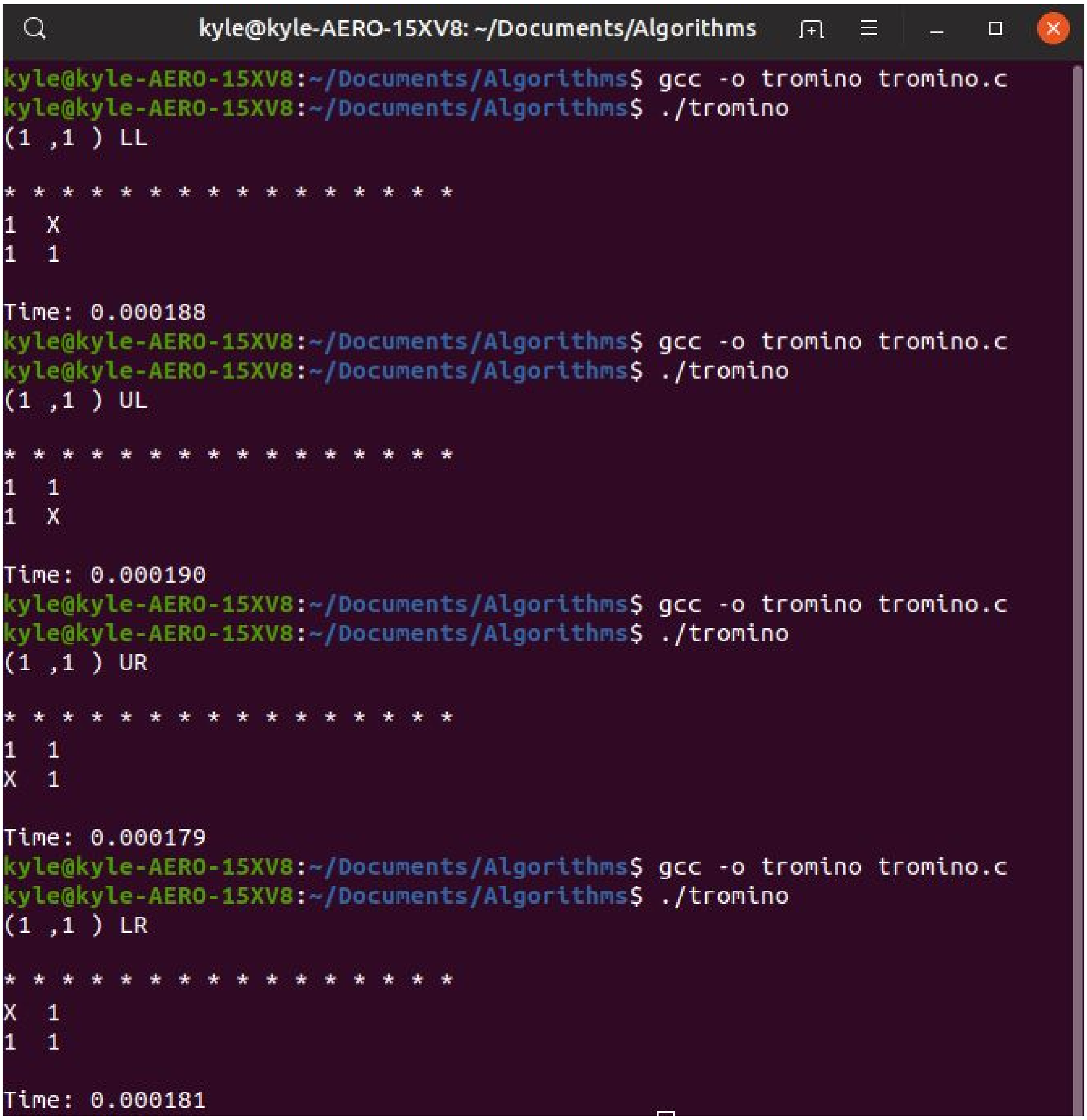
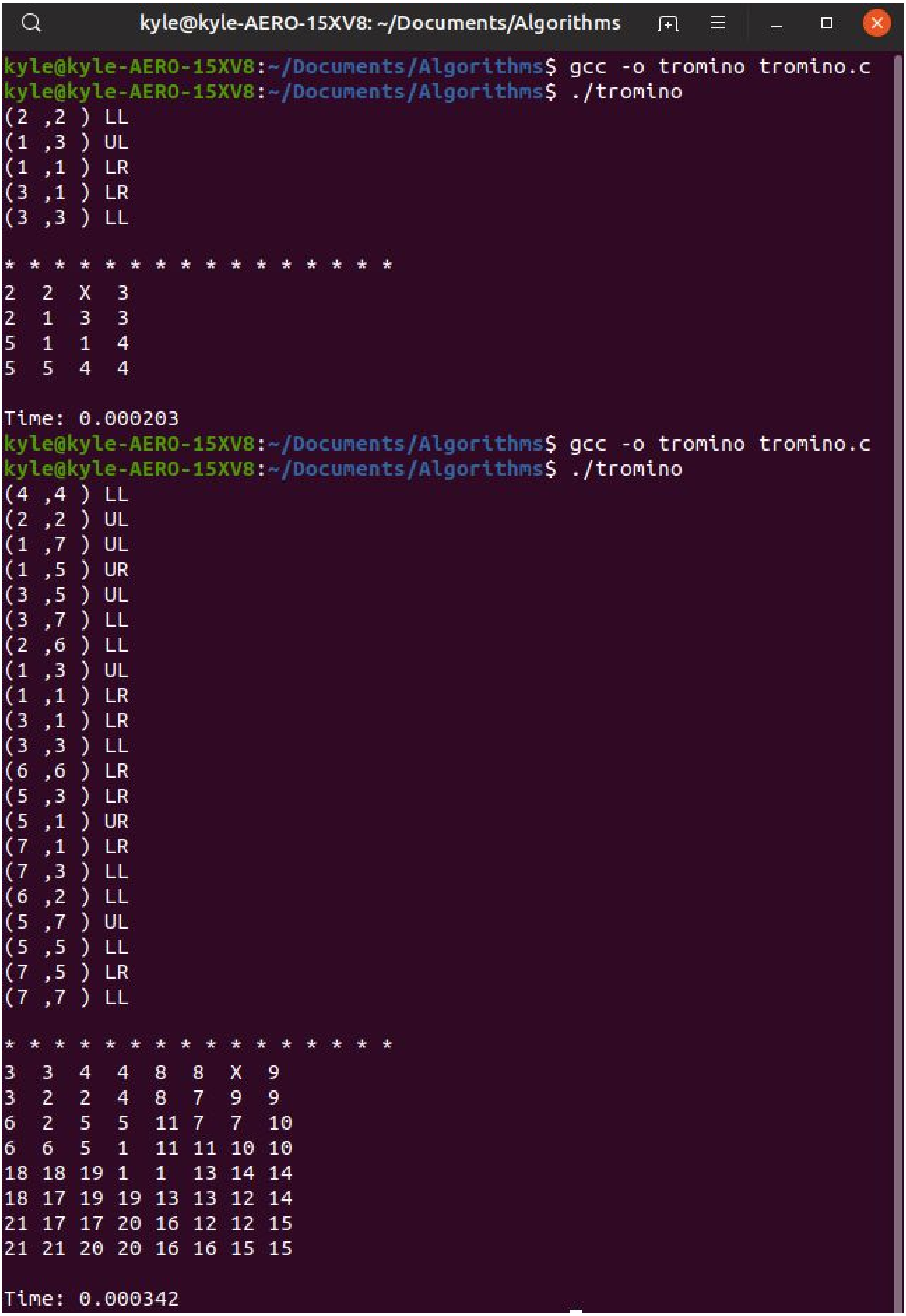
Kyle Hippe
Algorithms Assignment 1

