Algoritmos 1 - TRABALHO PRÁTICO 2:
O JOGO DOS DIAMANTES

Letícia da Silva Macedo Alves <leti.ufmg@gmail.com>

Matrícula: 2018054443

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

1. Introdução

A proposta do trabalho é a resolução do problema "Jogo dos Diamantes". As regras básicas são: Dado um conjunto de diamantes, pegar um par e então comparar os dois pesos. Se são iguais, ambas as pedras são destruídas. Se são distintos, a de menor peso é destruída e a de maior peso recebe novo peso igual a diferença entre os dois pesos. Isso se repete até que reste um diamante ou nenhum.

O objetivo do jogo é, portanto, que, dado o conjunto de diamantes de interesse, obedecendo às regras de comparação de seus pesos aos pares, ao final obteremos no máximo um diamante, de tal forma que, o peso retornado será zero (caso não reste nenhum diamante) ou o peso do diamante que sobrou, que deve ser o menor possível. Ou seja, o jogo consiste em minimizar o peso do único diamante restante ou, se possível, que fazer com que não reste nenhum (logo, o peso é zero).

Serão analisadas duas versões de algoritmo, ambas de corretude verificadas, porém de complexidades distintas. O trabalho focará na solução que utiliza Programação Dinâmica (PD), porém, com o objetivo de avaliar experimentalmente essa solução, utilizaremos como comparação uma versão do algoritmo que não utiliza PD e que se aproxima do método conhecido como Força Bruta. Teremos assim bons parâmetros de comparação de complexidade do algoritmo escolhido.

2. Implementação

Para modelar o problema dos diamantes, ele foi reduzido ao problema de encontrar a diferença mínima entre a soma dos elementos de dois subconjuntos (no caso, pesos de diamantes). Essa forma de resolução mostrou-se bastante adequada, atendendo perfeitamente ao que se esperava da solução do problema. O raciocínio para tal escolha e sua relação com as regras do Jogo dos Diamantes serão devidamente explicitados ao longo do documento.

O programa foi escrito utilizando a linguagem C++11 e foi implementada uma única classe: *JogoDosDiamantes*.

O formato de entrada, como o proposto, é por passagem dos arquivos de teste como parâmetro para o programa via linha de comando. Os arquivos de entrada são compostos de um número inteiro "K", que diz quantos diamantes possui o conjunto, e, em seguida o peso (também inteiros) de cada um dos diamantes.

Já a saída utiliza para imprimir o resultado é o padrão de sistema stdout (cout). Sendo assim, o programa imprime, ao final da computação, o valor que representa o peso (mínimo) retornado.

Além disso para a verificação de alocação e desalocação correta de memória, foi utilizado o Valgrind.

2.1. A classe JogoDosDiamantes

A classe *JogoDosDiamantes* possui os atributos: *int numDiamantes* e *int* pesosDiamantes*. O *numDiamantes* (quantidade de diamantes no conjunto) é lido da entrada pelo construtor da classe e utilizado para alocar um array de inteiros, o *pesosDiamantes*. Este array representa o conjunto de pedras através do peso de cada uma (isto é, cada posição guarda o peso de um diamante). É também no construtor que os pesos de cada diamante é lido da entrada e armazenado no array apresentado.

Além do construtor e do destrutor, a classe possui o método *void pesoRestante()*, que é o responsável por, de fato, encontrar qual será o peso restante após o jogo.

2.1.1 O método void pesoRestante()

É neste método onde, basicamente, tudo acontece. A implementação foi baseada no conceito de Programação Dinâmica. Isto é, considerando que a solução geral do problema pode ser computada através da solução de subproblemas menores, a PD melhora a performance do programa por "memorizar" os resultados parciais, que podem ser usados mais de uma vez durante a resolução do problema principal. Dessa forma, evita-se a recomputação de um mesmo subproblema cada vez que ele é usado, pois, na primeira vez que obtemos sua solução, ela é armazenada e pode ser requisitada posteriormente.

Optei, como estrutura para a técnica de "memorização", por uma tabela de booleanos, denominada *tabelaPD*. O número de linhas e colunas da tabela, respectivamente, é igual ao *numDiamantes+1* e *(somaPesosTotal/2)+1*, sendo que *somaPesosTotal* é a soma de todos os pesos em *pesosDiamantes*.

Os detalhes da tabela serão especificados mais adiante, mas a ideia geral de seu funcionamento é a seguinte: Uma posição *tabelaPD[l][c]* (sendo *l* a linha e *c* a coluna) terá valor *true* se algum subconjunto de pedras de índices 0 até *l*-1 (no array *pesosDiamantes*) some exatamente o peso de valor *c* (isto é, o valor da coluna em questão). A tabela é então preenchida seguindo este raciocínio.

