Trabalho Prático 2 - Algoritmos 1

Guilherme de Abreu Lima Buitrago Miranda - 2018054788

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

guilhermemiranda@dcc.ufmg.br

1. Introdução

O trabalho prático 2 da disciplina de Algoritmos 1 consiste em auxiliar Luiza e suas amigas a fazer a melhor viagem possível para o Arquipélago de San Blas, conjunto de 365 ilhas pertencentes ao Panamá e fortemente ameaçado pelo aquecimento global.

Devido ao fato de que o dinheiro disponível para a viagem é limitado, Luiza e suas amigas desejam aproveitá-lo da melhor maneira possível e, para tal, atribuíram para cada ilha uma pontuação, relativa à sua vontade de visitar o local.

Dessa forma, o algoritmo implementado deve dar às meninas duas opções de roteiro: uma com repetições (possivelmente ficando mais de um dia na mesma ilha) e o outro sem repetições. Para esse fim, foram implementadas duas funções principais para resolver os problemas, uma utilizando a abordagem gulosa e a outra, programação dinâmica.

2. Implementação

2.1. Entrada e saída de dados

A entrada esperada é feita por meio de um arquivo de texto, passado como argumento na execução do programa. Esse arquivo deve ter a seguinte estrutura: na primeira linha, dois inteiros N e M, indicando respectivamente o valor máximo a ser gasto na viagem seguido pela quantidade de ilhas.

As próximas M linhas também devem conter dois inteiros D e P representando o custo de passar um dia naquela ilha e a pontuação atribuída por Luiza e suas amigas para tal.

A saída gerada pelo programa é impressa no $stdout$ e tem sempre duas linhas: a primeira representa um roteiro onde as meninas podem ficar mais de um dia na mesma ilha, e a segunda um roteiro onde passam, no máximo, um dia em cada (solução gulosa e dinâmica, respectivamente). Assim, para cada uma das duas linhas, dois inteiros serão escritos: o primeiro se refere à pontuação daquele caso e o segundo à quantidade de dias de viagem.

2.2. Estruturas de dados

Para a modelagem do problema, foi empregado uma struct $Island$, que contém três atributos: custo por dia, pontuação e um fator (pontuação/custo por dia), que assume que não existe nenhuma ilha em que o custo de um dia seja gratuito.

Assim, para a implementação das soluções, foi criado um array de tamanho M, em que cada posição contém uma instância de $Island$, e o mesmo é passado tanto para o método que resolve utilizando o paradigma guloso, tanto para o método que utiliza programação dinâmica.

2.3. Algoritmo

Para a implementação das duas soluções, cada uma com seu respectivo paradigma de programação, a classe utilitária $tripUtils$ foi criada. Nela, além de um construtor privado (para impedir a criação de instâncias da mesma), são criados 5 métodos estáticos.

Os dois principais são o $runGreedy$ e o $runDynamic$, que se referem às duas soluções que o problema deve fornecer. Para a execução do $runGreedy$, outros dois métodos foram criados, o $merge$ e o $mergeSort$, que são responsáveis por ordenar as ilhas de acordo com seu fator pontos/custo diário, importante para a solução gulosa do problema.

Assim, logo no início da execução do algoritmo guloso, o merge sort é chamado, e o mesmo ordena as ilhas utilizando a técnica de divisão e conquista. Primeiramente o algoritmo calcula o ponto médio do array corrente, seguido pela resolução de dois subproblemas de forma recursiva, cada um com tamanho de metade do array anterior e, por fim, une ambas soluções em uma única. Dessa forma, após o fim da pilha de recursão, teremos o array ordenado utilizando como critério o fator supracitado.

Posteriormente a solução gulosa percorre o array ordenado de ilhas de forma decrescente, ou seja, da ilha de maior fator para a de menor, checando se o custo por dia é menor ou igual ao dinheiro em disponível no momento. Se sim, o algoritmo calcula quantos dias Luiza e suas amigas devem ficar naquela ilha, além de somar os pontos à pontuação final. No fim da execução, é retornado um par contendo a pontuação total e o número de dias que a viagem irá durar.

Já para a solução dinâmica, o único método extra implementado é o max, que retorna o valor máximo entre dois números, que será usado posteriormente. Assim, no início da solução dinâmica é criado uma matriz de $M + 1$ linhas, sendo M o número de ilhas e $N + 1$ colunas, sendo N o custo máximo da viagem.