## Chapter Exercises 5.1 & 5.2



```
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ gcc -o tromino tromino.c
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ ./tromino
(1 ,1 ) LL

* * * * * * * * * * * * * * * *
1  X
1  1

Time: 0.000188
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ gcc -o tromino tromino.c
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ ./tromino
(1 ,1 ) UL

* * * * * * * * * * * * * * * *
1  1
1  X

Time: 0.000190
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ gcc -o tromino tromino.c
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ ./tromino
(1 ,1 ) UR

* * * * * * * * * * * * * * * *
1  1
X  1

Time: 0.000179
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ gcc -o tromino tromino.c
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ ./tromino
(1 ,1 ) LR

* * * * * * * * * * * * * * * *
X  1
1  1

Time: 0.000181
```

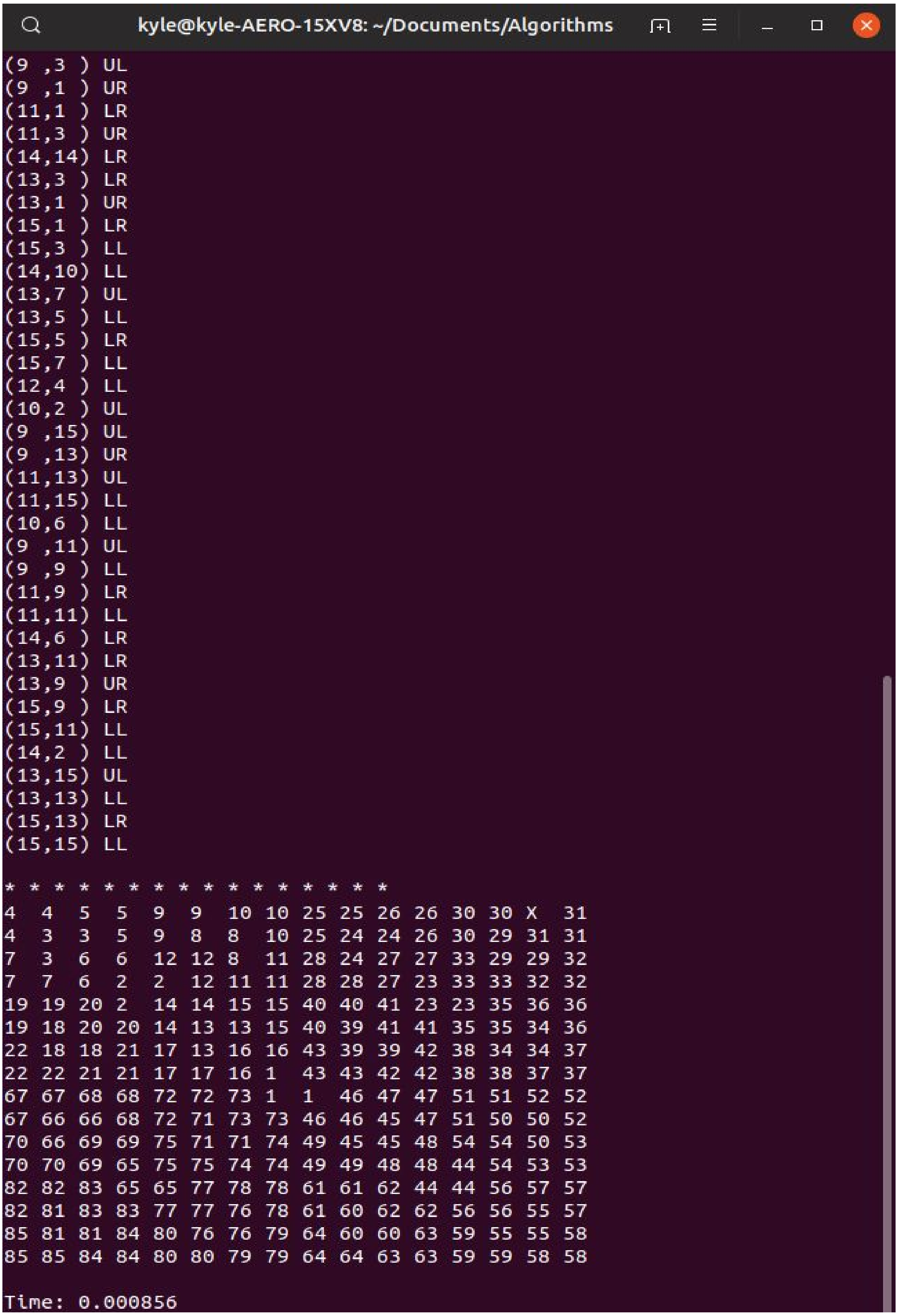
The picture above illustrates that the program successfully handles all base cases (deficient 2 by 2 boards) and can place Lower Left trominoes, Upper Left trominoes, Upper Right trominoes, and Lower Right trominoes while successfully printing the coordinates of the tile in reference to their inside corner.



```
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ gcc -o tromino tromino.c
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ ./tromino
(2 ,2 ) LL
(1 ,3 ) UL
(1 ,1 ) LR
(3 ,1 ) LR
(3 ,3 ) LL

* * * * * * * * * * * * * * * *
2  2  X  3
2  1  3  3
5  1  1  4
5  5  4  4

Time: 0.000203
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ gcc -o tromino tromino.c
kyle@kyle-AERO-15XV8:~/Documents/Algorithms$ ./tromino
(4 ,4 ) LL
(2 ,2 ) UL
(1 ,7 ) UL
(1 ,5 ) UR
(3 ,5 ) UL
(3 ,7 ) LL
(2 ,6 ) LL
(1 ,3 ) UL
(1 ,1 ) LR
(3 ,1 ) LR
(3 ,3 ) LL
(6 ,6 ) LR
(5 ,3 ) LR
(5 ,1 ) UR
(7 ,1 ) LR
(7 ,3 ) LL
(6 ,2 ) LL
(5 ,7 ) UL
(5 ,5 ) LL
(7 ,5 ) LR
(7 ,7 ) LL

* * * * * * * * * * * * * * * *
3  3  4  4  8  8  X  9
3  2  2  4  8  7  9  9
6  2  5  5  11 7  7  10
6  6  5  1  11 11 10 10
18 18 19 1  1  13 14 14
18 17 19 19 13 13 12 14
21 17 17 20 16 12 12 15
21 21 20 20 16 16 15 15

Time: 0.000342
```

