Unidade Curricular de Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

Mosaics

Grupo:
g124

Discentes:
Helder Godinho nº42741
Guilherme Grilo nº48921

março 2022

Docente:
Vasco Pedro

Índice

Introdução

O presente trabalho consiste na criação de um algoritmo que tem como objetivo calcular todas as maneiras possíveis para construir um determinado mosaico usando vários blocos de tamanhos diferentes, recorrendo para isso à programação dinâmica. A programação dinâmica é um método utilizado para a construção de algoritmos de forma a optimizar a resolução de problemas computacionais. Resumidamente, podemos utilizar a programação dinâmica em problemas cuja solução possa ser obtida a partir de soluções previamente calculadas, otimizando assim o tempo de compilação do próprio algoritmo pois cálculos repetido/inúteis são evitados.

Algoritmo

A função getSequence() obtém as sequências para cada cor e sempre que termina de calcular uma sequência, esta chama a função combinations(), sendo que esta última é que contém o algoritmo utilizado no nosso programa.

Na função combinations() começamos por inicializar um *array*, de nome table[], com tamanho que vai de 0 até à sequência para a qual queremos obter todas as combinações. De seguida colocamos o valor 1 na primeiro posição do *array*, pois este será o caso base do nosso algoritmo. Posteriormente utilizamos dois *for's* aninhados, sendo que o primeiro serve para percorrer o *array* table[] e o segundo para percorrer o *array* que contém os diferentes tipos de blocos. Já dentro do *for* mais interno calculamos todas as combinações das sequências antes da sequência que queremos obter, pois, através da programação dinâmica, é possível obter o valor das combinações de cada sequência utilizando para tal as combinações das sequências anteriores que já foram calculadas. Utilizamos ainda uma variável auxiliar que vai acumulando as combinações das sequências anteriores à sequência a calcular e no final coloca esse valor no *index* dessa mesma sequência. No fim retorna o valor com todas as combinações possíveis da sequência desejada.

Recursiva

$$comb(i) = \begin{cases} 1 & \text{se } i = 0 \\ \sum_{\forall j\,:\,b_j \leq i} \left\{ \; comb(i - b_j) \; \right\} & \text{se } i > 0 \end{cases} \tag{1}$$

i - representa a sequência de uma determinada cor cuja desejamos calcular todas as combinações possíveis com os diferentes blocos.

j - serve para percorrer o *array* que guarda os tamanhos dos blocos disponíveis.

O pseudocódigo:
combinations(I, J)

1. let table[0...I] be a new array //tabela para valores da função
2. a ← 0 //variável auxiliar
3. table[0] ← 1 //caso base
4. for i ← 1 to I do
5. for j ← 0 to J do
6. if b[j] ≤ i do
7. a ← a + table[i - b[j]]
8. table[i] ← a
9. a ← 0
10. return table[I]

Complexidade

Espacial

Inicialmente foi necessário inicializar um *array* que guardasse os diferentes tamanhos de blocos, ou seja, o espaço que este textitarray ocupa em memória será sempre um número linear pois depende do número de tamanhos diferentes que temos que guardar no mesmo, logo podemos definir a complexidade deste como sendo de O(k), onde k representa os diferentes tamanhos disponíveis.

Sabendo que, ao inicializarmos a matriz, necessitamos de saber o tamanho (linhas e colunas) que a mesma terá que possuir para suportar o mosaico, então podemos concluir que o espaço que iremos ocupar em memória será a multiplicação das linhas pelas colunas do próprio mosaico. Logo a complexidade espacial desta mesma matriz será de O(n*m) = O($\mathbf{n^2}$), onde n representa o número de linhas e m o número de colunas do mosaico.

Por fim, já na função combinations(), inicializamos um *array* que nos vai guardar todas as combinações possíveis de blocos para cada sequência antes da sequência que nós desejamos. Então a complexidade será, tal como no array de blocos, um número linear, O(x), onde x representa todas as combinações de cada sequência antes da que queremos determinar (inclusive).

Podemos finalmente concluir que a complexidade espacial final do nosso programa será:

$$O(k) + O(n^2) + O(x) = O(n^2) \tag{2}$$

k - array que guarda os tamanhos dos blocos.
$\mathbf{n^2}$ - a matriz que guarda o mosaico.
x - array que guarda as combinações de cada sequência.

De salientar ainda que a inicialização de variavéis não é considerada pois estas possuem um custo de O(1), logo, quando comparado com o custo final do programa, percebemos que é algo irrelevante para o mesmo.

