Project 1 Report ASA 2022/2023

Group: AL138

Student(s): Antonio Oliveira (104010) e Anees Asghar (107328)

Problem Description and Solution

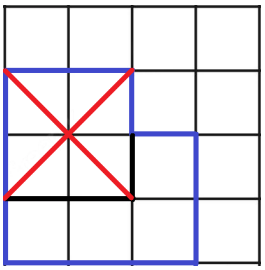
We have been asked to find the total number of distinct configurations in which the given area can be tiled using square tiles. We can refer to this area as a staircase. The proposed algorithm consists of the following:

We initialise our cursor on the first row (top-right corner of the staircase). Here, we may have multiple tile size choices that fit. For each tile-size choice available, we place a tile at that cursor position (i.e., remove the area from the staircase that that tile spans) and call the algorithm on the leftover staircase e.g. make a recursive call.

We realise that sometimes removing this tile may break our staircase structure[1]. This implies that we have to remove some tiles from the lower rows of the staircase to be able to safely remove this tile. In such cases, we push information for this tile removal on a stack, move the cursor to one of the lower rows and make a recursive call to our algorithm passing the backtrace stack as a parameter.

When our algorithm is run with a non-empty backtrace stack, after a tile is removed, we go down the stack and if the conditions for the earlier tile removals have now been met, we perform those tile removals. We call this procedure "backtracing".

In order to improve efficiency, our algorithm stores the solutions for the staircase structure to save on computing time. It implements a cache system which, in essence, is an unordered_map to pair a hash of a grid and its number of permutations.



Example[1] of removing 2x2 that would break the staircase structure. We have to hold off on the removal until the adjacent 1x1 is removed.

Project 1 Report ASA 2022/2023

Group: AL138

Student(s): Antonio Oliveira (104010) e Anees Asghar (107328)

Theoretical Analysis

```
Function solve(grid, backtrace):
    clean(grid)

    If(Len(gird)<=1) then
        Return 1
    Endif

    If(grid in cache) then
        return cache[grid]
    Endif

    permutations = 0

    For(i=1;i<=grid.maxFittingSquare;i++) do
        grid.tile(i)
    Endfor

    return permutations
endfunction
```

```
subprocedure tile(i):
    If(Not grid.canRemoveTile(i)) then
        backtrace.push(i, grid.cursor)
        grid.setCursorToNextBiggerLine()
        permutations += solve(grid, backtrace)
        Continue
    Endif

    grid.removeTile(i)

    If(Len(backtrace)==0) then
        permutations += solve(grid, backtrace)
        Continue
    Endif

    While(Len(backtrace!=0)) do
        If(grid.canRemoveTile(backtrace.top())) then
            grid.removeTile(backtrace.pop())
            permutations += solve(grid, backtrace)
        Else
            grid.setCursorToNextBiggerLine()
            permutations += solve(grid, backtrace)
        Endif
    Endwhile
endsub
```

- Reading input has a time complexity of O(n) where n is the number of rows.
- Running the solve function has an approximate time complexity of $O(2^n)$ where n is the area of the grid.
- Printing the result has a time complexity of O(1).
- Global time complexity is $O(2^n)$.

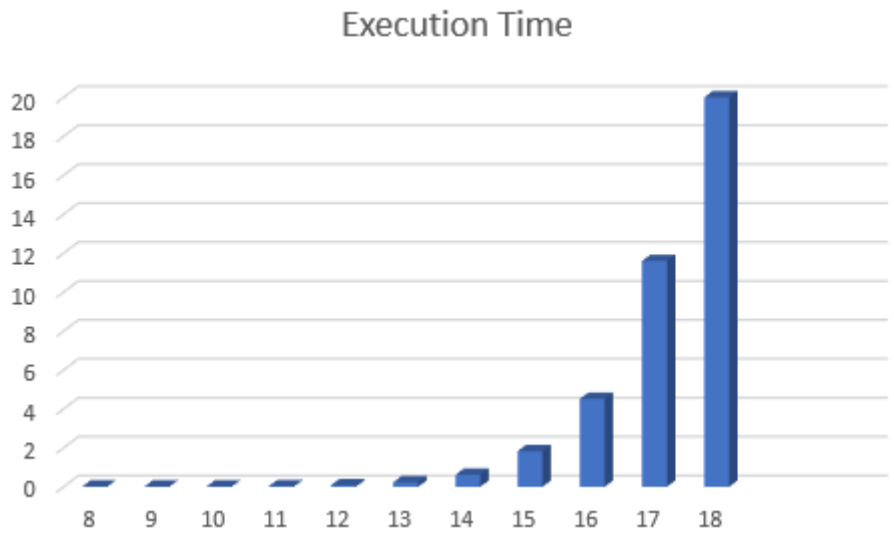
Project 1 Report ASA 2022/2023

Group: AL138

Student(s): Antonio Oliveira (104010) e Anees Asghar (107328)

Experimental Evaluation of Results

We test our algorithm on grids of sizes 8x8 to 18x18, where the area to tile is the complete grid. We are met with the following results:



As expected, with the increase in size of the grid, the area to tile increases polynomially and therefore we also see an exponential rise in execution time.

We also observe that for a grid of size 14x14, our algorithm takes approximately 14 ms to compute the result which stands in contrast to the 2.8 hrs we observed earlier for a smaller grid without making the use of memoization.

References

Hash function: A good hash function for a vector - c++