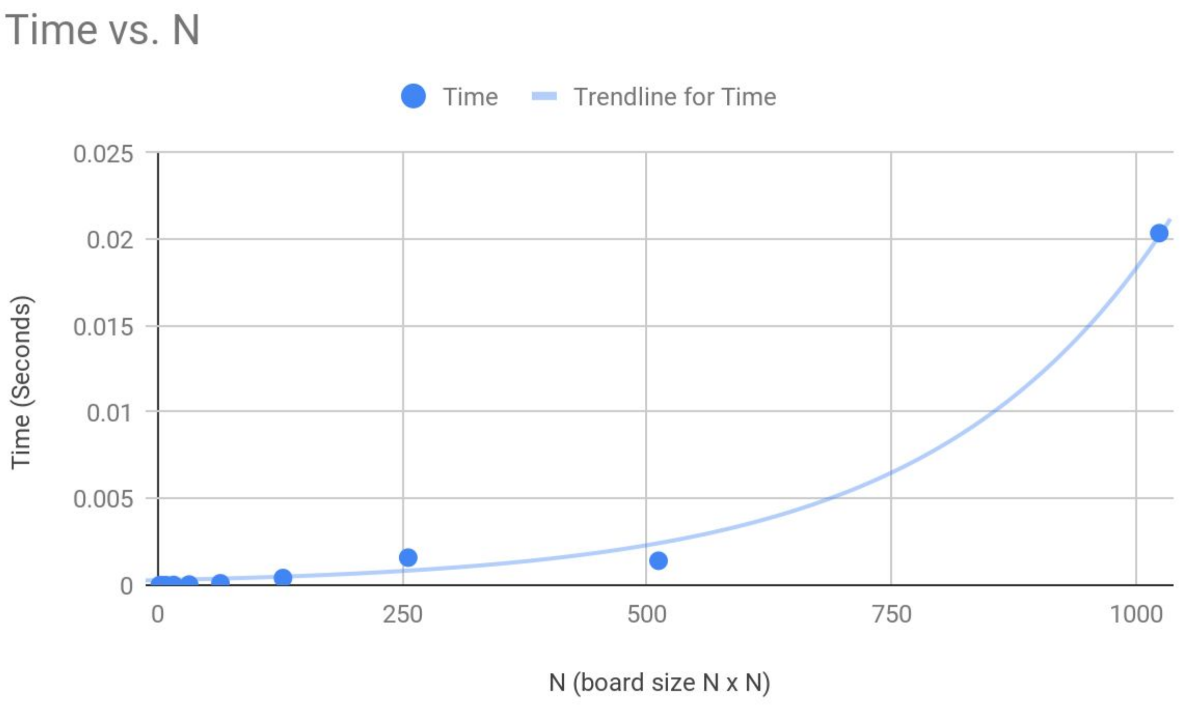
The above picture shows successful tilings and coordinates of deficient 4 by 4 boards and deficient 8x8 boards since they can be broken down into deficient 2 by 2 boards

