COIS 4050H – Assignment #2

Simon Willshire (0491272)

1.

Selection (Version 2) of groups size 5 is proven to be worst-case time O(n), however for the following algorithm, of groups size 3:

Reference Code:

```
# Selection Alg. (Version 2): With groups size 3
# In Pseudo-Python

def Selection(n[], i):
    if (len(n) < 50):
        return sort(n[])                # Return ith element in sorted n elements.
    else:
        gsize = floor(len(n) / 3)       # Divide into groups of 3, with up to 2 leftover elements.

        # Fill the groups,
        for i in xrange(1, len(n):
            ++g % 3
            groups[g] = n[i];

        # Sort groups
        for g in xrange(1, gsize):
            sort groups[g]

        # Find median of groups (+1 finds middle index)
        for i in xrange(1, gsize):
            med[i] = groups[i + 1]

        # Recursively call for smaller sequence of medians,
        m = Selection(med, ceil(gsize / 2))

        # Using m as pivot element,
        for l in xrange(1, floor(len(m)/2)):
            left[l] = m[l]
        for r in xrange(1, ceil(len(m)/2)):
            right[r] = m[r]

        # Partition and recurse by using m as pivot,
        if (len(left) >= i):
            return Selection(left, k)
        else:
            if (len(n) - len(right) >= k):
                return m
            else:
                return Selection(right, k - len(n) + len(right))
```

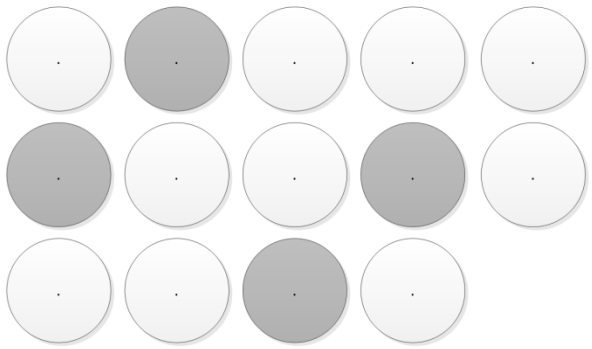
Proof:

Let T(n) represent the worst case time complexity for Selection (Version 2) with group sizes of 3.
Note: *Assume sorting call is O(1) time, as length of n is small (< 50 elements).*

Median of Medians: Finding the cost of grouping by 3:

Each median found will throw away more elements at every stage of recursion over the grouping of 5 elements, which is able to get a more accurate reading of the numbers available.

Each Median is then at least: $\left\lceil \frac{\left\lceil \frac{n}{3} \right\rceil}{2} \right\rceil - \left\lceil \frac{n}{6} \right\rceil = 2(\frac{n}{3})$



N = 14, Groups of 3.
Grey Representing Median

In General:

$$T(n) = \begin{cases} \theta(n) & \text{If n < 50, or otherwise.} \\ T\left(\left\lceil\frac{n}{3}\right\rceil\right) + T\left(n - 2\left\lceil\frac{n}{3}\right\rceil\right) + \theta(1) & \end{cases}$$

Groups of 5 Inequality: $T(n) \leq T\left(\frac{n}{5}\right) + T\left(7\frac{n}{10}\right) + O(n)$

Groups of 3 Inequality: $T(n) \leq T\left(\frac{n}{3}\right) + T\left(2\frac{n}{3}\right) + O(n)$

Proof by Induction

Base Case: $T(n) \geq \theta(n)$ for n < 50 (linear sort algorithm, ex. mergesort)

Inductive Hypothesis: Assume true for 1 to n-1.

Inductive Step:

for $n \geq 50$:

$$T(n) \leq T\left(\left\lceil\frac{n}{3}\right\rceil\right) + T\left(2\left\lceil\frac{n}{3}\right\rceil\right) + O(n)$$

$$\leq n\left(\left\lceil\frac{n}{3}\right\rceil\right) + n\left(2\left\lceil\frac{n}{3}\right\rceil\right) + O(n)$$

$$\leq \frac{n^2}{3} + \frac{2n^2}{3} + O(n)$$

$$\leq \frac{13n^2}{15} + O(n)$$

$$\neq O(n)$$

Therefore the Selection (Version 2) algorithm with groups of 3 has non-linear time complexity.

2.