Fica fácil pensar então que, como nosso objetivo é dividir os pesos em dois subconjuntos de tal forma que a diferença de suas somas seja a menor possível, o ideal para isso seria que, em *tabelaPD[numDiamantes][somaPesosTotal/2]* houvesse o valor *true*. Isso porque significaria que há um subconjunto de pesos que soma, exatamente ou quase, metade do peso total (ou melhor dizendo, a parte inteira ou "piso" de *somaPesosTotal/2*).

Logo, o subconjunto restante só pode somar o que restou dos pesos, ou seja, *somaPesosTotal – somaPesosTotal/2* (o "teto" de *somaPesosTotal/2*). Então teremos nossa resposta ótima: a diferença mínima entre os pesos de dois subconjuntos é igual a 0 (se *somaPesosTotal* é par) ou 1 (se *somaPesosTotal* é ímpar).

Porém, nem sempre essa divisão exata é possível, mas mesmo assim o algoritmo retornará a menor diferença possível entre dois subconjuntos. Para chegar a essa diferença, basta percorrer *tabelaPD[numDiamantes][c]* da direita para a esquerda, com *c* inicialmente igual a *somaPesosTotal/2*, que seria o ideal, e ir decrementando (até, no máximo, *c = 0*) até que encontremos um valor *true*. Dessa forma o peso restante será o módulo da diferença: *((somaPesosTotal – c) – c)*, ou seja, a soma do subconjunto 2 menos a soma do subconjunto 1.

3. Exemplo do algoritmo para o Jogo dos Diamantes

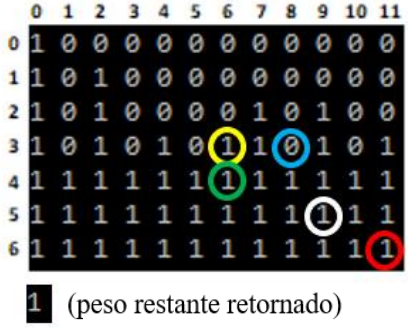
Consideremos a entrada:

```
6
2 7 4 1 8 1
```
(lembrando que o array que guarda os pesos vai do índice 0 até 5)

Temos então 6 diamantes com os respectivos pesos apresentados.

Nossa *somaPesosTotal* é 2+7+4+1+8+1 = 23. Como as colunas da tabela são as possíveis somas que poderíamos obter com o conjunto de diamantes, vão de índices 0 à 11 (que é a parte inteira de 23/2).

Nossa tabela de booleanos, *tabelaPD*, fica como se segue:



1 (peso restante retornado)

Como dito anteriormente, o número de linhas e colunas da tabela, respectivamente, é igual ao *numDiamantes+1* e *(somaPesosTotal/2)+1*.

É importante notar que, antes de começar a operar com os pesos dos diamantes, iniciamos a primeira coluna inteira com *true*. Isso porque, seguindo o raciocínio da tabela como um todo, é possível encontrar uma soma igual a 0 com qualquer subconjunto de pesos (basta que nenhum deles seja somado). Além disso, a primeira linha inteira (exceto a posição [0][0]) recebe *false*, pois com um subconjunto vazio de pesos não é possível obter nenhuma soma (a não ser a soma 0).

Vamos agora entender a ideia de preenchimento da tabela. A linha de índice 3 representa o 3º peso do array *pesosDiamantes*. No caso, o 3º elemento é o peso 4. Sendo assim, explicarei a seguir porque as posições circuladas em amarelo e azul recebem, respectivamente, valor *true* e *false*.

Primeiramente, ambas as posições estão na mesma linha, a linha que representa o peso de valor 4. O círculo em amarelo ([3][6]) está na coluna 6. Então, nosso objetivo é saber se, com os elementos 2, 7, 4 existe algum subconjunto que some 6. Inicialmente, o algoritmo coloca o valor *false* nessa posição (pois é o valor em [2][6]). O motivo disso que é, até o momento, nenhum subconjunto de 2, 7 conseguiu somar 6. Mas, em seguinda, verifica-se que 4 é menor ou igual a 6 (isto é, a soma que queremos obter). Então, talvez, adicionando o elemento 4 ao conjunto, seja possível somar 6 com algum subconjunto de pesos. No caso isso é de fato possível e o subconjunto de pesos é {2, 4}. Então:

tabelaPD[3][6] = tabelaPD[2][6-pesosDiamantes[2]] (o índice 2 no array de pesos, é o peso 4)

Ou seja, *tabelaPD[3][6]* será *true*, pois com o conjunto sem o peso 4 (ou seja, com os valores 2, 7) já era possível somar 2. Logo, adicionando o valor 4, é possível obter soma 6.