**//shows 16x16**



```
(9 ,3 ) UL
(9 ,1 ) UR
(11,1 ) LR
(11,3 ) UR
(14,14) LR
(13,3 ) LR
(13,1 ) UR
(15,1 ) LR
(15,3 ) LL
(14,10) LL
(13,7 ) UL
(13,5 ) LL
(15,5 ) LR
(15,7 ) LL
(12,4 ) LL
(10,2 ) UL
(9 ,15) UL
(9 ,13) UR
(11,13) UL
(11,15) LL
(10,6 ) LL
(9 ,11) UL
(9 ,9 ) LL
(11,9 ) LR
(11,11) LL
(14,6 ) LR
(13,11) LR
(13,9 ) UR
(15,9 ) LR
(15,11) LL
(14,2 ) LL
(13,15) UL
(13,13) LL
(15,13) LR
(15,15) LL

*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *
4  4  5  5  9  9  10 10 25 25 26 26 30 30 X  31
4  3  3  5  9  8  8  10 25 24 24 26 30 29 31 31
7  3  6  6  12 12 8  11 28 24 27 27 33 29 29 32
7  7  6  2  2  12 11 11 28 28 27 23 33 33 32 32
19 19 20 2  14 14 15 15 40 40 41 23 23 35 36 36
19 18 20 20 14 13 13 15 40 39 41 41 35 35 34 36
22 18 18 21 17 13 16 16 43 39 39 42 38 34 34 37
22 22 21 21 17 17 16 1  43 43 42 42 38 38 37 37
67 67 68 68 72 72 73 1  1  46 47 47 51 51 52 52
67 66 66 68 72 71 73 73 46 46 45 47 51 50 50 52
70 66 69 69 75 71 71 74 49 45 45 48 54 54 50 53
70 70 69 65 75 75 74 74 49 49 48 48 44 54 53 53
82 82 83 65 65 77 78 78 61 61 62 44 44 56 57 57
82 81 83 83 77 77 76 78 61 60 62 62 56 56 55 57
85 81 81 84 80 76 76 79 64 60 60 63 59 55 55 58
85 85 84 84 80 80 79 79 64 64 63 63 59 59 58 58

Time: 0.000856
```

The above picture shows tiling of a deficient 16 by 16 board along with coordinates.

