Assignment 5

Bhavya[U21CS100], Garvit[U21CS089], Swayam[U21CS079]

April 20, 2023

1 Tiling Problem

1.1 Question

Consider the following tiling problem. We have a courtyard with 2^n x 2^n squares and we need to tile the courtyard using L-shaped tiles or trominoes. Each trominoe consists of three square tiles attached to form an L shape as shown below.

Can this be done without spilling over the boundaries, breaking a tromino or having overlapping trominoes? The answer is no, simply because 2^n x $2^n = 2^{2n}$ is not divisible by 3, only by 2. However, if there is one square that can be left untiled, then 2^{2n} - 1 is divisible by 3. Can you show this?

Is there an algorithm that tiles any 2^n x 2^n courtyard with one missing square in an arbitrary location? As an example, below is a 2^3 x 2^3 where the missing square is marked "O". Does the location of the missing square matter?

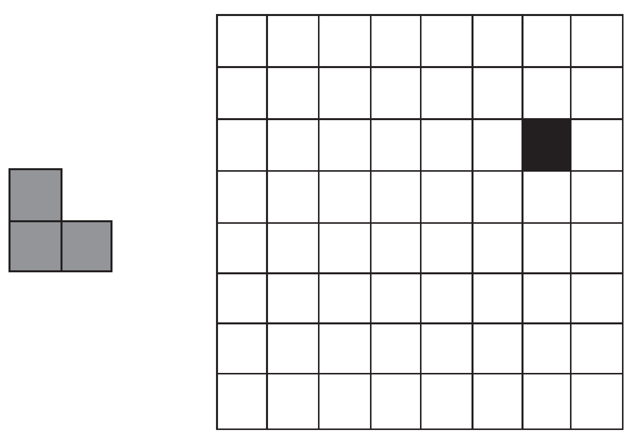


Figure 1: 8x8 Courtyard

1.2 Input Format:

Matrix Size - n
Position of the Statue - x y

2 Solution

2.1 Algorithm

1. Define the courtyard as a rectangular grid of cells.
2. Define the L-shaped tile as a set of 4 squares, each with a boolean value indicating whether the square is filled or not.
3. Generate all possible orientations of the L-shaped tile (there are 8 possible orientations, including rotations and reflections).
4. For each cell in the courtyard, attempt to place the L-shaped tile in each of its possible orientations, one at a time, checking if the tile fits and does not overlap with any other tiles already placed.
5. If a tile fits and does not overlap, mark the cells covered by the tile as filled.
6. Repeat step 4 and 5 for all remaining cells in the courtyard until the entire courtyard is covered or no more tiles can be placed.
7. If the entire courtyard is covered without any overlapping tiles, then the problem is solved. Otherwise, the problem has no solution.

2.2 Psuedo Code

Divide and Conquer Apporach

```
// grid is the given 2$^n$ x 2$^n$ grid initialised with all zero
// pos is the position of the statue on the grid given in the form of (x,y)
// N is the number to be placed as tile group

tiling(grid, x, y, N, statue.x, statue.y){
  pos(x, y)
  posvector[][]
  if (N != 0){
    m = pow(2, N-1)
    if (statue.x >= m and statue.y < m){
      pos2 = tiling(grid, x, y, n-1, m - 1, m - 1)
      pos = tiling(grid, x + m, y, n-1, statue.x- m, statue.y)
      pos3 = tiling(grid, x, y + m, n-1, m - 1, 0)
      pos4 = tiling(grid, x + m, y + m, n-1, 0, 0)
      posvector.push_back(pos2)
      posvector.push_back(pos3)
      posvector.push_back(pos4)
    }
    else if (statue.x < m and statue.y < m){
        position pos4 = tiling(grid, x + m, y + m, n-1, 0, 0)
        position pos3 = tiling(grid, x, y + m, n-1, m - 1, 0)
        position pos1 = tiling(grid, x + m, y, n-1, 0, m - 1)
        pos = tiling(grid, x, y, n-1, statue.x, statue.y)
        posvector.push_back(pos3)
        posvector.push_back(pos4)
        posvector.push_back(pos1)
    }
    else if(statue.x < m and statue.y >=m){
        pos2 = tiling(grid, x, y, n-1, m - 1, m - 1)
        pos1 = tiling(grid, x + m, y, n-1, 0, m - 1)
        pos = tiling(grid, x, y + m, n-1, statue.x, statue.y-m)
        position pos4 = tiling(x + m, y + m, n-1, 0, 0)
        posvector.push_back(pos2)
        posvector.push_back(pos4)
        posvector.push_back(pos1)
    }
    else if(statue.x >= m and statue.y >= m){
        pos2 = tiling(grid, x, y, n-1, m - 1, m - 1);
        pos3 = tiling(grid, x, y + m, n-1, m - 1, 0);
        pos1 = tiling(grid, x + m, y, n-1, 0, m - 1);
        pos = tiling(grid, x + m, y + m, n-1, statue.x - m, statue.y - m);
        posvector.push_back(pos2);
        posvector.push_back(pos3);
        posvector.push_back(pos1);
    }

    tilenum = tilenum + 1;
    for i = 0 till i = 3
    {
      matrix[(posvector[i]).y][(posvector[i]).x] = tilenum;
    }
  }
return pos;

}
```

