**Trabalho Prático 3 - Algoritimos 1**

**Pedro Henrique Alves Moutinho**

**Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil**
**pmoutinho@ufmg.br**

## 1. Introdução

O trabalho proposto consiste em otimizar o trabalho de colheitadeiras em um pomar respeitando as limitações das do tempo de colheita e disponibilidade de equipamento. Nesse sentido, são informados o número de fileiras do pomar e o número de macieiras em cada fileira e em seguida o programa recebe sequencialmente qual a quantidade de maçãs que cada macieira possui. Por fim o programa deve retornar qual a quantidade máxima de maçãs colhidas utilizando o processo de colheita otimizada e também qual foi o caminho percorrido para chegar a essa quantidade.

## 2. Implementação

O programa foi feito em C++, compilada pelo compilador G++ da GNU Compiler Collection.

### 2.1. Modelagem

Para representar o pomar foi utilizado o tipo vector recebendo um vector de inteiros, desse modo, cada posição representa uma macieira com a quantidade de maçãs que a mesma possui. Além disso, a mesma estrutura foi utilizada para gravar o resultado máximo de maçãs colhidas (resultados) e qual a melhor direção em que se pode chegar a fim de maximizar a colheita a partir de uma macieira qualquer **(melhoresCaminhos).**

### 2.2. Tratamento das Entradas

O primeiro passo da solução é receber o número de linhas (**numLinhas**) e colunas (**numMacieiras**) do pomar, após isso, é implementado um dois laços sendo um interno ao outro para ler quantas maçãs a macieira na posição[i][j] possui. Nesse sentido, a quantidade de maçãs de cada macieira é armazenada no vector **macieiras**.

### 2.3 - Otimizando o processo de colheita

Para encontrar o melhor caminho foi utilizado os conceitos de programação dinâmica bottom up. Desse modo, foi criada a função **calculaSomaPosicao** com o objetivo de calcular o máximo de maçãs a ser colhidas quando a última macieira a ser colhida é a que está na **posição[i][j]**. Desse modo, a função recebe os parâmetros relativos a posição e de forma recursiva procura se o melhor caminho para chegar naquela posição seria vindo pela diagonal anterior esquerda, pela posição diretamente anterior ou pela posição diagonal anterior direita. Dessa forma, esse processo é repetido até chegar a primeira fileira do pomar e quando isso acontece a função retorna a soma como a própria posição já que não existem mais fileiras a serem percorridas.

Nesse sentido, implementando os conceitos de programação dinâmica visando otimizar o processo sempre que o resultado para um posição é calculado o valor do mesmo é salvo no vector da seguinte maneira **resultados[i][j] = calculaSomaPosicao(i, j, …).** Além disso, o mesmo ocorre com o caminho utilizado para chegar nesse valor já que o mesmo é gravado em **melhoresCaminhos[i][j]** com a posição que otimizar o valor da colheita finalizando em **[i][j].** Desse modo, o programa executa efetivamente o cálculo no máximo uma vez para

cada posição, enquanto uma simples implementação recursiva executaria n vezes sendo n o número de chamadas para **calculaSomaPosicao(i,j, …).**

**2.4 - Saída**
Após alocar os resultados no vector **resultados** a última linha do mesmo é percorrida em busca de qual o caminho com maior resultado. Dessa forma, ao encontrar o pomar no qual a colheita é finalizada com o máximo de maçãs colhidas o programa grava a posição do mesmo. Por fim, o programa imprime o valor máximo e em seguida percorre o vector **melhoresCaminhos** a partir dessa posição informando qual foi o caminho percorrido para chegar aquele resultado.

**3 - Análise de Complexidade**
Todos os vectors utilizados ocupam um espaço de O(**numLinhas x numMacieiras)** já que cada posição dos mesmo representa o dado referente a uma macieira e o número de macieiras é dado por **numLinhas x numMacieiras.** Desse modo, a complexidade de espaço do algoritmo é de O(n x m).

Já em relação a complexidade de tempo temos um custo de O(n x m) para preencher as macieiras do sistema.

Além disso, caso o programa fosse implementado de maneira simplesmente recursiva o custo do mesmo seria de $O(n^3)$ no pior caso e no caso médio já que cada elemento n chamaria a função **calculaSomaPosicao(i,j, …)** e executaria o cálculo outras 3 vezes para saber se o melhor caminho seria esquerda, direita ou centro. Porém, com os conceitos de programação dinâmica foi possível construir o código de forma a otimizar esse custo já que no pior caso (árvores da ponta esquerda do pomar) o cálculo é executado de fato 2 vezes sendo uma para o elemento anterior e outra para o elemento na diagonal direita o que é compensado na outra ponta do pomar onde todos os cálculos já foram realizados. Já para os demais elementos do pomar efetuam o cálculo somente o elemento da direita.

Dessa forma, cada chamada da função **calculaSomaPosicao(i,j, …)** na média uma nova execução e como todas as árvores do pomar são percorridas a complexidade da mesma é de O(n x m).

Por fim, para impressão o custo é de O(n) pois é necessário somente imprimir o dado de uma coluna em cada linha do vector **melhroesCaminhos** além de imprimir o resultado.

Desse modo, a complexidade do programa é de O(n x m).

**4 - Conclusão**
Com o programa implementado foi possível perceber a importância da utilização da programação dinâmica e como o tempo de execução do algoritmo foi bem menor devido a isso já que o número de cálculos executados pelo mesmo foi significativamente menor do que em uma implementação simplesmente recursiva.