UNIVERSIDADE FEDERAL DE MINAS GERAIS
DCC – DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

TRABALHO PRÁTICO 2

Disciplina: Algoritmos 1

Turma: TN

Professora: Olga Nikolaevna Goussevskaia

Aluno: Daniel Souza de Campos

Curso: Ciência da Computação

Outubro/2020

Sumário

1 Introdução

O **Jogo dos Diamantes** é um procedimento realizado por uma ourives muito metódica ao receber um malote cheio de diamantes. Para cada diamante *i*, teremos um peso p_i associado a ele. Dessa forma, o método combina dois diamantes de pesos p_i e p_j da seguinte maneira:

1. Se $p_i = p_j$, ambos os diamantes são destruídos;
2. Se $p_i > p_j$, $p_i = p_i - p_j$ e vice-versa.

O dito jogo só acaba quando restar um ou nenhum diamante, sendo o objetivo do método obter, caso exista, o diamante com o menor peso. Se não restar nenhum diamante no final, o peso final é considerado 0.

2 Objetivo

O objetivo do trabalho é desenvolver um programa que, dado todas as entradas necessárias para se iniciar uma instância do **Jogo dos Diamantes**, calcule qual será, se houver, o peso do único diamante que restou do procedimento.

Além disso, a ideia é que, durante o desenvolvimento do programa, sejam empregadas duas técnicas diferentes para se resolver o problema, umas delas

usando, obrigatoriamente, **programação dinâmica**. No final, deve-se comparar os tempos de execução de alguns testes para ambas implementações.

3 Configurações

3.1 Recursos para o desenvolvimento

O programa foi desenvolvido na linguagem **C++** utilizando o editor de texto **Sublime Text**. Para controle de versionamento e armazenamento do trabalho foi utilizado o GitHub (disponível em: https://github.com/Pendulun/TP2ALG1). Para a verificação de vazamentos de memória foi utilizado o **Valgrind**. Para compilação foi utilizado o **g++** e o **Ubuntu 18.04 LTS** para **Windows 10** para conseguir executar o Valgrind.

A máquina utilizada para o desenvolvimento, execução e recuperação do tempo de execução dos testes possui 8Gb de memória RAM e processador Intel Core i5-7200U 2.5 GHz.

3.2 Padrão de entrada e saída

De acordo com a especificação do problema, o programa deve ser capaz de ler um arquivo contendo todas as informações necessárias para se criar uma instância do jogo.

O padrão é o que segue:

Exemplo das linhas da entrada

```
K // quantidade de diamantes
D1 D2 .. DK // pesos dos diamantes
```

Figura 1: Padrão de entrada de informações

E, para a saída, apenas indicar o peso do diamante restante ou 0 caso nenhum sobre.

Exemplo da saída

```
X // Peso retornado.
```

Figura 2: Padrão de saída do programa

3.3 Divisão de pastas e arquivos

O programa foi dividido em três pastas: **src**, **include** e **build**. Na pasta **src** estão todos os arquivos **.cpp**, na pasta **include** estão todos os arquivos **.hpp**

e na pasta **build** estão todos os arquivos **.o**. Além disso, existe o **makefile** para compilação do programa. O arquivo executável, depois de rodar o makefile, será gerado fora dessas três pastas no mesmo nível que o arquivo makefile.

4 Versão Não Dinâmica

As partes principais do problema são decidir quais dois diamantes comparar e computar a diferença dos seus pesos de forma ótima. No programa desenvolvido, a coleção de diamantes está salva em uma lista encadeada e esse será o objeto mais usado na resolução do problema.

4.1 Implementação

Depois de pensar por bastante tempo em formas de abordar o problema, foram levantadas algumas possibilidades:

1. Sempre comparar os diamantes com menor diferença de peso;
2. Sempre comparar os diamantes com menor peso;
3. Sempre comparar os diamantes com maior peso;

Porém, nenhuma dessas opções estavam 100% corretas e, consequentemente, não passavam em todos os testes. A abordagem que conseguiu passar em todos os testes foi:

- Sempre comparar os dois diamantes com maiores pesos sendo que $p_i \neq p_j$.