Em seguida, são criados dois laços for, o primeiro percorrendo o número de ilhas e o segundo o custo máximo, que checa, primeiramente, se um dos índices é zero. Caso positivo, a matriz é preenchida com 0 naquela posição (com isso, a primeira linha e a primeira coluna da matriz são todos zerados).

Caso contrário, checa-se se o custo por dia da posição em questão do array de ilhas é menor ou igual ao índice do for mais interno. Se sim, é invocado o método max para checar o que é maior: o valor atual daquela linha mas na coluna anterior ou um novo valor, sendo este a soma da pontuação da posição corrente do array de ilhas e da matriz na linha anterior e na coluna do índice do for mais interno menos o custo por dia da posição corrente do array. Se não, a posição corrente será igual à posição imediatamente da coluna anterior.

Em outras palavras, o algoritmo verifica a melhor maneira para maximizar a pontuação atingida naquela posição da matriz, para que, no fim da execução dos laços, sua última posição (última linha e última coluna) tenham a maior pontuação possível de ser atingida.

Não obstante, como deseja-se também encontrar qual o número total de dias da viagem, é necessário fazer um backtracking na matriz de memorização. Para isso, inicia-se dois contadores num for, i e j, como o número de ilhas e o custo máximo, respecti-

vamente. Enquanto ambos forem maiores que 0, se a posição atual for diferente que a posição na coluna anterior, o contador do número de dias é incrementado e j é decrementado pelo custo por dias da ilha em questão. Além disso, ao fim de cada iteração, i é decrementado.

Dessa forma, pode-se observar, na matriz de memorização, quais foram as mudanças feitas e, portanto, qual o número de dias da viagem no fim do processamento das ilhas e de seus respectivos custos. Por conseguinte, é retornado o par contendo a pontuação total e o número de dias que a viagem irá durar.

Não obstante, no arquivo main.cpp há a implementação de um procedimento que lê o conteúdo do arquivo de texto passado como parâmetro e invoca os métodos em questão da classe $tripUtils$, fazendo a impressão da saída no $stdout$.

2.4. Instruções de compilação e execução

O compilador utilizado para o desenvolvimento do programa foi o g++/gcc $8.3.0$, usando a flag $-std = c$++11 numa máquina com a versão 4.19.49 do Kernel Linux. Para executar o programa, ao entrar em seu diretório, no terminal, deve ser digitado o comando $make$. Com isso, será gerado um binário $tp2$ e o algoritmo pode ser executado passando um arquivo de texto como argumento, por exemplo: $./tp2\ entrada.txt$

3. Análise de Complexidade

Para a análise de complexidade de tempo e de espaço do algoritmo, vamos analisar cada uma das duas soluções separadamente.

3.1. Solução gulosa

Para a solução gulosa, no que se refere à complexidade de tempo, tem-se que, durante a ordenação, o merge sort divide o array em dois, o que demora um tempo costante $O(1)$, recursivamente resolve dois problemas, cada um com tamanho $m/2$, o que tem complexidade $2T(m/2)$ e depois combina as soluções, o que tem complexidade $O(m)$. Resolvendo a equação de recorrência, em que $T(m) = O(1)$ se $m = 1$ e $T(m) = 2T(m/2) + O(m)$ se $m > 1$, tem-se que a complexidade de tempo do merge sort é de $O(mlogm)$.

O restante da solução apenas percorre todas as ilhas, o que tem complexidade $O(m)$. Dessa forma, tem-se que a complexidade de tempo da solução gulosa é de $O(mlogm) + O(m) = O(mlogm)$.

Além disso, no que se refere à complexidade de espaço, a única memória extra utilizada para a solução gulosa é o array que armazena as ilhas a serem ordenadas e, posteriormente, processadas. Dessa forma, sua complexidade de espaço será igual a $O(m)$

3.2. Solução dinâmica

Para a solução dinâmica, no que tange a complexidade de tempo, tem-se dois laços principais aninhados, o mais externo percorrendo os números de 0 até o total de ilhas e, o mais interno, de 0 até o custo máximo. Dessa forma, extrai-se a complexidade $O(m * n)$ no laço.