Design

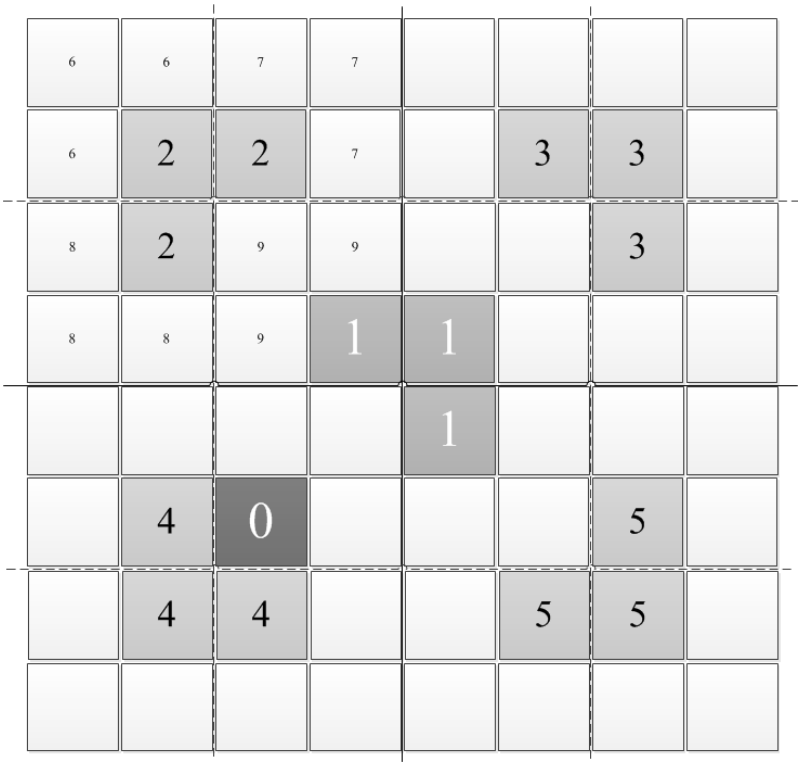
A grid with $2^k x\ 2^k$ dimensions may be split into $4x\ 2^{k-1} x\ 2^{k-1}$ boards, and can continue to do so recursively decrementing k until a final breaking case of $2x2$ board dimensions.

Each grid must be filled with a full (3 tile) piece, a part of another piece, or the first tile (labeled 0).

For each 2x2 segment, identify the missing segment, based on the last placed tile location which can be set from the recursive call.

The first should be placed with surrounding tile before the recursive call is invoked, as it is detrimental to the algorithm, as it may be placed on even or odd tiles, making the algorithm more bloated than it needs to be.

Once a call has placed its tile, it will call itself 4 times, one for each of the grid segments it may have produced, provided that k > 2 (recursive break at 2x2).



Note: *Tiles marked one, missing bottom left quadrant, where first tile marked 0 is placed.*

First Split:
calling 4 grids 4x4

Second split
calling 4 grids 2x2

Implementation in C#

Where k begins at 3 ($2^3 x\ 2^3$) grid

Note:

missing =
{TOP_LEFT = 1,
TOP_RIGHT = 2,
BOTTOM_LEFT = 3,
BOTTOM_RIGHT = 4};

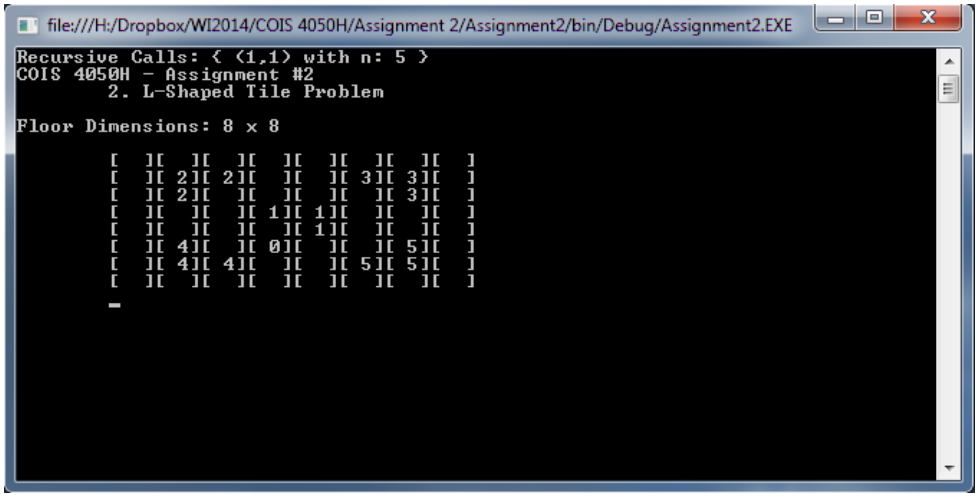
For **placeTile(…)** call which checks if the location is free, and places the tiles according to the coordinates given and missing value.

Every recursive call divides the dimensions of the grid, as well as the coordinates of the grid portion to be called into 2.