Isso funcionou sempre que os diamantes na coleção, em um dado momento, não possuíam todos os mesmos pesos, isto é, existe pelo menos um diamante com peso diferente dos demais. Para o caso em que todos os diamantes tem peso igual, pode-se perceber que, se houver um número par de diamantes na coleção, o resultado final será 0, já que iríamos destruindo, par a par, os diamantes até não sobrar nenhum. Caso houver um número ímpar de diamantes, basta retornar o peso de um dos diamantes pois só sobrará um deles no final. Dessa forma, o problema seria apenas identificar se existe um estado de igualdade total na coleção, e, se falso, identificar os dois diamantes com maiores e diferentes pesos. Segue um pseudo-código dessa versão:

```
naoDinamico():
  Seja colecaoDiamantes uma lista com os diamantes
  Enquanto o tamanho da colecaoDiamantes > 1:
    Defina: maiorP,segundoMaiorP, maiorIndex, segundoIndex,contador iguais a 0
    Defina isEqual = areEqual()
```

```
    Se isEqual == true:
      Se colecaoDiamantes.size() é par:
        Retorne 0;
      Senão:
        Retorne o peso do primeiro diamante
    Para cada diamante i:
      Se pi > maiorP:
        segundoP = maiorP
        segundoIndex = maiorIndex
        maiorP = pi
        maiorIndex = contador
        continue;
      Se pi > segundoMaiorP e pi != maiorP:
        segundo = pi
        segundoIndex = contador
      contador=contador+1
    fimPara
    Compute a diferença entre os dois maiores
    Retire de colecaoDiamantes o diamante com peso menor
    Atualize o diamante com peso maior como peso sendo igual a diferença
  fimEnquanto
  Se colecaoDiamantes.size()==1:
    Retorne o peso do diamante que sobrou
  Senão
    Retorne 0
```

Sendo que a função “areEqual()” pega o peso do primeiro diamante e o compara com os pesos dos outros elementos na lista, dando a resposta se todos são iguais ou não.

4.2 Análise de complexidade

Seguindo o pseudo-código descrito anteriormente:

- O “Enquanto” irá rodar no máximo n-1 vezes, sendo O(n);
- O “areEqual()” irá rodar no máximo n-1 vezes também (uma vez a cada iteração no “Enquanto”), iterando, no máximo, $\sum_{x=1}^{n} x$ vezes total ao longo do programa, o que é também O(n);
- O “Para cada diamante i”, irá rodar também, no máximo, $\sum_{x=1}^{n} x$ vezes para achar os maiores pesos na coleção de diamantes. Também é O(n).

Portanto, para cada iteração do “Enquanto”, ocorrerão as iterações do “areEqual()” e do “Para”: $O(n) \times \big(O(n) + O(n)\big) = O(n) \times O(n) = O(n^2)$.

5 Versão Dinâmica

5.1 Implementação

A versão dinâmica foi feita com base no algoritmo ***Subset Sums*** no livro de [Kleinberg e Tardos, 2006] ***Algorithm Design***. Esse é um problema que evolui do Problema de Agendamento de Intervalos (***Scheduling Problem***). Nesse problema, tínhamos n intervalos com uma duração d_i, para cada intervalo i, e queríamos simplesmente achar o maior conjunto de intervalos em conformidade que poderiam ser alocados em um recurso independente de qual fosse o maior tempo de término f_i possível para um intervalo. Esse é um problema diferente do Agendamento de Intervalos com Pesos (***Weighted Scheduling Problem***), no qual priorizamos os pesos ou ganhos de cada intervalo.

O nosso problema é uma variação do *Scheduling Problem* no sentido de limitar o término máximo de um conjunto de intervalos. No caso, não teremos intervalos, mas sim, pesos de diamantes. Do mesmo modo, no final, como queremos um diamante com o menor peso possível, o problema seria encontrar uma divisão em dois conjuntos dos diamantes que igualasse, ou tornasse mais próximos, os pesos totais dos dois conjuntos. Dessa forma, se conseguirmos algum conjunto de diamantes, dentre todos os diamantes, em que a soma dos pesos seja no máximo, e preferencialmente, igual à metade da soma dos pesos de todos os diamantes, podemos usar essa soma máxima de pesos do conjunto final para calcular o valor final da resposta. Dessa forma, limitamos o peso máximo possível do conjunto de diamantes: $p_{max} = \frac{\sum_{i=1}^{n} p_i}{2}$.