Posteriormente há, também, um laço para backtracking, que procura alterações nas linhas da matriz para contabilizar o número de dias da viagem. Como essa operação

percorre apenas as linhas em buscas de diferenças, tem-se que sua complexidade é de $O(m)$.

Assim sendo, pode-se concluir que a complexidade de tempo para a solução dinâmica é de $O(m * n)$.

Não obstante, para complexidade de espaço, é preciso considerar duas estruturas: o armazenamento das ilhas, que, conforme supracitado, ocupa $O(m)$ em memória e a matriz de memorização. Tal matriz tem número de linhas igual ao número de ilhas e número de colunas igual ao custo máximo, ou seja, ocupa $O(m * n)$ em memória.

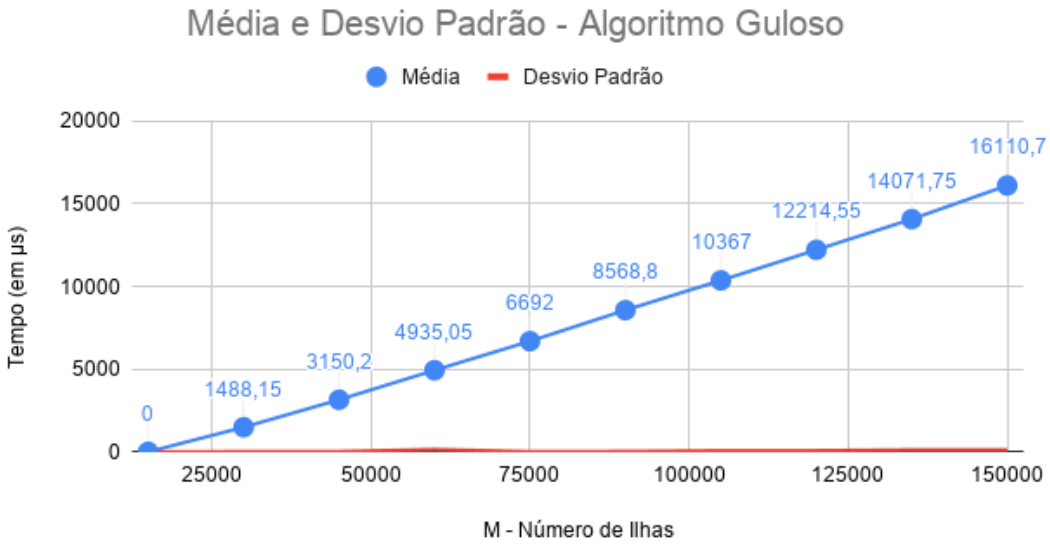
Dessa forma, tem-se que a complexidade de memória da solução dinâmica é de $O(m) + O(m * n) = O(m * n)$.

4. Avaliação Experimental

Para a análise experimental, foram gerados diversos casos de teste, com o número de ilhas M crescendo de 0 até 150000 e N mantido constante em 1500, para não afetar os resultados.

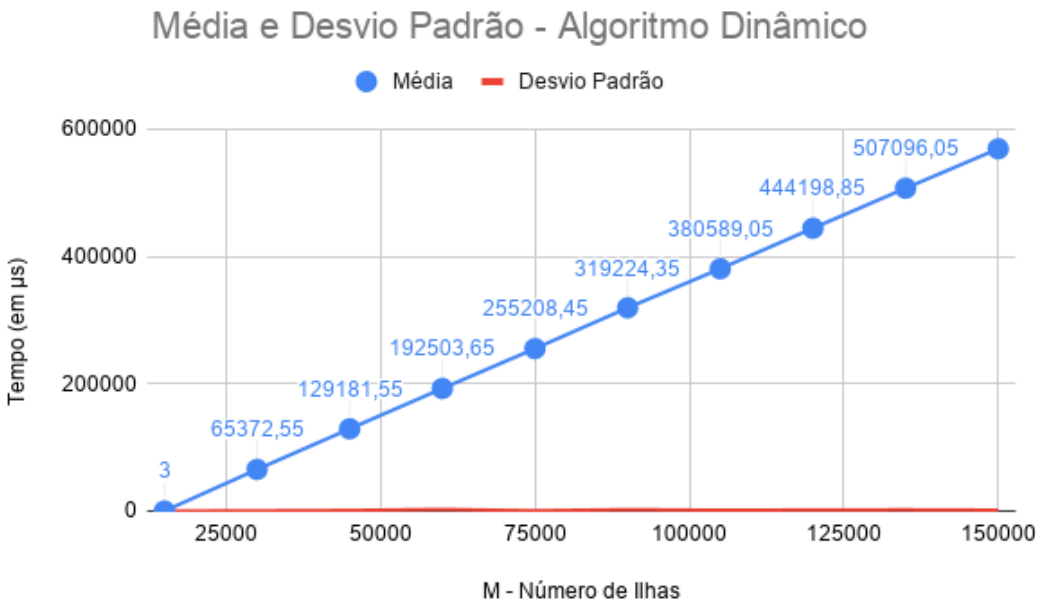
A fim de evitar a interferência devido à troca de contexto do processador junto ao sistema operacional, cada teste foi executado 20 vezes e, a partir dos tempos, calculou-se a média e o desvio padrão.