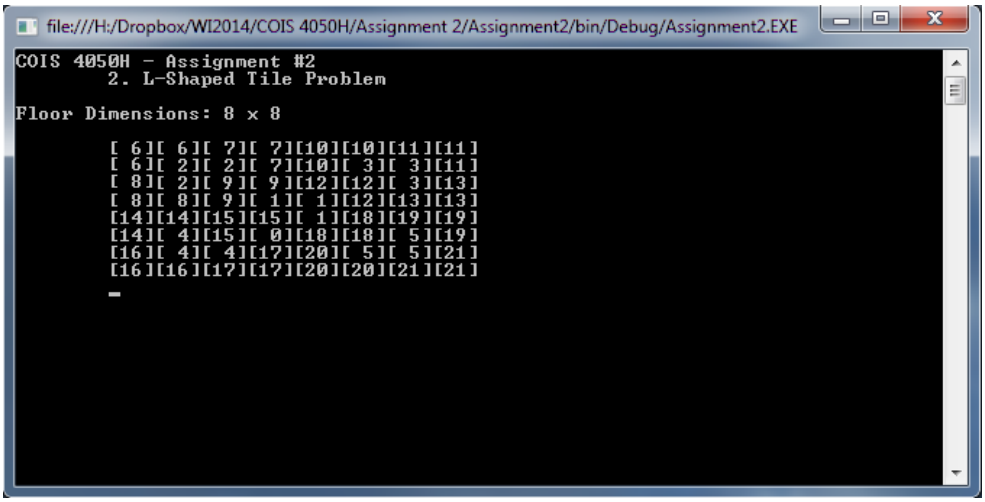
Using **n** as our current grid dimensions of one side, we are able to recursively call other grids from our section of grid. This is done by stepping **x + n,** and **y + n** for grids left and below the current recursed grid.

```
static void recursePlaceTile(int n, int x, int y)
{
    int missing = 0;

    if (n == tileCount)
    {
        if (oY + 1 > (tileCount / n))
            if (oX > (tileCount / n) - 1)
                missing = 3;
            else
                missing = 4;
        else
            if (oX > (tileCount / n) - 1)
                missing = 1;
            else
                missing = 2;
    }
    else
    {
        if ((y % ((n * 2) + 1)) < (n / 2))
            if (x > (tileCount / n) - 1)
                missing = 3;
            else
                missing = 4;
        else
            if (x > (tileCount / n) - 1)
                missing = 1;
            else
                missing = 2;
    }

    placeTile(missing, x, y);

    // Recurse until segment of 2x2 (2^1),
    if (n > 2)
    {
        n /= 2;
        x /= 2;
        y /= 2;

        // Recurse for each quadrant split,
        recursePlaceTile(n, x, y);
        recursePlaceTile(n, x + n, y);
        recursePlaceTile(n, x, y + n);
        recursePlaceTile(n, x + n, y + n);

        Console.WriteLine("Recursive Calls: { ("+x+","+y+") with n: " + (n+1) +" }");
    }

}
```

Step k == 4 (Second Recursive step):



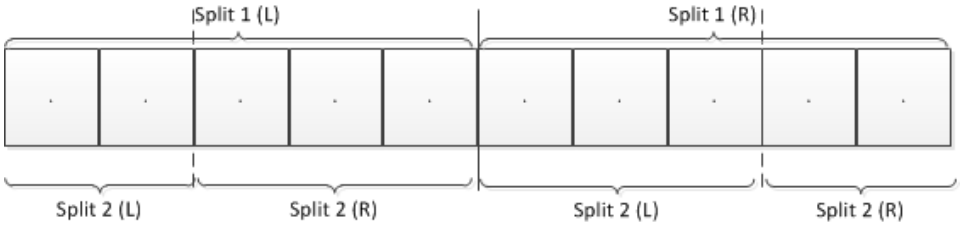
Step k == 2 (Third Recursive step):



3.

Design

An array of integers may be split into $n/2$, and those arrays (left and right) can be halved once again into $n/4$ (Where n represents the original array size).



Once the arrays have been split to comparable sizes (pair of elements), each can be compared. If any arrays are of size one, use them in the next comparison after an element has been eliminated.

Once all arrays have been compared, if any element still exists, it is then the majority number in the array.

Implementation in C#

Where array size (n) is 40,

Note: *I misspelled elements as elephants, so I had a little fun with the comments.*

Having assembled an array, and invoking **findMajority()**, the call will split the array into two.

Every split, left side takes floor, right side takes ceiling of the division (Otherwise may have missing elements.

Once the array has been equally split, they are recursively called for left and right.

