Desenvolvimento de Algoritmos para Solucionar o Problema "Somando Dominós"

Gabriel Solonynka Hoffmann[1], Vanessa Albino da Silveira[1]

[1]Ciência da Computação – Universidade do Vale do Rio dos Sinos (UNISINOS)
São Leopoldo – RS – Brasil

{gabrielhoffmann254, silveiravnss}@gmail.com

Resumo. *Este artigo descreve o desenvolvimento de duas diferentes soluções para o problema proposto na cadeira de Análise e Projeto de Algoritmos. O problema é nomeado "Somando Dominós" e consiste em apresentar a partir de um conjunto de dominós, se é possível combinamos de forma que a soma da parte de cima seja igual a de baixo, descartando no máximo um dominó quando for necessário.*

1. Introdução

O problema apresentado informa que para um valor de entrada "n", deve ser informado "n" pares de dominó com valores no formato "A B", sendo "A", "B" e "n" valores numéricos de 0 a 1000. A partir da sequência lida de dominós, devemos procurar a possibilidade de inversão desses para que a soma do lado de cima de todos os dominós seja igual a de baixo.

Caso não seja encontrada uma combinação possível, existe a possibilidade de descartar um par da lista, desde que mantenha-se a maior soma possível.

2. Primeira Solução

A primeira solução foi feita em Java, ela funciona lendo o arquivo de dominós e armazenando eles em uma lista da classe dominó (dominoList). Essa classe tem como atributo os dois lados do dominó um valor de descarte (o quanto impactaria na soma dos dois lados caso o dominó fosse descartado) e um valor de inversão (o quanto impactaria na soma caso o dominó fosse invertido). A lista é ordenada de forma que para um dominó [A,B] "A" sempre seja maior que "B" para facilitar os testes.

É criada uma lista temporária (tempDominoList) que é incrementada a cada teste com um novo dominó. Primeiro se testa um dominó por um em dominoList, depois a tempDominoList passa a armazenar dois dominós para comparação e assim por diante. O teste é feito fazendo um looping na lista completa de dominós e comparando se a soma dos valores de inversão dos dominós ser igual a diferença entre a soma de cima e a soma de baixo da lista completa. Para o descarte é utilizado o mesmo loop e conferimos no cálculo ao invés do valor de inversão, o valor de descarte.

Exemplo: [6,3], [2,1], [3,1]: o valor de inversão do par [6,3] é (6 - 3)* 2 = 6, a diferença entre a soma de cima e a soma de baixo é (6+2+3) - (3+1+1) = 6. Então o valor que buscamos é uma combinação de dominós com o valor de inversão ou descarte = 6 que é o caso do par citado anteriormente.

3. Segunda Solução

A segunda solução foi uma melhoria da primeira, ela funciona com a mesma estrutura de classes, porém com alterações na lógica para um desempenho mais eficaz.

A maior diferença entre as soluções é que utilizamos de recursão nos testes de tempDominoList e utilizamos comparações que nos permitem filtrar muitos casos de testes antes de criar novas iterações na lista. Quando um caso passa da soma necessária para atingir a diferença descartamos o teste e todos os possíveis testes subsequentes com aquela combinação de dominós. Para filtrar alguns casos impossíveis de sucesso com a remoção só fazemos um looping dentro da lista total de dominós caso a soma dos valores de inversão dos dominos sendo testados com o dominó de maior valor de remoção sejam maiores do que diff pois se fosse menor significaria que nem o dominó com maior valor de descarte poderia ser combinado com os dominós tempDominoList, então dominós menores também não poderiam entrar na comparação.

4. Resultados

4.1. Resultados da Primeira Solução

O algoritmo tem uma lógica que percorre muitos loops dentro dele, um de cada vez. isso dificulta muito o teste de casos quando n é maior que 9, pois valor de sucesso demora a ser encontrado. A partir desse valor, o tempo de execução começa a crescer exponencialmente. Um exemplo abaixo da relação entre tempo e o número de dominós a serem comparados quando o algoritmo roda em seu pior caso.