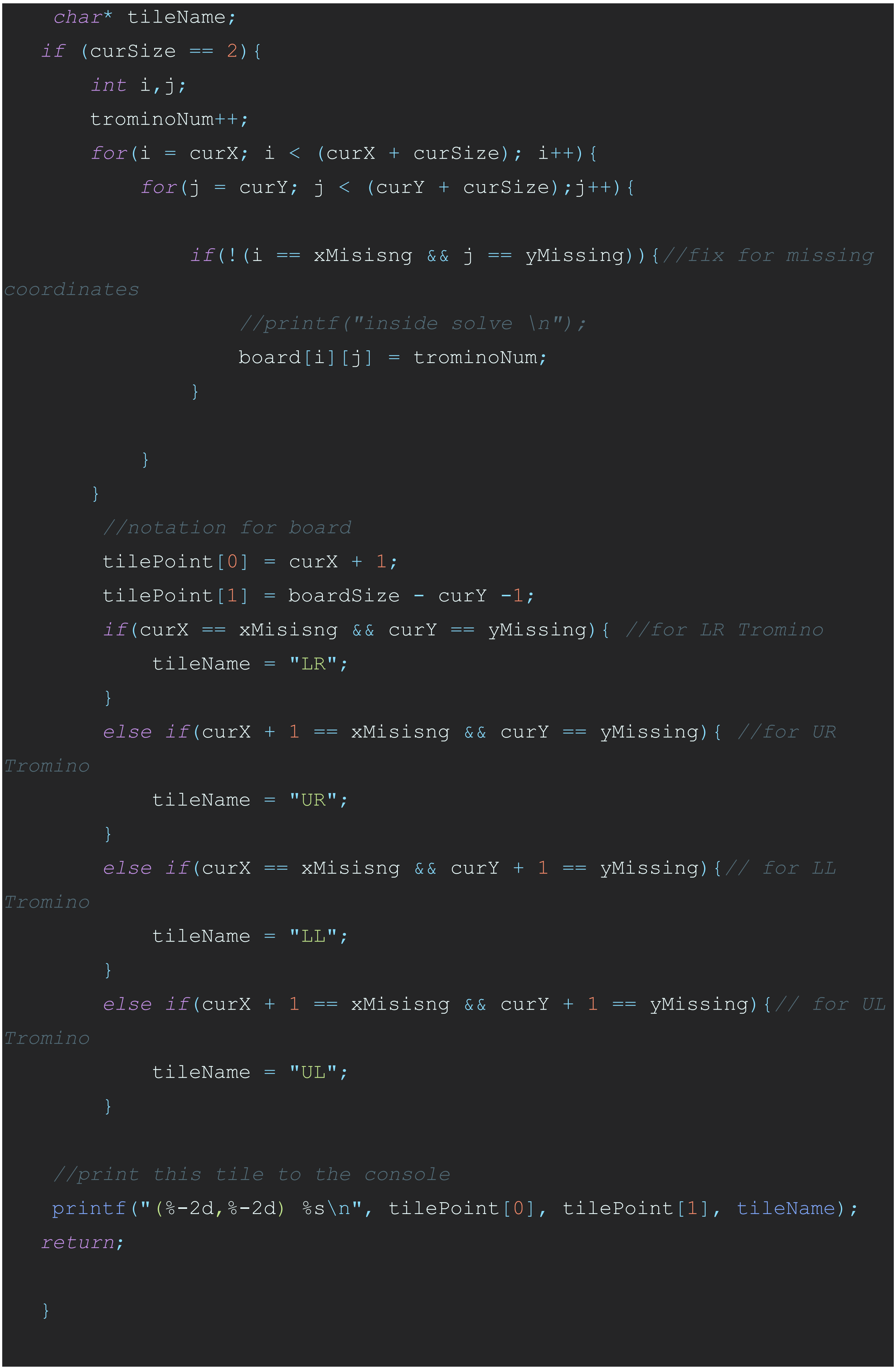


| N (Board size) | Time |
|---|---|
| 2 | 0.000005 |
| 4 | 0.000006 |
| 8 | 0.000004 |
| 16 | 0.000016 |
| 32 | 0.000037 |
| 64 | 0.000110 |
| 128 | 0.000414 |
| 256 | 0.001592 |
| 512 | 0.001412 |
| 1024 | 0.020348 |
| 2048 | Segmentation Fault |

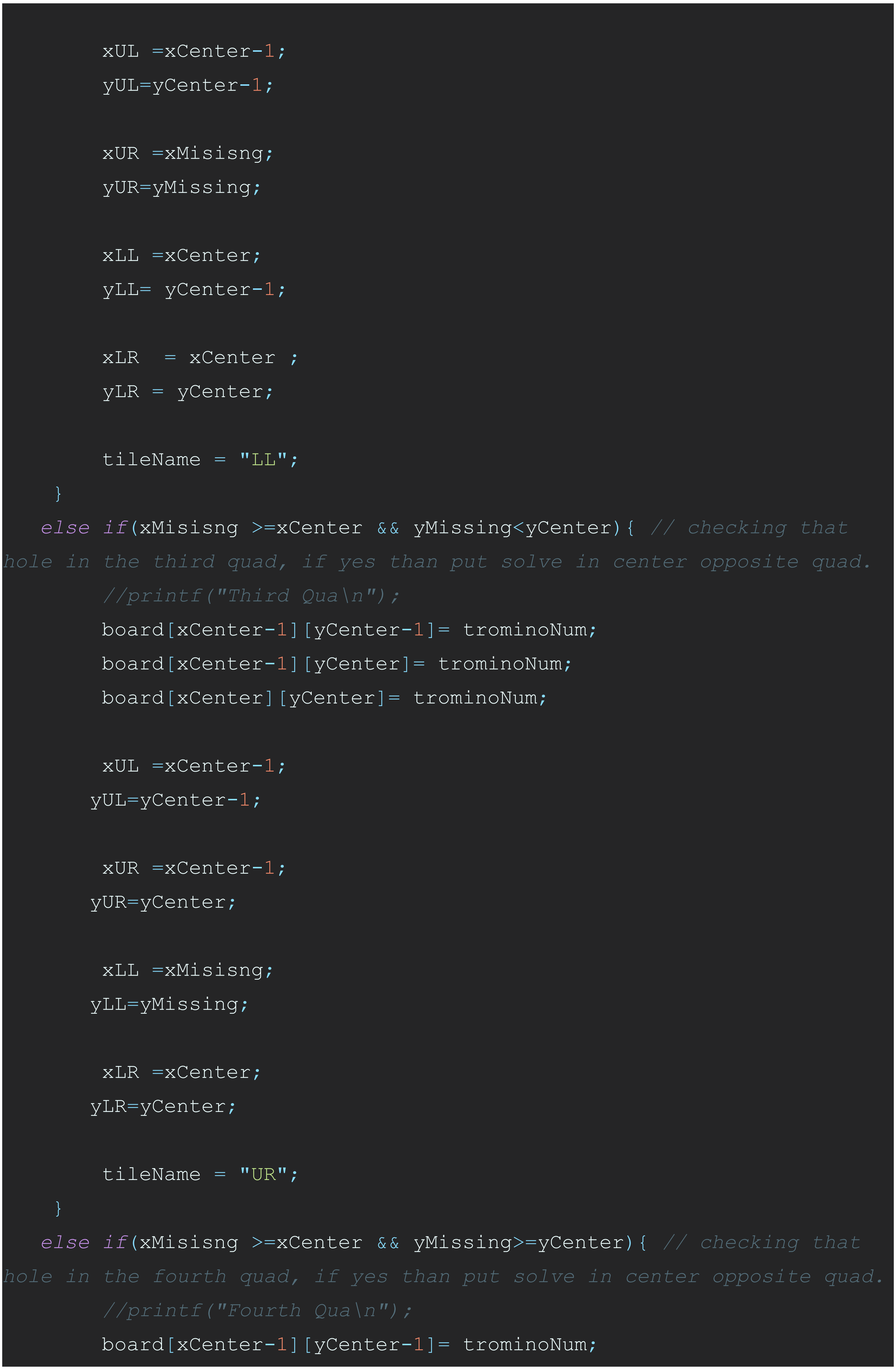
The chart and tables above represent the timings of each run measured from start of algorithm to just before drawing the board on the console. The graph shows the plot of these points as compared to a n^2 line, showing time for execution roughly follows an n^2 trend, as required by the algorithm.