Agora, analisemos o valor *false* circulado em azul ([3][8]). Como estamos, novamente, na linha 3, continuamos analisando o impacto da adição do peso 4 ao conjunto de pesos analisado. Dessa vez, queremos saber se algum subconjunto entre 2, 7, 4 consegue somar 8. Mas com o conjunto 2, 7 não conseguíamos somar 4, pois *tabelaPD[2][8-pesosDiamantes[2]]* é *false*. Logo, adicionando o peso 4 ao conjunto, também não será possível obter soma 8. Assim, *tabelaPD[3][8]* será *false*.

Vamos ainda analisar o valor *true* circulado em verde ([4][6]). A linha de índice 4 refere-se ao 4º peso, isto é, o peso de valor 1. Nós queremos a soma 6 (índice da coluna nessa posição) para algum subconjunto entre os valores 2, 7, 4, 1. Mas já vimos que é possível somar 6 observando a posição [3][6], que é *true*. Logo, adicionando o valor 1, continua havendo um subconjunto que soma 6 (no caso, {2, 4}). Assim, a posição *tabelaPD[4][6]* será *true*.

Com esse último exemplo, notamos que, na verdade, após um *true* ser obtido em *tabelaPD[l][c]*, todos os valores na mesma coluna das linhas abaixo dessa com *true* receberão *true* também, já que um subconjunto com soma *c* já foi encontrado e isso segue verdadeiro adicionando mais pesos ao conjunto.

Por último, para uma observação importante, considere o valor *true* circulado de branco ([5][9]). Nesse caso, pretende-se obter soma 9 com algum subconjunto em 2, 7, 4, 1, 8. A *tabelaPD[2][9]* já é *true*, pois com os elementos 2, 7 é possível somar 9. Logo, todas as posições abaixo, na mesma coluna, receberão *true* (como explicado no parágrafo anterior). Note que, em *tabelaPD[5][9]* com o conjunto 2, 7, 4, 1, 8 continua sendo verdade que algum subconjunto consiga somar 9. Por exemplo {2, 7} ou ainda {1, 8}. Com isso, observamos que não importa qual subconjunto some um valor *c*. Basta que tal subconjunto exista e pode, inclusive, haver mais de um subconjunto que satisfaz essa propriedade.

Finalmente, ao preenchermos toda a tabela, podemos encontrar a solução ótima. Basta verificar que *tabelaPD[6][11]* (circulado em vermelho) é *true*. Isso significa que, utilizando todos os diamantes, é possível obter um subconjunto tal que a soma seja a parte inteira de 23/2, isto é, 11. Ou seja, já encontramos, com a primeira iteração de *c*, a menor diferença entre dois grupos de pesos de diamantes. Se encontramos um subconjunto que soma 11, o outro subconjunto soma 23-11 = 12. Logo, o peso restante retornado é 12-11 = 1.

4. O Jogo dos Diamantes e a diferença mínima entre dois subconjuntos

Nesta sessão explicarei melhor a relação entre o problema "Jogo dos Diamantes" e o problema de achar a diferença mínima entre dois subconjuntos. Vejamos então o porquê de o primeiro poder ser reduzido ao segundo.

Relembremos primeiro as regras do Jogo dos Diamantes.

Temos um conjunto de pedras com seus respectivos pesos e devemos "combiná-las" aos pares (considerando duas pedras quaisquer com os pesos p1 e p2) da seguinte forma:

-Se p1 = p2, então as duas pedras são destruídas

-Se p1 > p2, a pedra com p2 é destruída e a restante sobra com um novo peso p1-p2

O jogo termina com um ou nenhum diamante restante. Queremos que o peso restante seja o menor possível (o algoritmo então retorna 0, se não sobrou nenhuma pedra ou o peso da única pedra que sobrou). Assim, devemos escolher as pedras aos pares de maneira ótima, isto é, dentre todas as combinações possíveis de diamantes, queremos aquela que retorna o menor peso ao final.

Consideremos separar o conjunto de diamantes em dois. Queremos minimizar a diferença (em módulo): |"soma dos diamantes do subconjunto2" – "soma dos diamantes do subconjunto1"| = peso restante.

Seguindo a lógica das regras do jogo, sempre que escolhemos um par de diamantes, um deles vai para o subconjunto1 e o outro para o subconjunto2. Note que, dessa forma, se os dois diamantes possuem mesmo peso, um anulará o peso do outro no peso restante (cálculo acima), ou seja, as duas pedras são "destruídas". Porém, se os dois diamantes possuem pesos diferentes, a pedra de menor peso (p2) anula seu peso da pedra de maior peso (p1), então, a pedra de menor peso é "destruída" e a de maior peso "sobra com um novo peso p1-p2" no cálculo do peso restante.