O algoritmo funciona, basicamente, achando, para cada subconjunto de diamantes, a soma máxima y_{max} menor ou igual a um limite de peso lp para cada limite lp_i entre 0 e p_{max}. Esses valores de y_{max_i} para os limites entre 0 e

$p_{max} - 1$ devem ser calculados para ajudar a encontrar o valor de interesse final $y_{max_{p_{max}}}$. Desse modo, estará sendo criado uma tabela contendo a soma máxima de pesos y_{max} para cada limite de peso possível até p_{max}.

Finalmente, ao analisar $y_{max_{p_{max}}}$, teremos um valor que corresponde a maior soma de pesos possível até p_{max}, sendo que queremos, preferencialmente, $y_{max_{p_{max}}} = p_{max}$. Caso verdadeiro, significa que temos um conjunto de diamantes c_1 cuja soma de pesos p_{c_1} é igual a p_{max} e, portanto, temos um segundo conjunto c_2 com a mesma soma de pesos $p_{c_2} = p_{c_1}$. Portanto, o resultado final seria 0 ao fazer a diferença $p_{c_2} - p_{c_1}$. Caso $y_{max_{p_{max}}} \neq p_{max}$, isso significa que existe um conjunto c_1 cuja soma de pesos p_{c_1} é menor que p_{max}. Logo, o conjunto c_2 possui uma soma $p_{c_2} > p_{max}$ e, portanto, $p_{c_2} > p_{c_1}$. Dessa forma, a diferença $p_{c_2} - p_{c_1}$ nos dará o peso do único diamante restante.

Sendo w_i o peso do diamante i na coleção de diamantes, segue um pseudo-código da implementação:

```
SubsetSum():
  Seja n=colecaoDiamantes.size() e W igual a p_max
  Defina um Array M[0..n][0..W]
  Para cada w = 0,...,W:
    M[0,w]=0
  Para cada i=1,2,...,n:
    Para cada w=0,1,...,W
      Se w<w_i :
        M[i][w] = M[i-1][w]
      Senão:
        M[i][w] = max(M[i-1][w], w_i+M[i_1][w-w_i])
    FimPara
  FimPara
  Defina y_max_pmax = M[n][W]
  Retorne (∑_{i=1}^{n} w_i − y_max_pmax) − y_max_pmax
```

5.2 Análise de complexidade

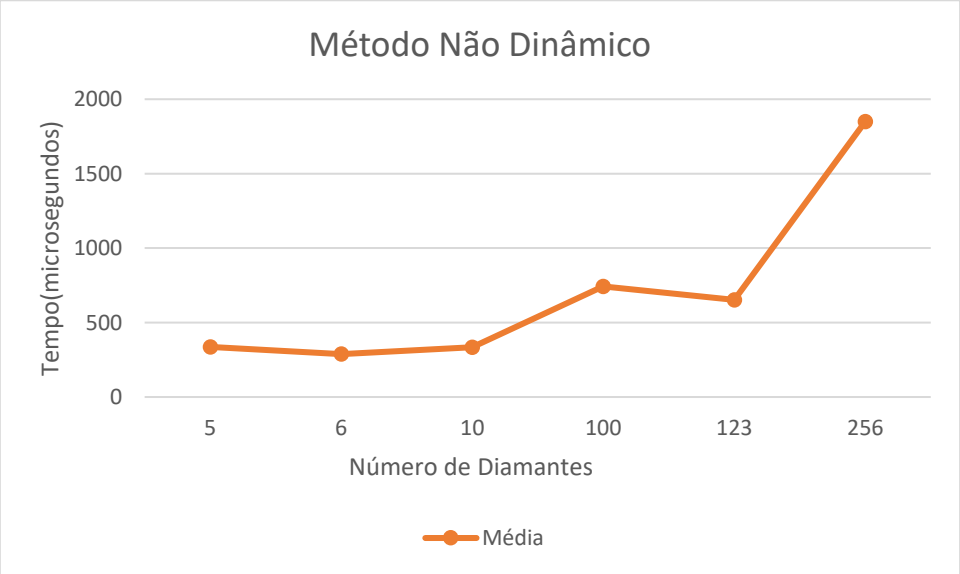
De acordo com o livro de [Kleinberg e Tardos, 2006] ***Algorithm Design***, é possível provar que a complexidade desse algoritmo é $O(np_{max})$.

