side state))
               (not (symbol=? (farmer-side state) (goat-side state)))
               empty)
         (else state)))

; safe Tests:
;(print "all on one side")
;(define start (make-state 'w 'w 'w 'w))
;(safe start)
;(safe '(w w w w))    ;;  all on one side
;(safe '(e e e e))
;(print "wolf eats goat")
;(safe '(w e e w))    ;;  wolf eats goat
;(print "goat eats cabbage")
;(safe '(w w e e))    ;;  goat eats cabbage
;(print "object alone")
;(safe '(e w w w))    ;;  farmer alone
;(safe '(w e w w))    ;;  wolf alone
;(safe '(w w e w))    ;;  goat alone
;(safe '(w w w e))    ;;  cabbage alone
```

```
(define (opposite side)
  (cond ((symbol=? side 'e) 'w)
        ((symbol=? side 'w) 'e)))

;(print "Test opposite:")
;(opposite 'w)
;(opposite 'e)
```

```
(define (farmer-takes-self state)
  (safe (make-state (opposite (farmer-side state))
                    (wolf-side state)
                    (goat-side state)
                    (cabbage-side state))))

(define (farmer-takes-wolf state)
  (cond ((equal? (farmer-side state) (wolf-side state))
         (safe (make-state (opposite (farmer-side state))
                    (opposite (wolf-side state))
                    (goat-side state)
                    (cabbage-side state))))
        (else empty)))

(define (farmer-takes-goat state)
  (cond ((equal? (farmer-side state) (goat-side state))
         (safe (make-state (opposite (farmer-side state))
                    (wolf-side state)
                    (opposite (goat-side state))
                    (cabbage-side state))))
        (else empty)))

(define (farmer-takes-cabbage state)
  (cond ((equal? (farmer-side state) (cabbage-side state))
  (safe (make-state (opposite (farmer-side state))
                    (wolf-side state)
                    (goat-side state)
                    (opposite (cabbage-side state)))))
        (else empty)))
```

```
(define (path state goal)
  (cond ((equal? state goal) 'success)
        (else (or (path (farmer-takes-self state) goal)
                  (path (farmer-takes-wolf state) goal)
                  (path (farmer-takes-goat state) goal)
                  (path (farmer-takes-cabbage state) goal)))))

  ;;; Find solution
(path (make-state 'w 'w 'w 'w) (make-state 'e 'e 'e 'e))
```

2 Another look at Quicksort

2.1 Quicksort Algorithm

http://en.wikipedia.org/wiki/Quicksort

function quicksort(array)
  var list *less*, *greater*
  if length(array) ≤ 1
    return array // zero or one elements is already sorted
  select and remove a pivot value, *pivot*, from array
  for each x in array
    if $x \leq$ *pivot* then append x to *less*
    else append x to *greater*
  return concatenate(quicksort(*less*), *pivot*, quicksort(*greater*))

2.2 The Lazy Quicksort

A version of *quicksort* in Clojure.
https://eddmann.com/posts/quicksort-in-clojure/

```
(defn qsort [[pivot & tail]]
    (when pivot
      (lazy-cat (qsort (filter #(< % pivot) tail))
                [pivot]
                (qsort (remove #(< % pivot) tail)))))
```

2.3 Quicksort Using A Functional C Approach

```
listPtr Quicksort( listPtr l )
{
   if( l == NULL )
      return NULL;
   else
   {
      int pivot = head(l);
      listPtr left  = filter( lessThan, pivot, tail(l) );
      listPtr right = removeFrom( lessThan, pivot, tail(l) );

      return cat( Quicksort(left), cons(pivot, Quicksort(right)) );
   }
}
```

Is this a large departure from what we have done in the past?

Functions:

1. `head()`
2. `tail()`
3. `cons()`
4. `cat()`
5. `filter()`
6. `removeFrom()`