Note que, reduzindo o problema do Jogo dos Diamantes ao de encontrar uma divisão do conjunto em dois subconjuntos tal que a diferença entre suas somas seja a menor possível, o que importa de fato é a soma dos elementos que estarão em cada um dos subconjuntos. Mas, uma vez que já temos os dois subconjuntos completos que retornam a solução ótima, não faz diferença, na prática, saber quais foram os pares de diamantes "combinados", pois o cálculo de peso restante considera apenas a visão macroscópica, isto é, a soma de todos os elementos de cada subconjunto. Por exemplo:

Sendo, mais uma vez, os pesos: 2, 7, 4, 1, 8, 1

A divisão ótima em dois subconjuntos pode ser, por exemplo: {4, 7} e {2, 1, 1, 8} ou ainda {7, 2, 1, 1} e {4, 8}, pois os módulos da diferenças das somas são ambos iguais a 1, que é o peso resultante retornado pelo algorítmo. Então, uma vez divididos os subconjuntos, em {4, 7} e {2, 1, 1, 8}, não é importante saber se o diamante de peso 4 foi combinado com o de peso 2 ou com o de peso 1, por exemplo. Além disso, mais uma vez, é reforçado que pode haver mais de uma divisão ótima em dois subconjuntos.

Na verdade, o algoritmo é ainda mais abstrato, uma vez que não há, de fato a divisão explícita de um conjunto em dois subconjuntos. O que importa para os cálculos (e que é visível pela análise da tabela de booleanos do algoritmo) é se, para uma certa soma, se tal divisão em subconjuntos existe ou não e qual será então o valor ótimo da diferença a ser retornado.

5. Instruções de compilação e execução

A compilação é realizada utilizando o compilador GCC. Como o projeto faz uso de Makefile, o processo de compilação dos códigos-fonte é resumido a um único comando "make". Ou seja, uma possível sequência de comandos para compilação e execução do programa em sistema Unix é:

"$ make"

"$ ./tp2 jogo_do_diamante.txt"

6. Análise de Complexidade

6.1.Espaço

A complexidade de espaço do problema é dominada pelo espaço alocado para a *tabelaPD* de booleanos. Sendo assim, como a tabela possui (*numDiamantes+1*)*((*somaPesosTotal*/*2*)+*1*) elementos, a complexidade espacial do programa é O(*numDiamantes***somaPesosTotal*), já que podemos desconsiderar em análises de complexidade as constantes 1 (somando) e ½ (multiplicando).

Vale ressaltar que elementos que não envolvem diretamente alocações e manipulações de arrays (como, por exemplo, variáveis auxiliares durante o código) são O(1) em memória, podendo ser desconsideradas na análise espacial.

6.2.Temporal

Como o programa realiza um trabalho constante por espaço na *tabelaPD*, a complexidade temporal também é O(*numDiamantes***somaPesosTotal*), pois o que fazemos é, basicamente, preencher cada elemento da tabela.

Podemos também desconsiderar operações que são O(1) em tempo de execução na análise (como simples atribuições de valores, chamada de função, comparações, retornos, operações aritméticas simples etc).

7. Análise Experimental

Para cada tamanho de entrada K e S (isto é, o número de diamantes do conjunto e soma de pesos total) o programa foi executado 10 vezes. Então, com os 10 valores de tempos de execução, foi calculada a média e o desvio padrão para cada tamanho de entrada.

Os casos de testes usados possuem os seguintes valores para K e S:

Teste	K	S
1	5	266
2	6	23
3	10	818
4	100	6597
5	123	8541
6	256	15543

Com base nesses dados, foi construído o gráfico da página a seguir.