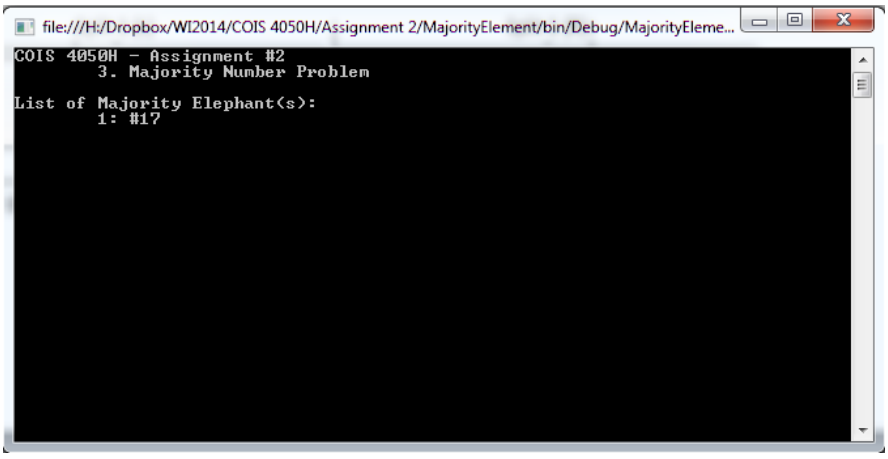
If any majority element(s) are returned, the list sizes are compared; one with more elements will be of greater majority.

Once the recursive break of size 2 occurs, elements are compared individually, returning a majority or null.

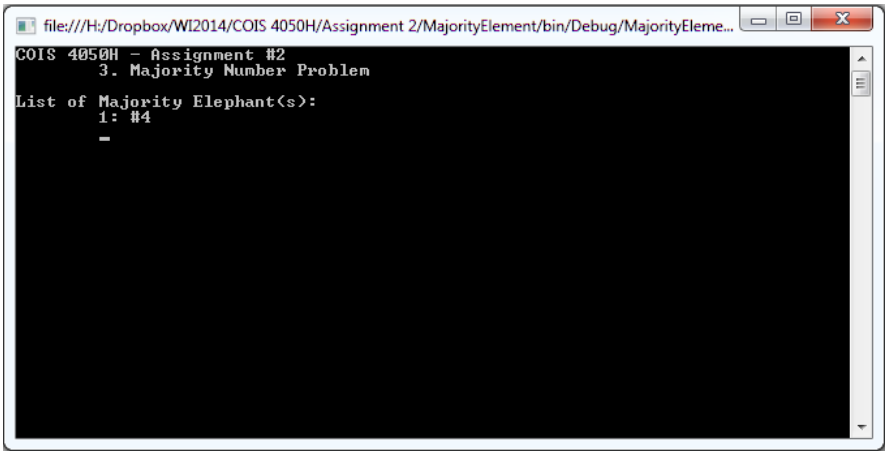
If any arrays of size 1 occur, the result is returned.

```
static int[] findMajority(int[] arr)
{
    // Recurse until we have comparison of 2 elephants,
    if (arr.Length > 2)
    {
        int[] left = new int[(int)Math.Floor(Convert.ToDouble(arr.Length) / 2)];
        int[] right = new int[(int)Math.Ceiling(Convert.ToDouble(arr.Length) / 2)];

        for (int l = 0; l < left.Length; l++)
            left[l] = arr[l];
        for (int r = 0; r < right.Length; r++)
            right[r] = arr[left.Length + r];

        left = findMajority(left);
        right = findMajority(right);

        if (left != null && right != null)
            if (left.Length > right.Length)
                return left;
            else //if (left.Length < right.Length)
                return right;
        else if (left != null)
            return left;
        else
            return right;
    }
    else if (arr.Length == 2)       // The arrays are of length 2, compare elephants,
        {
            if (arr[0] == arr[1])   // Resize the to a single majority elephant,
            {
                int e = arr[0];
                arr = new int[1];
                arr[0] = e;
                return arr;         // The elephants are the same, return;
            }
            else
                return null;        // No majority elephant, return null.
        }
    else                            // Join any lost elephants,
        return arr;
}
```

For 40 numbers randomly generated from 0 to 100:



For 40 numbers randomly generated from 1 to 5:



Time Complexity

Split: $\theta(1)$

Recursion: $2T\left(\frac{n}{2}\right)$

Comparison: $2n$

Recurrence Relation: $2T\left(\frac{n}{2}\right) + 2n$

Using the Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

If $f(n) \in \theta\left(n^{\log_b a}\right)$

$\leq T(n) \in \theta\left(n^{\log_b a}\right)$: Case 2 (no offset constant)

$\leq T(n) \in \theta(nlogn)$: $\log_b a = \log_2 n = logn$

Therefore this majority algorithm has a time complexity of: $\boldsymbol{\theta(nlogn)}$