Trabalho Prático 3

Roberth A. Parreiras[1]

[1]Universidade Federal de Minas Gerais (UFMG)

roberthparreiras@ufmg.br

Resumo. *Esse trabalho prático tem como objetivo resolver o problema de encontrar a mesa de maior área que cabe em uma casa. Dessa forma, para se resolver isso, foi fornecido uma planta da casa e as possíveis mesas para caber nela, sendo escolhido a dimensão da maior mesa que cabe na maior área.*

1. Introdução

Com o avanço da tecnologia, nos dias atuais se tornou ainda mais fácil se conectar com as pessoas, mesmo elas estando distantes uma das outras, via encontros online.

Entretanto, minha vó prefere encontros presenciais com toda a família, e, após a pandemia, ela decidiu que quer ter uma mesa tão grande quanto possível para comportar o máximo de membros da família nesses encontros presenciais. Dessa forma, como ela desconfia de vários profissionais da área, ela decidiu me passar o desafio de encontrar essa mesa que ela tanto quer, visto ela considerar eu o neto mais esperto estudante de computação.

Assim, mesmo minha vó sendo contrariada, ela aceitou a planta da casa que a arquiteta forneceu, com os espaços disponíveis para a mesa, e escolheu junto com minha tia as possíveis mesas que gostou, sendo feita uma lista das mesmas.

Agora, com a planta e as possíveis mesas, tenho o trabalho de listar todos os espaços vazios da casa e verificar se existe alguma mesa da lista que caiba neles, sendo retornado o comprimento e a largura da maior mesa que cabe.

O trabalho foi dividido da seguinte forma: a seção 2 apresenta a modelagem em alto nível. A seção 3 apresenta o pseudo-código. A seção 4 apresenta a descrição da solução. A seção 5 apresenta a análise de complexidade. A seção 6 apresenta a demonstração de porque o algoritmo funciona. A seção 7 apresenta a conclusão. Por fim, a seção 8 apresenta as instruções de compilação.

2. Modelagem

Para resolver o problema, foi-se utilizado a lógica de resolução do *Problema de Máximo Retângulo Vazio*, sendo transformado os valores da planta do arquivo de entrada em binário: ”#” transformado em 0 e ”.” transformado em 1.

Sendo *[i, j]* a posição linha e coluna da planta, o *Problema de Máximo Retângulo Vazio* consiste em encontrar o maior retângulo em uma *matriz* de 0 e 1, sendo 1 a área vazia da *matriz*. Dessa forma, se a posição da *matriz [i, j]* for diferente de 0, então soma *matriz [i, j]* com *matriz [i - 1, j]*, até serem somadas todas as linhas, encontrando o valor máximo de cada posição da *matriz*. Assim, obterá uma lista com o a área, comprimento e largura de todas as linhas.

Logo após encontrar essa lista, checa quais mesas cabem em cada retângulo, retornando o comprimento e largura da maior mesa que cabe nas dimensões do mesmo.

3. Pseudo-código

3.1. Pseudo-código de MaxAreaPorLinha

```
INICIO_FUNCAO
MaxAreaPorLinha(matriz, inteiro posLinha, inteiro numColuna)
Cria pilha posicao;

WHILE(coluna < numColuna):

    IF(posicao.vazio ou matriz[posLinha][posicao.topo] <=
       planta[posLinha][coluna]):
        posicao.insere(coluna);     ++coluna;
    END_IF

    ELSE: (1)
        topo = posicao.topo;   posicao.removeTopo;

        IF(posicao.vazio):
            areaDoTopo = matriz[posLinha][topo] * coluna;
        END_IF

        ELSE:
            areaDoTopo = matriz[posLinhas][topo] *
            (coluna - posicao.topo - 1);
        END_ELSE

        IF(areaMaxima < areaTopo):
            areaMaxima = areaTopo;
        END_IF
    END_ELSE
END_WHILE

WHILE(!posicao.vazio):
    faz (1)
END_WHILE

RETORNA areaMaxima;

END_FUNCAO
```

3.2. Pseudo-código de AreaMaxima

```
INICIO_FUNCAO
AreaMaxima(matriz, inteiro linha, inteiro coluna)
```

```
FOR(i = 1; enquanto i < linha; ++i):
    FOR(j = 1; enquanto j < coluna; ++j):
        IF(matriz[i][j] positivo):
            matriz[i][j] += matriz[i - 1][j];
        END_IF
    END_FOR

    area = MaxAreaPorLinha(matriz, i, coluna)
END_FOR
END_FUNCAO
```

4. Descrição da solução

Para resolver o problema, primeiro encontra a área máxima de cada linha para depois encontrar qual as dimensões da melhor mesa que se encaixa na área.