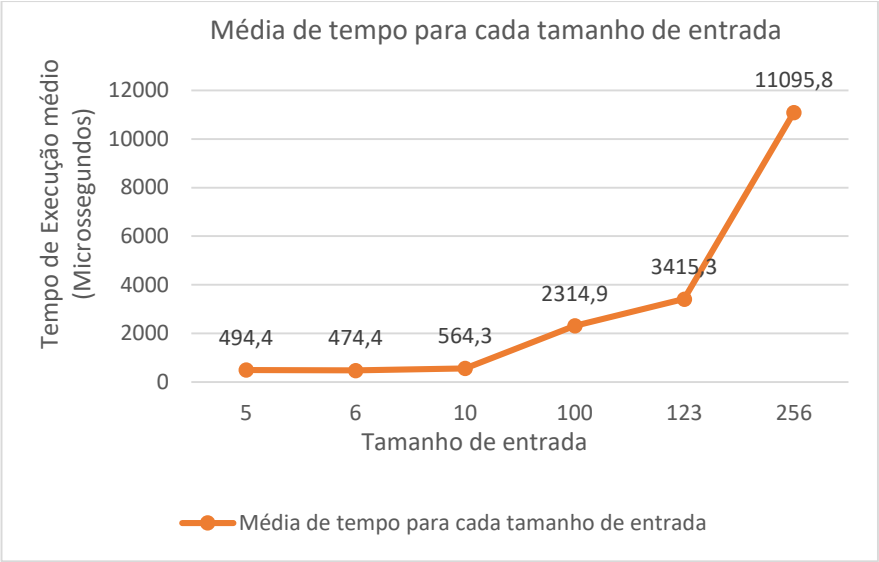
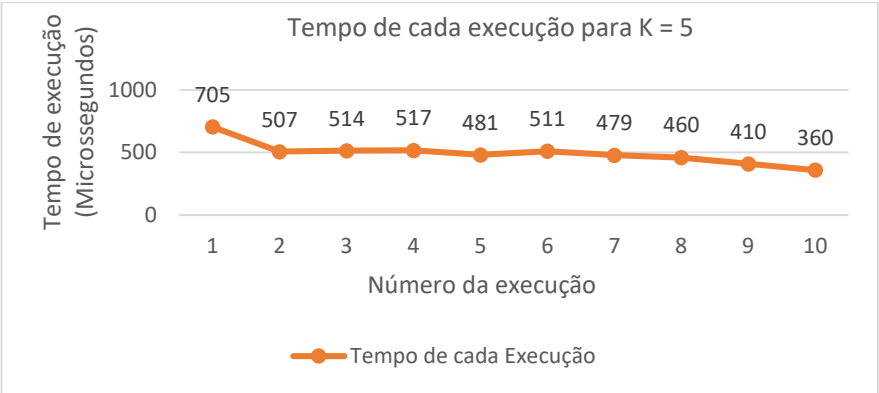


*Gráfico de média de tempo de execução (eixo y) realizada com base em 10 iterações do programa para cada tamanho de entrada (eixo x). O tamanho de entrada K refere-se a quantidade de diamantes do conjunto (com os respectivos valores de S já citados).

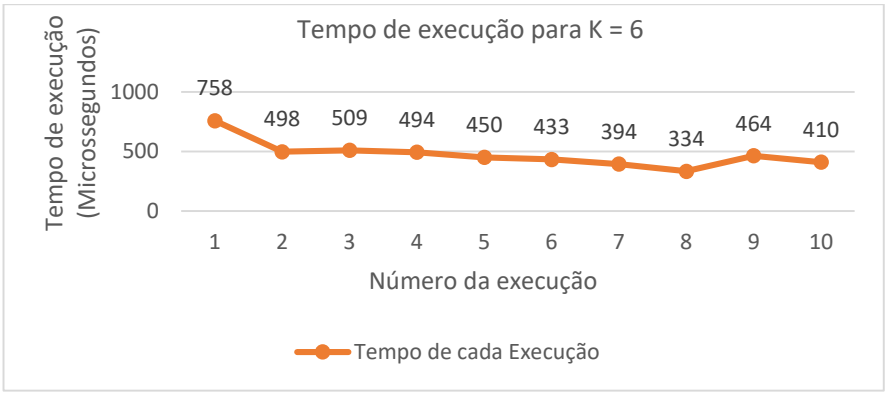
Com a análise experimental, é observável o aumento gradativo do tempo de execução com base tanto no aumento de entrada K, bem como com as diferentes somas totais de pesos S. Isso corrobora a análise de complexidade temporal citada anteriormente. Isso porque, quando aumentamos a quantidade de diamantes do conjunto, o tempo de execução aumenta proporcionamente a esse aumento de K e também à soma dos pesos desses K diamantes. Isso é visível no fato de a entrada de tamanho 5 ter média de tempo um pouco maior do que a de tamanho 6. Como já citado, o caso de teste com 5 diamantes possui soma de pesos 266 e no caso com 6 diamantes a soma é 23.

Segue abaixo os gráficos com os tempos de execução coletados (para gerar o gráfico acima) com os respectivos valores de média e desvio. O eixo x é a iteração do programa (isto é, o número da execução) e o eixo y o tempo de execução dado em microssegundos.

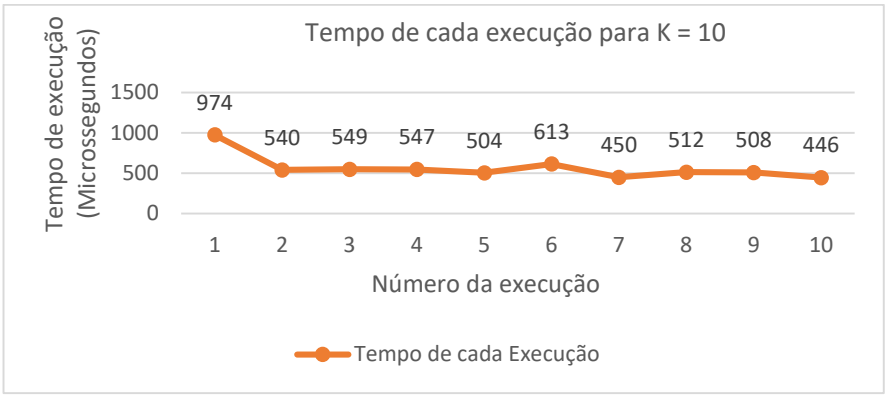
• Tamanho 5 de entrada: (Média = 494,4; Desvio = 89,7)