Temporal

De forma a obter a sequência de cada cor no mosaico, utilizámos a função getSequence(), que consiste em dois *for's* aninhados: o textitfor exterior serve para percorrer todas as linhas da matriz do mosaico, logo na pior das hipóteses estamos perante uma complexidade O(n), sendo n o número de linhas do mosaico. Já o *for* interior serve para percorrermos até à penúltima coluna do mosaico, sendo por isso de complexidade O(m-1) = O(m) no pior dos casos, onde m é o número de colunas da matriz. Como estamos perante dois *for's* aninhados, podemos concluir que a complexidade dos mesmos será:

$$O(n) * O(m) = O(n * m) \tag{3}$$

Referir ainda que todas operações como declaração e inicialização de variáveis e *arrays*, alteração de valor nas próprias variavéis e *return's* são operações com custo O(1), logo não serão importantes de considerar na análise temporal do nosso algoritmo.

Na função combinations(), calculámos todas as combinações possíveis para cada sequência de cor, de acordo com os diferentes tamanhos de blocos disponíveis. Para esse efeito, utilizámos, tal como na função acima, dois *for's* aninhados: o *for* exterior serve para percorrer o *array* que possui todas as combinações diferentes para as todas as sequências de cor de 1 até à sequência desejada, sendo por isso de complexidade O(k), sendo k o tamanho do *array* que vai de 0 até à sequência que queremos obter que, no pior dos casos, significa que estamos a percorrer uma linha inteira da matriz do mosaico, logo podemos considerar que k é também o número de colunas do próprio mosaico; já o *for* interno serve para percorrer o *array* que possui todos os diferentes tamanhos de blocos que temos à disposição, logo este tem complexidade O(x), onde x é o tamanho do *array* dos blocos. Podemos concluir que a complexidade será:

$$O(k) * O(x) = O(k * x) \tag{4}$$

Devido ao facto de que, na função getSequence(), mais especificamente dentro do *for* interno, chamamos a função combinations() sempre que encontramos uma nova sequência de forma a calcular a obter todas as combinações possíveis de obter para essa mesma sequência, então a complexidade será a multiplicação das complexidades das duas funções anteriormente citadas,

pois, como chamamos a função combinations dentro do *for* mais interno da função getSequence(), então estamos a executar os dois *for's* da combinations() dentro dos dois *for's* da getSequence(), logo a complexidade temporal final do nosso algoritmo será:

$$O(n * m) * O(k * x) \tag{5}$$

Como no pior dos casos, k possui o mesmo significado que m (que é o número de colunas do mosaico), então no final teremos:

$$O(n * m^2 * x) \tag{6}$$

n - número de linhas do mosaico.
m - número de colunas do mosaico.
x - tamanho do *array* que possui os diferentes tamanhos dos blocos.

Comentários adicionais

Inicialmente a nossa maior dificuldade neste trabalho foi fazer a função que nos iria determinar todas as sequência de cores presentes no mosaico pois a presença de "."(que terminam a sequência e devem ser ignorados na cálculo final) subiu um pouco o grau de dificuldade deste trabalho mas rapidamente conseguimos ultrapassá-la.

Ao tentarmos submeter o nosso programa no *Mooshak*, notámos que o mesmo não passava nos testes do mesmo, pois só originava *Wrong Answers*. Após revermos o nosso código e o enunciado do trabalho algumas vezes, chegámos à conclusão que o facto de que o *output* final do programa poderia ser um número muito grande (maior que 32 bits), pois seria a multiplicação de todas as combinações de blocos possíveis de todas as sequências presentes no mosaico. De pois de muito pensarmos, chegámos à conclusão que teríamos de alterar o tipo de variável de algumas variáveis do tipo int (inteiro) para long, cuja diferença assenta no número de bits que cada uma suporta: enquanto que o tipo int suporta "apenas"números inteiros de até 32 bits, o long suporta números inteiros de até 64 bits. Ao realizarmos essas mesmas alterações, o nosso programa acabou por ser *Accepted* no *Mooshak*.

Conclusão

Neste trabalho foi-nos pedido que criássemos um programa que calculasse todas as diferentes formas de construir o mosaico pretendido utilizando blocos com tamanhos diferentes (informação dada no enunciado) e concluímos que conseguimos ultrapassar essa mesma tarefa com sucesso.

Apesar de algumas dificuldades durante a realização deste trabalho, fomos capazes de cumprir com todos os objetivos que nos foram propostos pois conseguimos que este passasse no *Mooshak* com sucesso.

Este trabalho foi ainda muito importante para a consolidação de conhecimentos ao nível desta unidade curricular, mais especificamente permitiu-nos perceber melhor o que é a programação dinâmica, como a implementar e também os benefícios que esta forma de abordagem traz a um programa.

Bibliografia

Pedro, Vasco in "Aulas de Estruturas de Dados e Algoritmos II". 2022 in Universidade de Évora.
Pedro, Vasco in eda2-t1-mosaics.pdf. Universidade de Évora's Moodle.