2.3 C++ Code

```
#include<iostream>
#include<cmath>
#include<vector>
#include <time.h>

int matrix[8][8] = {{0},{0}};
int tilenum = 0;
class position{
    public:
    int x;
    int y;

    position(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
};

position tiling(int x, int y, int n, int b1, int b2)
{
    position pos(x, y);
    std::vector<position> posvector;
    if(n!=0)
    {
        int m = pow(2,n-1);
        if((b1>=(m)) && (b2<(m)))//Bill is in quadrant 1
        {
            position pos2 = tiling(x, y, n-1, m - 1, m - 1);
            pos = tiling(x + m, y, n-1, b1- m, b2);
            position pos3 = tiling(x, y + m, n-1, m - 1, 0);
            position pos4 = tiling(x + m, y + m, n-1, 0, 0);
            posvector.push_back(pos2);
            posvector.push_back(pos3);
            posvector.push_back(pos4);

        }else if((b1<(m)) && (b2<(m)))//Bill is in quadrant 2
        {
            position pos4 = tiling(x + m, y + m, n-1, 0, 0);
            position pos3 = tiling(x, y + m, n-1, m - 1, 0);
            position pos1 = tiling(x + m, y, n-1, 0, m - 1);
            pos = tiling(x, y, n-1, b1, b2);
            posvector.push_back(pos3);
            posvector.push_back(pos4);
            posvector.push_back(pos1);
        }else if((b1<(m)) && (b2>=(m)))
        {
            position pos2 = tiling(x, y, n-1, m - 1, m - 1);
            position pos1 = tiling(x + m, y, n-1, 0, m - 1);
            pos = tiling(x, y + m, n-1, b1, b2-(m));
            position pos4 = tiling(x + m, y + m, n-1, 0, 0);
            posvector.push_back(pos2);
            posvector.push_back(pos4);
            posvector.push_back(pos1);
        }else if((b1>=(m)) && (b2>=(m)))
        {
            position pos2 = tiling(x, y, n-1, m - 1, m - 1);
            position pos3 = tiling(x, y + m, n-1, m - 1, 0);
            position pos1 = tiling(x + m, y, n-1, 0, m - 1);
            pos = tiling(x + m, y + m, n-1, b1 - m, b2 - m);
```

```
            posvector.push_back(pos2);
            posvector.push_back(pos3);
            posvector.push_back(pos1);
        }

        tilenum++;
        for(int i=0; i<3; i++)
        {
            matrix[(posvector[i]).y][(posvector[i]).x] = tilenum;
        }
    }
    return pos;
}

int main()
{
  int n;
  int x,y;
  std::cout<<"Enter the value of n:"<<std::endl;
  std::cin >>n;
  std::cout<<"Enter the coordinates of statue:"<<std::endl;
  std::cin>>x>>y;
  int r = pow(2,n);
  position posBill = tiling(0, 0, n, x, y);
  matrix[(posBill).y][(posBill).x] = 100;
  for(int i=0; i<r; i++)
  {
      for(int j=0; j<r; j++)
      {
          std::cout<<matrix[i][j]<<'\t';
      }
      std::cout<<"\n\n\n";
  }
}
```