The code:

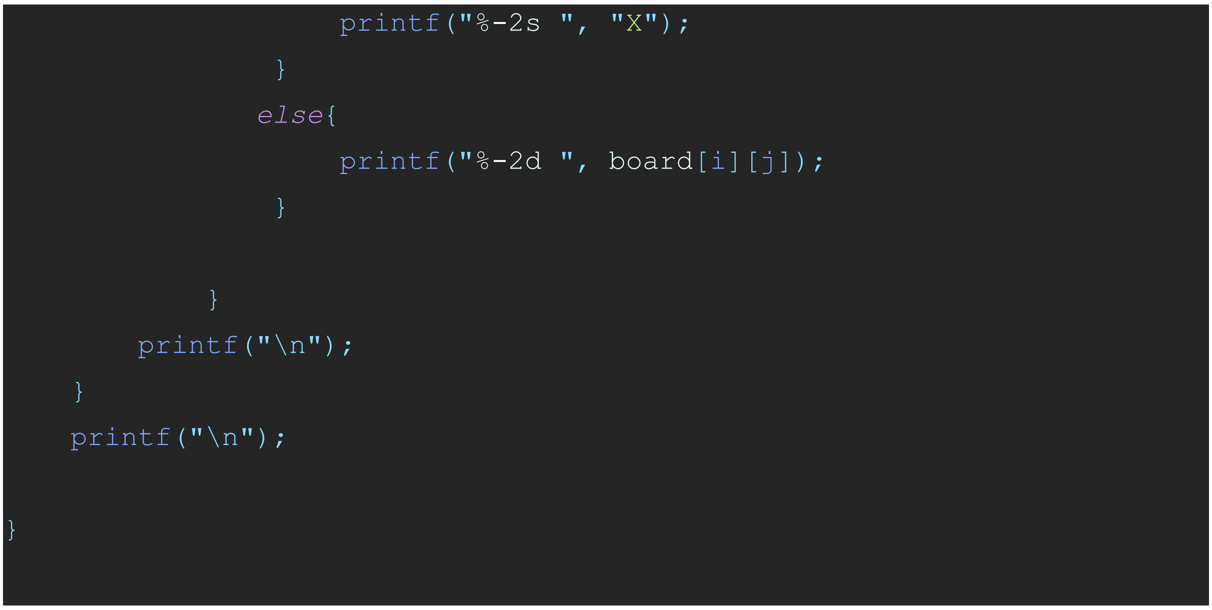
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int trominoNum = 0;


void solve(int boardSize, int xMisisng, int yMissing, int curX, int curY, int curSize, int (*board)[boardSize]);
void printMatrix(int coloumn, int (*board)[coloumn]);

int main(int argc, char *args[]){
   int size = 16;
    int board [size][size];
    int xMissing = 0, yMissing = size-2;
    clock_t start, end;
    double totalTime;

    board[xMissing][yMissing] = -1;

    start = clock();
   solve(size, xMissing, yMissing, 0,0, size, board);
    end = clock();
    totalTime = ((double) (end - start)) / CLOCKS_PER_SEC;

   printMatrix(size, board);
    printf("Time: %f\n", totalTime);

}

//board size is total board size
//xMissing, yMissing, are coordinates of missing square
//curX curY are coordinates of subsections (0,0)
//curSize is the size of the current sub-board
//board is the overall representation of the board
void solve(int boardSize, int xMisisng, int yMissing, int curX, int curY, int curSize, int (*board)[boardSize]){
   //base case
    int tilePoint[2] = {0,0};
```



```
    char* tileName;
   if (curSize == 2){
       int i,j;
       trominoNum++;
       for(i = curX; i < (curX + curSize); i++){
           for(j = curY; j < (curY + curSize);j++){

               if(!(i == xMisisng && j == yMissing)){//fix for missing
coordinates
                   //printf("inside solve \n");
                   board[i][j] = trominoNum;
               }

           }
       }
        //notation for board
        tilePoint[0] = curX + 1;
        tilePoint[1] = boardSize - curY -1;
        if(curX == xMisisng && curY == yMissing){ //for LR Tromino
            tileName = "LR";
        }
        else if(curX + 1 == xMisisng && curY == yMissing){ //for UR
Tromino
            tileName = "UR";
        }
        else if(curX == xMisisng && curY + 1 == yMissing){// for LL
Tromino
            tileName = "LL";
        }
        else if(curX + 1 == xMisisng && curY + 1 == yMissing){// for UL
Tromino
            tileName = "UL";
        }

    //print this tile to the console
    printf("(%-2d,%-2d) %s\n", tilePoint[0], tilePoint[1], tileName);
   return;

   }
   /***** Divide and conquer below here *****/
   //check for missing square, and if there is none, put one there
   ++trominoNum;
    int halfSize=curSize/2,xCenter,yCenter;
    int xUR ,yUR,  xUL ,yUL, xLR ,yLR,  xLL ,yLL;
    xCenter=curX+halfSize;
    yCenter=curY+halfSize;
    tilePoint[0] = xCenter;
    tilePoint[1] = yCenter;

    //TODO Add pringting tile here
    if(xMisisng <xCenter && yMissing< yCenter){ // checking that hole
in the first quad, if yes than put solve in center opposite quad.
        //printf("First qua\n");
        board[xCenter-1][yCenter]= trominoNum;
        board[xCenter][yCenter-1]= trominoNum;
        board[xCenter][yCenter]=trominoNum;

        xUL =xMisisng;
        yUL=yMissing;

        xUR =xCenter-1;
        yUR=yCenter;

        xLL =xCenter;
        yLL=yCenter-1;

        xLR =xCenter;
        yLR=yCenter;

        tileName = "LR";

    }
   else if(xMisisng <xCenter && yMissing>=yCenter){ // checking that
hole in the second quad, if yes than put solve in center opposite quad.
        //printf("Second qua\n");
        board[xCenter-1][yCenter-1]= trominoNum;
        board[xCenter][yCenter-1]= trominoNum;
        board[xCenter][yCenter]=trominoNum;
```



