Projeto 1 ASA

93075 Gonçalo Azevedo   99205 Diogo Vieira

December 14, 2022

1 Descrição do problema

Para uma configuração da grelha, observamos que o número de combinações distintas de serem formadas é igual à soma do número de configurações distintas de todas as grelhas que resultam da adição de um bloco à configuração inicial. A solução é assim facilmente implementável com uma **recursão** onde, para cada grelha, é chamada recursivamente a função de cálculo com configurações que resultam da adição de blocos à configuração inicial, uma chamada recursiva por cada bloco passível de ser adicionado à grelha inicial (1x1, 2x2, 3x3, etc). Na recursão o **caso base** representa uma configuração sem quaisquer blocos livres, ou seja, um ramo na árvore recursiva que obteve uma configuração com sucesso (totalmente preenchida).

Para o efeito, representámos internamente as configurações das grelhas com matrizes, mais especificamente com vectores de vectores de shorts em C++, onde o valor de 1 simboliza uma posição livre na grelha e o valor de 0 simboliza uma posição ocupada ou inválida. Em cada nível da recursão é calculado o primeiro quadrado livre da matriz, considerando a ordem da esquerda para a direita e de cima para baixo. É nessa posição livre que será feita a tentativa de adição de blocos de diferentes dimensões (todas as dimensões até m ou n, conforme o menor) para a chamada recursiva. Todas as dimensões de blocos são adicionadas e para cada nova grelha resultante é feita a chamada recursiva, se dessa adição de blocos uma posição ocupada ou inválida é utilizada, a chamada recursiva retorna 0, simbolizando uma configuração inválida. O retorno de 1 de uma chamada corresponde a uma matriz sem posições livres, isto é, apenas composta por zeros, o que na árvore recursiva simboliza uma folha onde uma configuração válida foi encontrada.

2 Análise Teórica

- **Input**: A leitura do input é caracterizada por um simples loop que é executado n vezes para ler os valores $c1, c2, c3...c_n$ correspondentes às posições livres das linhas da matriz, logo, temos $\Theta(n)$.

```
Let C be a new array
for i ← 1 to n do
    read into C[i]
end for
```

- **Construção da matriz interna**: Após a leitura dos dados, é contruida uma matriz que representa a grelha internamente, para cada linha da matriz, as entradas são inicializados a 1 para todos os valores até c_n sendo este processo $O(n \times m)$ para o caso em que todos os valores $c_1 = c_2 = c_3 = ... = c_n = m$.

```
Let M be a new array of arrays
for i ← 1 to n do
    for j ← 1 to C[i] do
     M[i][j] ← 1
    end for
end for
```

- **Posições livres**: Para cada chamada recursiva é necessário calcular a primeira posição livre da matriz onde serão inseridos os novos blocos, esta operação é $O(n \times m)$ para o caso em que a posição livre corresponde à última posição da matriz e também no caso em que $c_1 = c_2 = c_3 = ... = c_n = m$

```
for i ← 1 to n do
    for j ← 1 to C[i] do
        if M[i][j] == 1 then
            free_x ← i, free_y ← j
            exit                                    ▷ Break both loops
        end if
```

end for
end for

- **Adição de blocos à configuração**: Em cada recursão, para a configuração atual, é feita a adição de blocos utilizando a função addToGrid que recebe como argumento a dimensão do bloco a adicionar. Esta dimensão será de 1 a n ou m, conforme o menor. Para cada valor i correspondente à dimensão do bloco a adicionar, a função tem uma complexidade O(i²) que corresponde ao tempo que a função leva a colocar as i posições da matriz a 0 ou -1 para casos inválidos

  min
  if $n < m$ **then**
  $min \leftarrow n$
  else
  $min \leftarrow m$
  end if
  for $i \leftarrow 1$ to min **do**
  addToGrid(i)
  end for

  No total, este processo apresenta uma complexidade $\sum_{i=1}^{min}(i^2) \in O(min^3)$

- **Chamada recursiva**: Finalmente, é feita a chamada recursiva para cada nova configuração, isto é, todas as configurações com blocos adicionados. Para cada chamada recursiva, tal como no cálculo das novas matrizes, é chamada n ou m vezes, conforme o menor dos valores (um para cada nova matriz calculada). Assim, concluindo, uma possível fórmula para a recorrência apresentada poderá ser

  Let k be equals to the sum of all c_1, $c2$, c_3, ..., c_n values, k is invalid if any c_x value is 0 but the given matrix has a non zero value on the x line

$$\text{Given } f(n) = \begin{cases} O(n^3) + O(m \times n), & \text{if } n < m \\ O(m^3) + O(m \times n), & \text{if } m < n \end{cases}$$

$$\text{We have } T(k) = \begin{cases} 1, & \text{if k = 0} \\ 0, & \text{if k is invalid} \\ \sum_i^n (T(k - i^2)) + f(n), & \text{if } n < m \\ \sum_i^m (T(k - i^2)) + f(n), & ifm < n \end{cases}$$

Tendo em conta esta recorrência não nos é óbvio que tipo de complexidade assimptótica poderá ter o algoritmo sem recorrer ao uso de métodos como desenho da árvore recursiva ou substituição. Um desenho da árvore recursiva dá a entender que a complexidade assintóptica será exponencial nos valores de n ou m (conforme o menor) que em seguida ficam condicionados aos valores de $c_x : x \in \{1...m\}$. Sabemos que com a utilização de **memoization** e pela ordem de resolução das chamadas recursivas , muitos valores $T(k)$ serão repetidos e haverá um corte brutal no número de cálculos, apenas havendo um acréscimo de $O(logk)$ como contrapartida para cada chamada recursiva, correspondente ao lookup (e possível inserção) numa **Red Black Tree**. O valor de k inválido é um resultado muito comum uma vez que são poucas as chamadas recursivas que resultam em matrizes válidas, salvo raras exceções em que $c_1 = c_2 = ...$ $= m$, sendo esta variação dependente dos valores $c_x : x \in \{1...m\}$. Assim, é de prever um algoritmo exponencial que cresce com o valor de $n + m$ e que é por sua vez também dependente dos valores de c_x.