6 Testes realizados

Para realização da medição do tempo, foram feitas 10 contabilizações de tempo de execução do programa para cada um dos seis testes disponibilizados no moodle nas duas implementações diferentes. Segue os resultados:

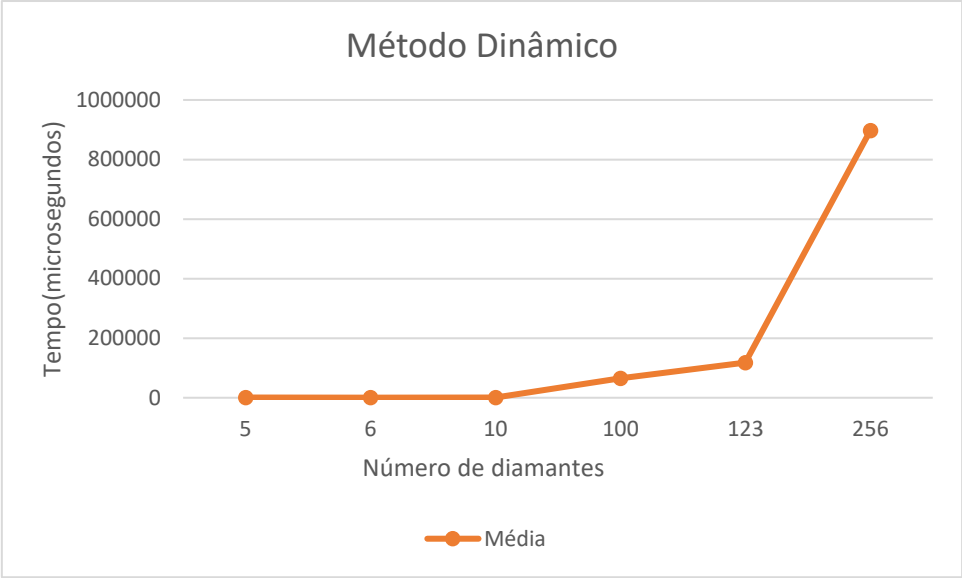
Para o método não dinâmico:

Método	Qt. Diamantes	Média	Desvio Padrão
Não Dinâmico	5	336,9	164,98
	6	288,2	143,47
	10	334,8	164,83
	100	742,8	196,31
	123	652,8	115,25
	256	1849,5	224,11



Para o método dinâmico:

Método	Qt. Diamantes	Média	Desvio Padrão
Dinâmico	5	398,7	154,8
	6	330,6	128,92
	10	490,2	109,89
	100	65608,4	2159,71
	123	117904,2	1201,62
	256	897002,7	76276,61



7 Conclusão

Este trabalho tinha o objetivo de empregar duas soluções diferentes para o Jogo dos Diamantes sendo que um deles deve usar a programação dinâmica. A primeira abordagem utiliza do pressuposto de sempre pegar os dois diamantes com os maiores pesos sendo ambos diferentes e, apenas quando necessário, pegar dois diamantes com pesos iguais. A segunda abordagem, com programação dinâmica, cria uma tabela de tamanho n x metade da soma dos pesos totais.

Ambas abordagens passaram nos testes e tiveram suas complexidades calculadas. De acordo com os resultados obtidos nos testes, a primeira abordagem se mostrou mais eficiente que a que usa programação dinâmica. Isso se deve ao fato de a segunda abordagem estar muito ligada à soma dos pesos dos diamantes e, quanto maior essa soma, maior a metade dela e isso se reflete na complexidade encontrada de $O(np_{max})$. A primeira não está tão ligada à essa soma, mas sim, ao número de diamantes na entrada. Com certeza deve ser possível estabelecer uma relação entre o número de diamantes na entrada e a metade da soma total dos pesos para decidir qual abordagem utilizar.