```
        xUL =xCenter-1;
        yUL=yCenter-1;

        xUR =xMisisng;
        yUR=yMissing;

        xLL =xCenter;
        yLL= yCenter-1;

        xLR  = xCenter ;
        yLR = yCenter;

        tileName = "LL";
    }
   else if(xMisisng >=xCenter && yMissing<yCenter){ // checking that hole in the third quad, if yes than put solve in center opposite quad.
        //printf("Third Qua\n");
        board[xCenter-1][yCenter-1]= trominoNum;
        board[xCenter-1][yCenter]= trominoNum;
        board[xCenter][yCenter]= trominoNum;

        xUL =xCenter-1;
       yUL=yCenter-1;

        xUR =xCenter-1;
       yUR=yCenter;

        xLL =xMisisng;
       yLL=yMissing;

        xLR =xCenter;
       yLR=yCenter;

        tileName = "UR";
    }
   else if(xMisisng >=xCenter && yMissing>=yCenter){ // checking that hole in the fourth quad, if yes than put solve in center opposite quad.
        //printf("Fourth Qua\n");
        board[xCenter-1][yCenter-1]= trominoNum;
        board[xCenter-1][yCenter]= trominoNum;
        board[xCenter][yCenter-1]= trominoNum;

        xUL =xCenter-1;
        yUL=yCenter-1;

        xUR =xCenter-1;
        yUR=yCenter;

        xLL =xCenter;
        yLL=yCenter-1;

        xLR =xMisisng ;
        yLR=yMissing;

        tileName = "UL";
    }

    //print non base case tile
    printf("(%-2d,%-2d) %s\n", tilePoint[0], tilePoint[1], tileName);

   //recurse over each quadrant
    solve(boardSize,xUL , yUL, curX,curY,halfSize,board); // recursive call to the first quadrant
    solve(boardSize,xUR , yUR, curX,curY+halfSize,halfSize,board); // recursive call to the second quadrant
    solve(boardSize,xLR , yLR, curX+halfSize,curY+halfSize,halfSize,board); // recursive call to the third quadrant
    solve(boardSize,xLL , yLL, curX+halfSize,curY,halfSize,board); // recursive call to the fourth quadrant
}

void printMatrix(int size, int (*board)[size]){ // printing the matrix
    int i,j;
    printf("\n* * * * * * * * * * * * * * * *\n");
    for (i = 0; i < size;i++){
        for ( j = 0; j < size;j++){

                if(board[i][j]==(-1)){
```



```
                    printf("%-2s ", "X");
                }
                else{
                    printf("%-2d ", board[i][j]);
                }

            }
        printf("\n");
    }
    printf("\n");

}
```

# Chapter Exercises

5.5) input: array a of length n, value val
output: true if indexes i, j exist such that a[i] + a[j] = v
and false if otherwise

```
mergesort(a)
  if (n == 1)
     return a

   var l1 = a[0] ... a[n/2]
   var l2 = a[n/2+1] ... a[n]

   return merge(l1, l2)

merge(a //array, b //array)
   var c // array

   while a and b not empty
     if a[0] > b[0]
       add b[0] to end of c
       remove b[0] from b
     else
       add a[0] to end of c
       remove a[0] from a

   while a not empty
     add a[0] to end of c
     remove a[0] from a
   while b not empty
     add b[0] to end of c
     remove b[0] from b

   return c
```

sorted array a from mergesort

```
find nums(sa, val)
  var i ← 0
  var j ← 0

  while (i < j)
    if sa[i] + sa[j] = va)
      return true
    else if sa[i] + sa[j] > va
      j--
    else
      i++