3 Time Complexity Analysis

3.1 Recurrence Relation and Tree

3.1.1 Recurrence Relation

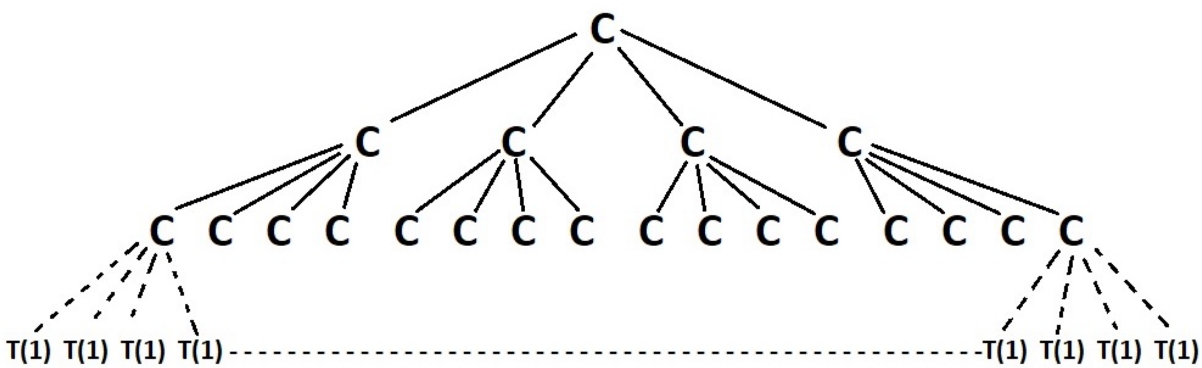
Since in each recursive call, we are dividing the matrix into 4 Quadrants and the cost for combining the subproblems is constant the recurrence relation will be given by the following equation -

$$T(n) = 4 * T(n/4) + \theta(1)$$

Assumption: n is the no. of elements in the matrix.

3.1.2 Recurrence Tree

Using the above recurrence relation we can depict the following recurrence tree -



Levels	Cost
0	c
1	$4^1 * c$
2	$4^2 * c$
.	.
.	.
.	.
h	$4^h * c$

- Height of Tree $h = log_4(n)$
- Number of Nodes at level-h = $4^{log_4(n)} = n$
- Time Complexity = $T(n) = \Theta(n)$

3.2 Subsitution Method

3.2.1 Inductive Hypothesis:

From the Recursion tree shown above, we can guess the solution this problem to be : $\Theta(n)$ i.e $T(n) = cn$.

Proving the above hypothesis using Strong Induction:

3.2.2 Base Case:

For $n = 1$,

$T(n) = cn = c$

also, $T(n) = 4 * T(n/4) + c$

$T(n) = 4 * T(1/4) + c = c$

Hence the hypothesis holds for the base case.

3.2.3 Recursive Case:

Let us assume the guess holds for all m $(1, n)$ union Z.