Apoiando-se nesses dados, gerou-se o seguinte gráfico para a solução gulosa:



Como o esperado, vemos um crescimento ligeiramente maior que linear em relação ao número de ilhas M, o que corrobora a complexidade de tempo encontrada na seção anterior $O(MlogM)$ (o log faz pouca diferença na visualização dos dados), dominado pela ordenação do merge sort.

Não obstante, tem-se o seguinte gráfico para a solução utilizando programação dinâmica:



Como o esperado, vê-se um crescimento linear em relação ao número de ilhas M, já que o custo máximo N foi mantido constante, comprovando a complexidade de tempo $O(N * M)$ identificada na seção anterior.

Não obstante, pode-se observar que o tempo de execução do algoritmo dinâmico cresce muito mais rapidamente que do algoritmo guloso. Isso acontece pois a solução gulosa depende apenas do número M de linhas, enquanto a solução dinâmica tem complexidade $O(N * M)$, ou seja, sempre que M cresce, seu crescimento acompanha um fator N, nesse caso, fixado em 1500.

5. Prova de Corretude

5.1. Algoritmo Guloso

O algoritmo guloso não é ótimo para qualquer solução, o que pode ser demonstrado por contraexemplo. Suponha que exista um orçamento de 200 reais e duas ilhas: a primeira com custo 90 e pontuação 4 e a segunda com custo 100 e pontuação 5. Pelo critério adotado do fator, a primeira ilha tem 22.5, enquanto a segunda tem 20. Assim, a solução gulosa, percorrendo as ilhas de maneira decrescente depois de ordenadas pelo fator, escolheria passar dois dias na ilha de custo 90, totalizando 8 pontos, enquanto a solução ótima é passar dois dias na ilha de custo 100, totalizando 10 pontos,

5.2. Algoritmo Dinâmico

Já o algoritmo dinâmico é ótimo para qualquer solução. Tal prova reside no fato de que a solução considera sempre todas as possibilidades, ou seja, ela considera o que aconteceria caso a ilha fosse adicionada e caso ela não fosse. Evidentemente, por se tratar de uma solução dinâmica, nem toda árvore de possibilidades é varrida, já que existem casos idênticos que não são tratados pois já apareceram anteriormente na matriz de memorização

Dessa forma, como todas as possibilidades são, de uma forma ou de outra, verificadas, pode-se concluir que não existe nenhuma solução melhor que afinal e, portanto, a mesma é ótima.

6. Conclusão

A implementação da solução foi de fundamental importância para efetivar os conceitos e paradigmas vistos em sala de aula para a resolução de problemas, em especial os algoritmos gulosos, a programação dinâmica e a técnica de divisão e conquista.

Certamente a maior dificuldade encontrada foi na implementação da memorização (memoization) da solução dinâmica, que requer um cuidado especial com a ordem com que as ilhas são processadas.

Dessa forma, o trabalho foi, de fato, muito interessante, tanto para praticar o desenvolvimento na linguagem C++, tanto para melhorar a experiência na criação de soluções com diferentes paradigmas de programação.

7. Referências

Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009. Versão Traduzida:Algoritmos – Teoria e Prática 3a. Edição, Elsevier, 2012.

GeeksForGeeks. Merge Sort.

N. Ziviani. Projeto de Algoritmos com Implementações em Pascal e C. Cengage, 2011.