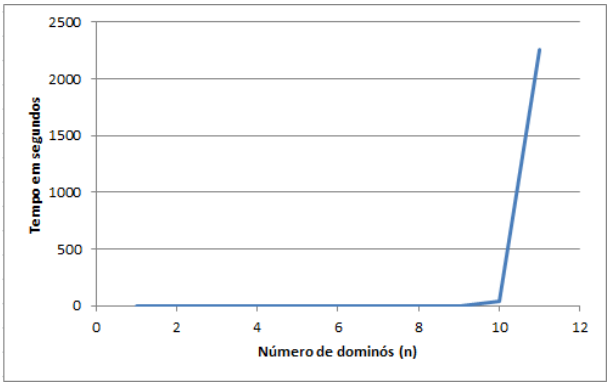

Figura 1. Gráfico com o tempo de execução do algoritmo relação entre o tempo em segundos e o número de dominós na lista mencionada na seção 4.1.

Dos testes solicitados, apenas in, in1 e in2 tiveram um tempo hábil para serem encontradas suas soluções

4.2. Resultados da Segunda Solução

O algoritmo tem uma alta complexidade de O(2^N) no pior caso, pois a cada teste de possibilidade na lista tem que percorrer um loop dentro dela mesmo e isso dificulta muito o teste de casos quando n é maior que 29 e o valor de sucesso demora a ser encontrado. A partir desse valor, o tempo de execução começa a crescer exponencialmente. Um exemplo abaixo da relação entre tempo e o número de dominós

a serem comparados quando o algoritmo roda em seu pior caso.

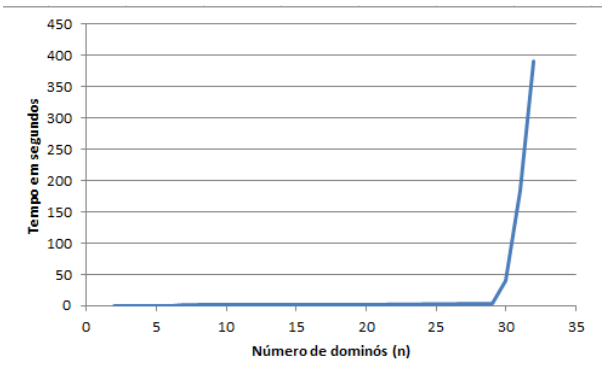

Figura 2. Gráfico com o tempo de execução do algoritmo relação entre o tempo em segundos e o número de dominós na lista mencionada na seção 4.2.

Dos casos de testes solicitados, aqueles dos quais conseguimos obter resultados foram todos os casos de input in, in1 e in2 e quando chega no in3 que os valores de n são maiores, as combinações foram encontradas quase instantaneamente em todos os testes que não envolviam descartes de dominó, pois esses não chegam no tempo de execução do pior caso.

5. Conclusão

A melhor solução encontrada funciona bem para casos com n sendo um valor abaixo de 30, mas assim como visto em aula, quando uma lista passa a ser repetida dentro dela mesma, no pior caso, com uma complexidade de O(2^N) a partir de um certo ponto temos uma evolução exponencial no tempo de execução conforme n aumenta. O gráfico abaixo retirado do site DEV Community mostra bem esse funcionamento.

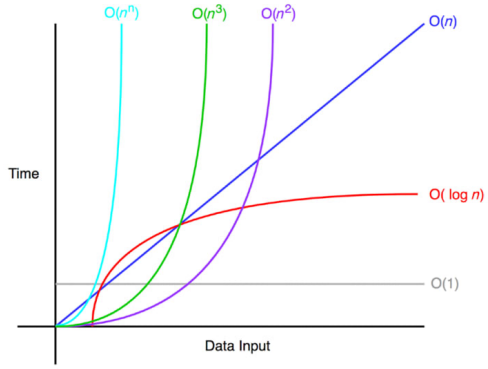

Figura 3. Gráfico com a complexidade de algoritmos mencionado na seção 5.

Referências

Bonvic, B. (2020) “Understanding Big-O Notation With JavaScript”, Em: DEV Community, https://dev.to/b0nbon1/understanding-big-o-notation-with-javascript-25mc.