In that case, $T(n) = 4 * T(n/4) + c$,

here we know, $1 < n/4 < n$, thus the above assumption holds for $n/4$

So, $T(n) = 4 * c * (n/4) + c$

$T(n) = cn + c$

Consider the modified Hypotheis as $T(n) = cn - d$

thus, $C - 4d < 0$

Therefore, T(n) = cn = $\Theta(n)$

3.3 Master Theorem

$T(n) = 4 * T(n/4) + c$

$T(n) = 4 * T(n/4)\ \Theta(1) = a * T(n/4) + f(n)$

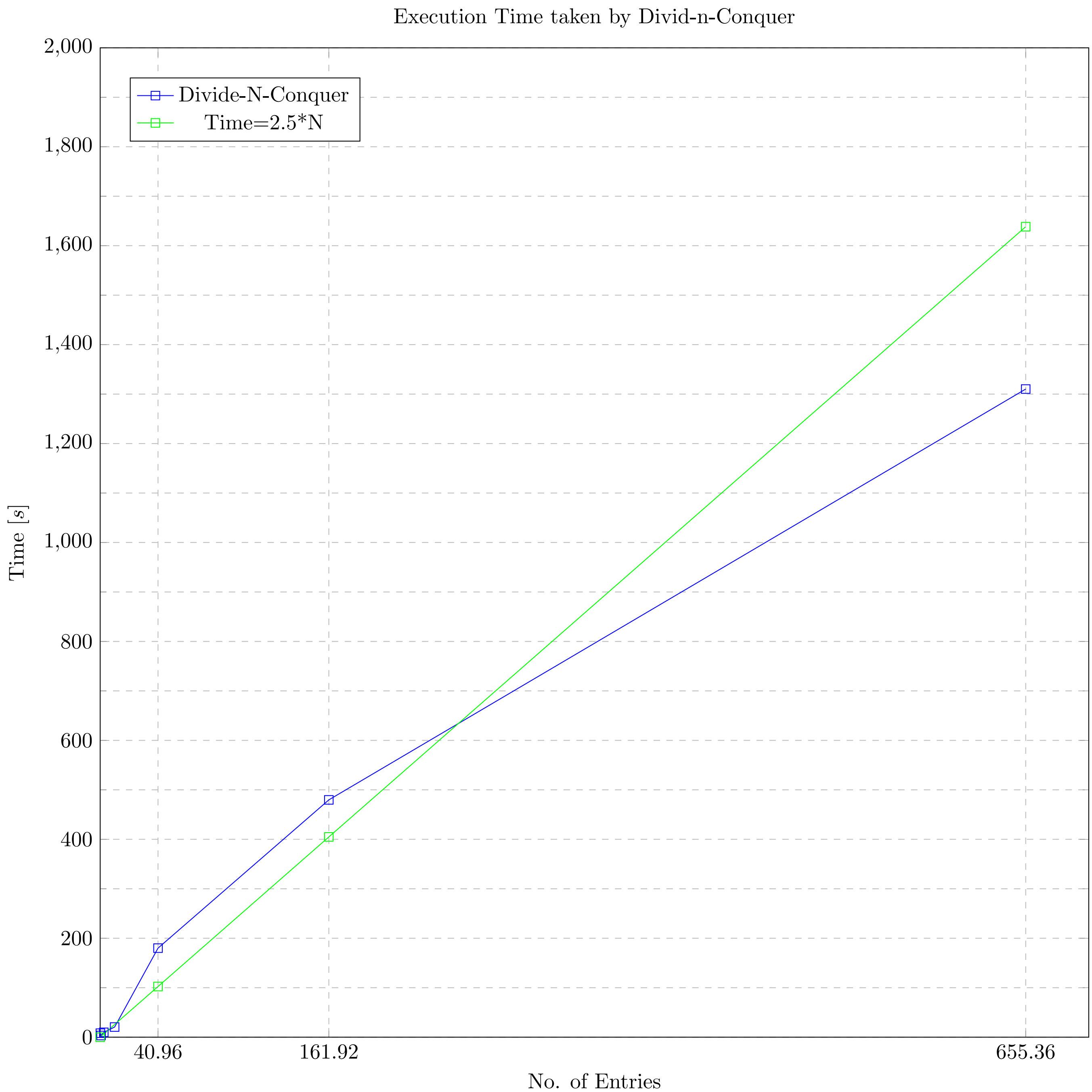
Here, $log_b(a) = log_4(4) = 1$

$$n^{log_b(a)} = n$$

Since f(n) = $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$

Therefore, $T(n) = O(n)$ by the virtue of the Master Theorem

4 Graphs



Execution Time taken by Divid-n-Conquer
Divide-N-Conquer
Time=2.5*N
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
Time [s]
40.96
161.92
655.36
No. of Entries