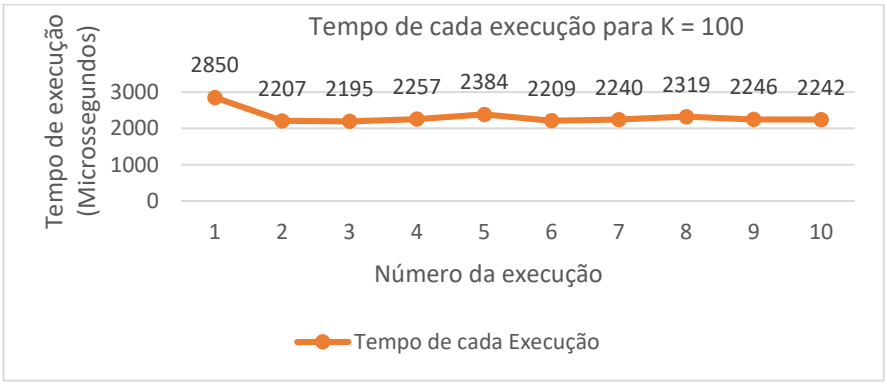
• **Tamanho 6 de entrada: (Média = 474,4; Desvio = 113,1)**



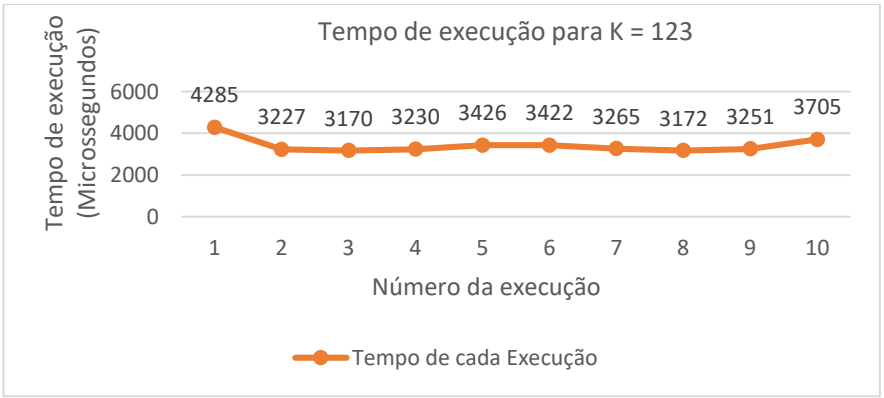
• **Tamanho 10 de entrada: (Média = 564,3; Desvio = 152,0)**



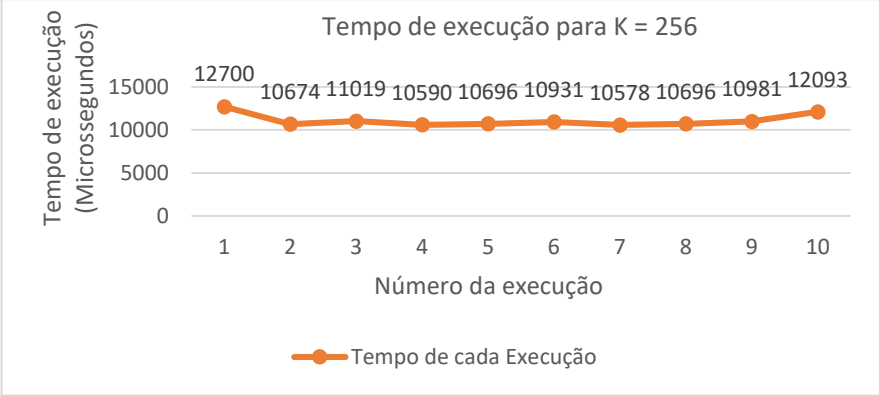
• **Tamanho 100 de entrada: (Média = 2314,9; Desvio = 196,4)**



• **Tamanho 123 de entrada: (Média = 3415,3; Desvio = 346,2)**



• Tamanho 256 de entrada: (Média = 11095,8; Desvio = 717,6)



8. Comparação: Sem e com o uso de Programação Dinâmica

8.1. Um algoritmo Força Bruta para o problema

O algoritmo apresentado até agora neste documento utilizou o conceito de Programação Dinâmica (PD) para encontrar a solução do problema. Nesta sessão faremos uma breve apresentação de outro algoritmo que se aproxima do conceito de "Força Bruta".