  return false // none fou
```

Analysis of above algorithm: The sorting of list a is $O(n\log n)$ which gives a sorted list a of which we can iterate over one more time to find if the indexes exist. This creates a total time complexity of $n\log n + n$ which, due to the rules of big-OH, simplify to $O(n\log n)$

↓

Dropping of constants and addition values

## Assignment 1

1s)

| | | | |
|---|---|---|---|
| ▨ | 2 | 3 | 3 |
| 2 | 2 | 1 | 3 |
| 5 | 1 | 1 | 4 |
| 5 | 5 | 4 | 4 |

2)

| | | | |
|---|---|---|---|
| 2 | 2 | 3 | 3 |
| 2 | 1 | 1 | 3 |
| 5 | 1 | 4 | ▨ |
| 5 | 5 | 4 | 4 |

3)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 3 | 4 | 4 | 8 | 8 | 9 | 9 |
| 3 | 2 | 2 | 4 | 8 | 7 | 7 | 9 |
| 6 | 2 | 5 | 5 | 11 | 11 | 7 | 10 |
| 6 | 6 | 5 | 1 | 1 | 11 | 10 | 10 |
| 18 | 18 | 19 | 1 | 13 | 13 | 14 | 14 |
| 18 | 17 | 19 | 19 | ▨ | 13 | 12 | 14 |
| 21 | 17 | 17 | 20 | 16 | 12 | 12 | 15 |
| 21 | 21 | 20 | 20 | 16 | 16 | 15 | 15 |

4)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 3 | 4 | 4 | 8 | 8 | 9 | 9 |
| 3 | 2 | 2 | 4 | 8 | 7 | 9 | ▨ |
| 6 | 2 | 5 | 5 | 11 | 7 | 7 | 10 |
| 6 | 6 | 5 | 1 | 11 | 11 | 10 | 10 |
| 18 | 18 | 19 | 1 | 1 | 13 | 14 | 14 |
| 18 | 17 | 19 | 19 | 13 | 13 | 12 | 14 |
| 21 | 17 | 17 | 20 | 16 | 12 | 12 | 15 |
| 21 | 21 | 20 | 20 | 16 | 16 | 15 | 15 |

5) $T(n) = \begin{cases} 1 & n=1 \\ 1 + 4T(\frac{n}{2}) & n>1 \end{cases}$

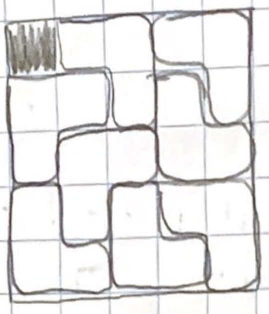
$\ell = 4$ $k = 2$ $f(n) = O(1)$

$\alpha = \log_2 4 = 2$

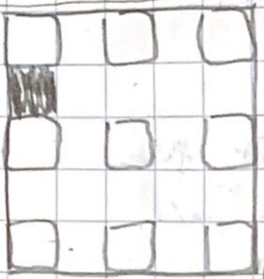
$f(n) \in O(n^{2-\epsilon})$ for $\epsilon > 0$ (ex $\epsilon = 1$)

Therefore $\Theta(n^2)$

6)



7)



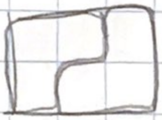
7 explanation)

▨ - removed tile
□ - highlighted tile

Any 5×5 board that does not have a highlighted tile removed cannot be tiled because there are 9 highlighted tiles but only 8 trominos ($^{25}/_3 = 8.33$ rounded down) may be placed on a 5×5 board making this board impossible to tile since it does not have a highlighted tile removed

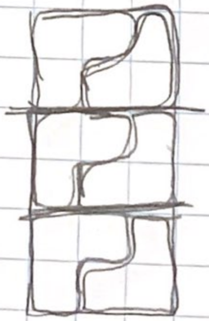
and, no tromino may cover 2 highlighted tiles,

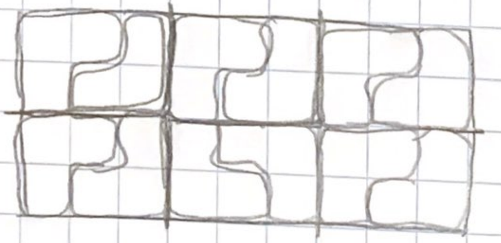
8) base case 2(1) × 3(1)



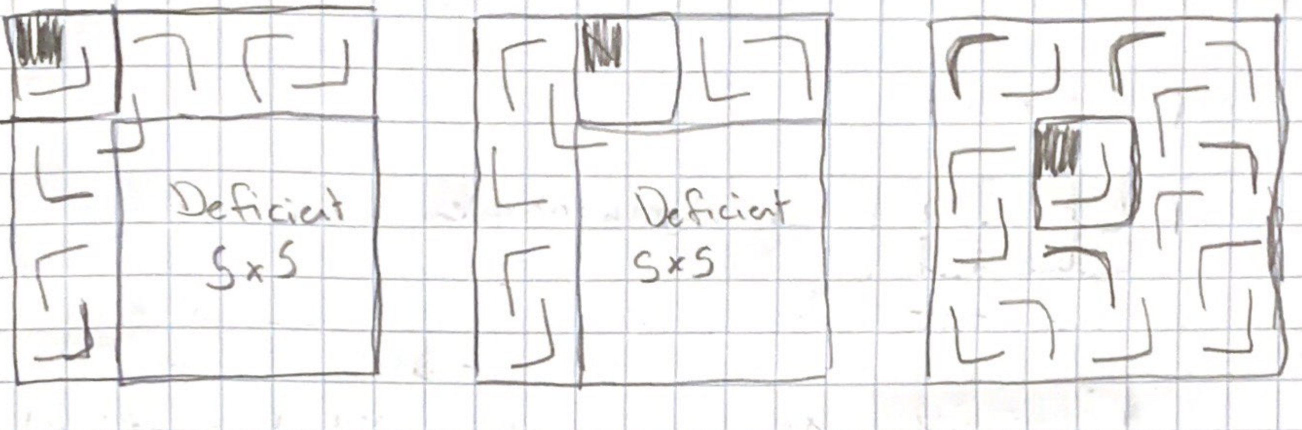
Any 2(i) × 3(j) board may be tiled first by isolating a 2×3 corner, then placing 2 trominos as shown by the base case, and then placing the base case complex adjacently in the x or y direction to fill board

eg 2(3) × 3(1)   2(2) × 3(3)



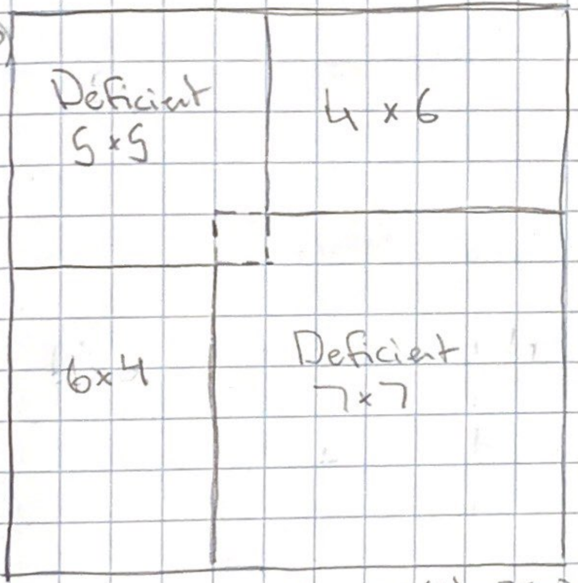


9)



Since you can remove any of the 4 of the 2×2 boxes highlighted in the diagrams above, and rotate all possible 7×7 difficat boards to have the deficient tile land in one of the highighted 2×2 boxes, any 7×7 deficint board may be tiled according to the images above.

10)



Any deficient 11×11 can be rotated so the deficient tile falls within the 7×7 region as indicated by the picture. Since this 7×7 area is now deficient it can be tiled by question 9 Since the 5×5 and the 7×7 region overlap, a completed 7×7 region creates a deficient 5×5 board with a corner removed, a solveable case by question 6 and 7. Completing this leaves 2 $2(2) \times 3(2)$ regions which since they are in the form $2i \times 3j$, can be solved by question 8