Assim, para encontrar a área máxima de cada linha, cria-se uma *pilha* que vai armazenar o valor da coluna(vamos chamar de *matriz [i, j]*), sendo esse valor acrescentado por 1 a cada iteração do *while*, (apenas quando o valor da posição *matriz [i, j - 1]* <= *matriz [i, j]*). Caso contrário, então seleciona o valor que está no topo da *pilha*(vou chamar de val), desempilha ele e multiplica a posição *matriz[i, val]* com (*j - (novo topo da pilha) - 1*), obtendo o resultado que vou chamar de *areaAtual*. Assim, checa se *areaTotal* é maior que a área que já está armazenada. Já o segundo *while* entra se a *pilha* estiver vazia, e caso esteja, faz o mesmo do *else* do *while* anterior. Finalmente, retorna a área máxima e seu comprimento.

Agora, utiliza a função *AreaMaxima* para calcular a maior área da *matriz*. Nesse sentindo, entra no loop *for* e checa se a posição *matriz[i][j]* é diferente de zero. Caso seja, então *matriz[i][j]* += *matriz[i - 1][j]*. Logo após, chama a função *MaxAreaPorLinha* em cada linha, gerando uma lista com a área máxima de cada linha.

Após isso, utiliza um *MergeSort* para deixar a lista de retângulos e mesas em ordem decrescente em relação a área delas, para depois entrar na função *MaiorMesa* e checar qual o comprimento e largura da melhor mesa para a melhor área. Nesse sentido, podem ser rotacionada as mesas caso necessário para poder caber em alguma área.

5. Análise de complexidade

5.1. Complexidade de Tempo da função *MaxAreaPorLinha*

Inicialmente, cria-se uma *pilha* vazia para armazenar as iterações *j* na linha, sendo *j* a coluna da *matriz*. Após, entra em um loop *while*, que é iterado enquanto *j* for menor que o número total de colunas da *matriz*; e dentro dele, utiliza-se os métodos da *pilha*: **empty()**, **top()**, **push()** e **pop()**; sendo todos complexidade de tempo $O(\mathbf{1})$.

Assim, como ocorre o teste do primeiro *if* sendo: *matriz[i][pilha.***top()***]* <= *matriz[i][j]*, então se for verdadeiro, é empilhado cada iteração *j* na *pilha*. Dessa forma, caso contrário, entra no *else* que recebe o valor do topo e o desempilha da *pilha*. Portanto, o número de iterações será a quantidade de **push()** que for realizado no *if* mais a quantidade de **pop()** realizado no *else*, chegando a uma complexidade: **chamando *j* de C temos** $O(\mathbf{C} * \mathbf{push()}) + O(\mathbf{C} * \mathbf{pop()})$, que é igual a $O(\mathbf{2} * \mathbf{C})$. Portanto, a complexidade de tempo total é $O(\mathbf{C})$.

5.2. Complexidade de Espaço da função *MaxAreaPorLinha*

A complexidade de espaço total de *MaxAreaPorLinha* é (criação de espaços constantes + *pilha*) = O(**1 + C**), sendo dominado por O(**C**).

5.3. Complexidade de Tempo da função *AreaMaxima*

A função *AreaMaxima* contém um *for* dentro de um *for*, sendo o primeiro iterando de 1 até o número máximo de linhas e o segundo iterando de 1 até o número máximo de colunas, sendo linha e coluna da *matriz*. Para essa função, utiliza-se um conjunto de vetores que armazenam o valor da área, comprimento e largura, com cada um sendo de tamanho total de linhas O(**L**). Dessa forma, o segundo *for* executa a soma de *matriz[i, j]* com *matriz[i* - 1*, j]*, caso *matriz[i, j]* seja não nulo. Já o primeiro *for* executa a função *MaxAreaPorLinha*, de complexidade de tempo O(**C**), a cada iteração, sendo armazenados esses valores dentros dos vetores.

Portanto, a complexidade de tempo total é dada por
(primeiro *for* * *MaxAreaPorLinha*) = O(**L * C**).

5.4. Complexidade de Espaço da função *AreaMaxima*

A complexidade de espaço total de *AreaMaxima* é (criação de espaços constantes + primeiro vetor + (segundo vetor + terceiro vetor)) = O(**1 + (L * C) + (2 * L)**), sendo dominado por O(**L * C**).