Tal algoritmo sem o uso de PD, para fins de comparação, é recursivo e seu pseudocódigo é:

```
pesoRestanteRecursivo(pesos, K, indice, soma1, soma2):
    Se (indice = = K):
        Retorne módulo de (soma1 - soma2)
    Retorne o minimo entre pesoRestanteRecursivo (pesos, K, indice+1, soma1+pesos[indice], soma2)
    e pesoRestanteRecursivo (pesos, K, indice+1, soma1, soma2+pesos[indice])
```

(Sendo K o número de diamantes do conjunto e, na chamada da função pela main, indice, soma1 e soma2 são passados como 0, isto é, pesoRestanteRecursivo(pesosDiamantes, K, 0, 0, 0))

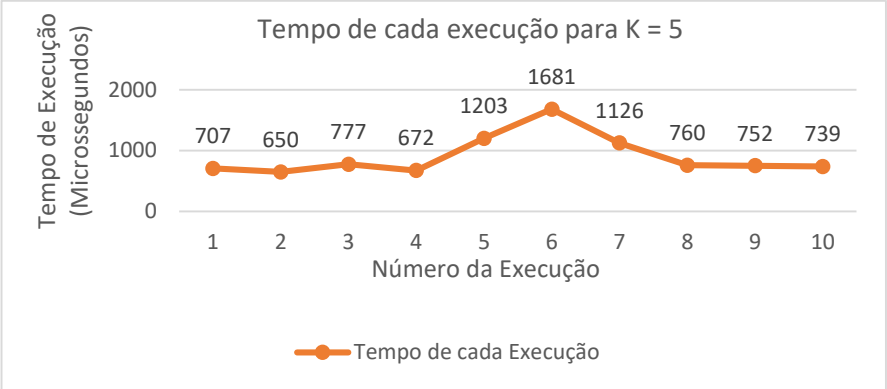
Seguindo essa ideia, todas as possíveis somas de pesos do conjunto são geradas e verifica-se qual é a solução ótima. Como para gerar todas essas possíveis somas nos baseamos na inclusão ou exclusão o elemento pesos[indice] no subconjunto 1, para todo peso há duas possíveis escolhas: estar no subconjunto 1 ou não estar (por consequência, ficará no subconjunto 2).

Logo, a complexidade temporal é $O(2^K)$, isto é, uma complexidade exponencial. Já a complexidade espacial para essa recursão é linear, O(K), já que, a pilha de chamadas recursivas possui tamanho no máximo K.

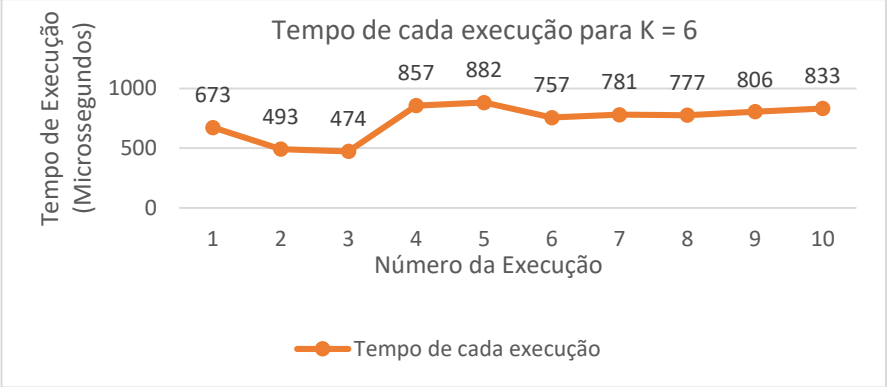
Abaixo estão os gráficos com os tempos de execução do algoritmo Força Bruta para 10 iterações de execução para cada tamanho de entrada, da mesma forma que fizemos para o algoritmo de PD.

Uma observação é que utilizei apenas as entradas de tamanho 5, 6 e 10 desta vez, pois o algoritmo mostrou-se ineficiente para as entradas de tamanho 100, 123 e 256. Não foi possível então obter uma resposta do programa para entradas muito grandes devido à sua complexidade.