3 Avaliação experimental dos resultados

Foram geradas 28 instâncias, com valores incrementais para a soma $n + m$, sendo variados os valores de n e m, por exemplo, por vezes utilizamos um valor de n fosse menor em poucas unidades que m, por vezes o contrário e por vezes foram utilizados valores iguais. Foi possível observar uma tendência exponencial com o crescimento de $n + m$, apesar de, em algumas experimentações não representadas, a variação para valores iguais de $n + m$ fosse significativa, resultado de valores diferentes de c_1, c_2, ..., c_n isto é k, ainda assim não escapando à tendência exponencial. O crescimento de $n + m$ parece prevalecer no que toca ao aumento exponencial do running time do algoritmo. Assim, podemos concluir que o algoritmo tem um complexidade assintóptica exponencial para o valor de input $n + m$.

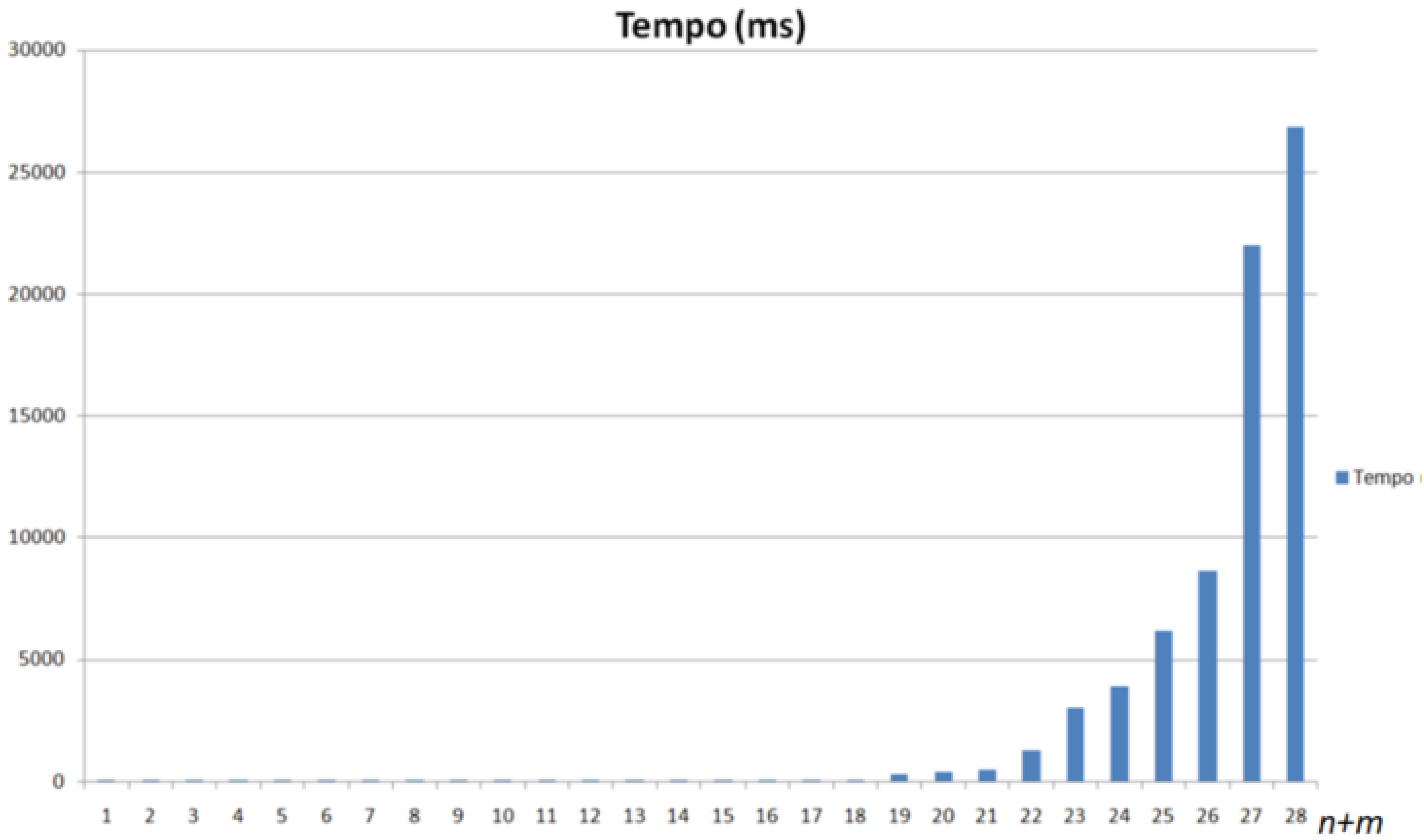

Figure 1: Algorithm running time variation with the sum of input values $n + m$