5.5. Complexidade de Tempo da função *MaiorMesa*

Ela possui um loop *for* dentro de um outro loop *for*, sendo esse primeiro *for* iterado de 1 até número total de linhas da *matriz* e o segundo *for* é iterado o número total da lista de mesas. Chamando número total de linhas **L** e o número total de mesas **M**, a complexidade total de tempo da função é (linhas * mesas) = O(**L * M**).

5.6. Complexidade de Espaço da função *MaiorMesa*

Como ela possui dois loop *for* e eles iteram sobre a lista de mesas e o número total de linhas, a complexidade total de espaço é (criação de espaços constantes + lista de mesas + número total de linhas) = O(**1 + L + M**), sendo dominado por O(**L + M**).

5.7. Complexidade de Tempo da função *Main*

A função *Main* contém um primeiro *for* para receber via terminal todas as informações da *matriz*. Ele é executado para todas as linhas e colunas da *matriz* do arquivo de entrada, tendo a complexidade de tempo como O(**L * C**). Depois, há a criação de um conjunto de vetores que armazenam a área, comprimento e largura recebidas da função *AreaMaxima*, com complexidade de tempo O(**L * C**).

Após, há a criação de um conjunto de vetores que armazenam a área, comprimento e largura das mesas que são passadas pelo arquivo de entrada, entrando em um loop *for* que itera, chamando o número de mesas como M, O(**M**) vezes. Logo após, ordena usando um *MergeSort* O(**A * logA**), com **A** como área. Depois, chama a função *MaiorMesa* para encontrar a maior mesa, sendo ela O(**L * M**). Então a complexidade total de tempo é
(primeiro *for* + *AreaMaxima* + segundo *for* + *MergeSort* + *MaiorMesa*) =
O((**L * C**) + (**L * C**) + **M** + (**A * logA**) + (**L * M**)), dominado por
O((**L * C**) + (**A * logA**) + (**L * M**)).

5.8. Complexidade de Espaço da função *Main*

A complexidade total é (primeiro *for* + *AreaMaxima* + segundo *for* + *MergeSort* + *MaiorMesa*) = O((**L + C**) + (**L * C**) + **M** + **A** + (**L + M**)).

5.9. Complexidade Total do programa

A Complexidade de Tempo Total do programa é (*MaxAreaPorLinha* + *AreaMaxima* + *MaiorMesa* + main) =
O(**C** + (**L * C**) + (**L * M**) + ((**L * C**) + (**A * logA**) + (**L * M**))).

A Complexidade de Espaço Total do programa é (*MaxAreaPorLinha* + *AreaMaxima* + *MaiorMesa* + main) = O((**C** + (**L * C**) + (**L + M**) + (**L * C**)).

6. Demonstração

Def. OPT[*j*] = área máxima obtida na linha, em um intervalo 1, 2, ..., *j*, sendo *j* a posição de cada coluna no vetor linha.

Objetivo. OPT[*n*] = área máxima possível entre um conjunto de colunas *j*.

Caso 1. OPT[*j*] não seleciona uma área num intervalo de colunas *j*.
. Essa área não é a solução ótima para a linha, ou seja, não possui a maior área.

↕ propriedade da substrutura ótima(prova via troca de argumentos)

Caso 2. OPT[*j*] seleciona uma área num intervalo de colunas *j*.
. Recebe, se existir, o valor do topo da pilha P_j.
. Seleciona a solução ótima entre a área encontrada com a área máxima atual A.

Equação de Bellman.

$$OPT[\,j\,] = \begin{cases} 0, \;\; se \;\; j = 0 \\ max(OPT[j-1] * (j - P_{j-1} - 1), A), \;\; se \;\; P_j \;\; != \;\; vazio \\ max(OPT[j-1] * j, A), \;\; caso \;\; contrário \end{cases}$$

Utiliza essa equação para cada linha da *matriz*.

7. Conclusão

Esse trabalho prático tem como objetivo implementar um algoritmo que encontra a maior mesa que cabe na maior área. Nesse sentido, foi implementado um código que encontra essa mesa por meio da lógica de resolução de *Máximo Retângulo Vazio*, com uma complexidade de tempo e espaço aceitável.

8. Instrução de compilação

Para compilar o programa, deve-se digitar no terminal ”g++ -c main.cpp”.

Para criar um executável do programa, deve-se digitar no terminal ”g++ -o tp03.exe main.cpp”.