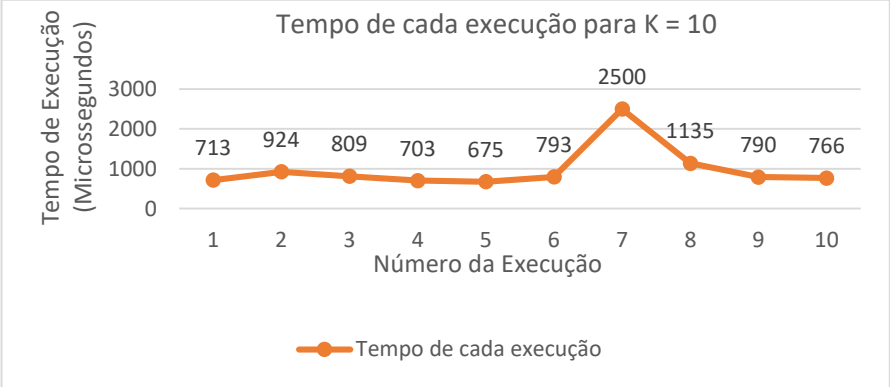
• Tamanho 5 de entrada: (Média = 906,7; Desvio = 331,1)



• Tamanho 6 de entrada: (Média = 733,3; Desvio = 143,7)



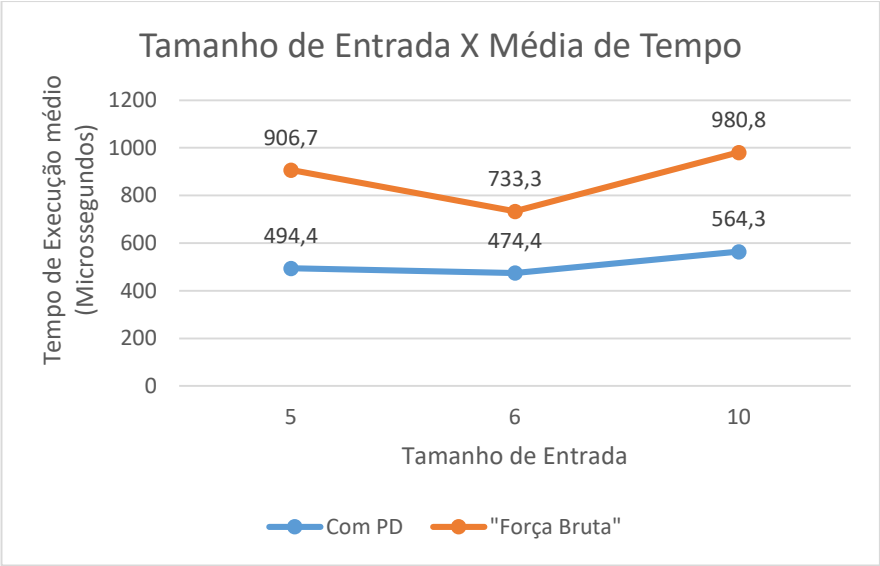
• Tamanho 10 de entrada: (Média = 980,8; Desvio = 550,2)



8.2. Comparação entre as duas versões

Finalmente, nessa seção faremos uma comparação visual entre as duas versões de algoritmo propostas para o problema.

Para melhor visualizar o que acontece com seus tempos de execução com o aumento das entradas, foi gerado o gráfico da página seguinte: as médias de tempo de execução para os mesmos casos de teste com as entradas de tamanho 5, 6 e 10 para cada versão do algoritmo.



*Comparação de médias de tempo de execução (eixo y) realizada para entradas de tamanho K = 5, 6 e 10 (eixo x). O tamanho de entrada K refere-se à quantidade de diamantes presente no conjunto do jogo.

É notável que, mesmo para entradas pequenas, o algoritmo que utiliza PD possui menor tempo de execução para todas as três entradas analisadas. Os resultados permitem, mais uma vez, reforçar as diferenças entre as complexidades temporais. Enquanto o que usa PD é O(*numDiamantes* somaPesosTotal*), o Força Bruta recursivo é $O(2^K)$, isto é, $O(2^{numDiamantes})$.

Porém, analisando a complexidade espacial, a versão Força Bruta apresentou menos memória requerida, sendo O(K), isto é, O(*numDiamantes*) chamadas recursivas. Já a versão usando PD, como visto, é O(*numDiamantes*somaPesosTotal*) também para memória.

Apenas a versão Dinâmica utiliza o valor S referente a soma total dos pesos dos diamantes na análise de complexidade.

9. Conclusão

O uso de Programação Dinâmica mostrou-se adequado e eficiente no cálculo do *pesoRestante* de menor valor possível, como propôs a especificação. A redução do problema à menor diferença entre a soma de dois subconjuntos garantiu a corretude, retornando os valores ótimos esperados para os testes.

Além disso, a complexidade temporal utilizando PD melhorou significativamente se comparada à complexidade do algoritmo de Força Bruta (embora o algoritmo usando PD possua maior complexidade espacial). Dessa forma, é notória a importância de se dividir um problema maior em subproblemas cujos resultados são mais fáceis de computar e que podem ficar “memorizados” (armazenados em uma tabela) para serem posteriormente usados em problemas maiores evitando recomputações desnecessárias de soluções parciais.