Luana Taís Thomas
21200415-4
luana.thomas@edu.pucrs.br

Os três jarros do sheik
Trabalho 01 - Algoritmos e Estruturas de Dados II

15 de setembro de 2022

# 1 Resumo

Este relatório tem o objetivo de demonstrar, analisar e discutir o desenvolvimento de três soluções para o problema proposto no Trabalho 1 de Algoritmos e Estruturas de Dados II. O desafio era ajudar o sheik de um emirado distante a descobrir a quantidade mínima de movimentos necessários para mover a água contida em 3 jarros com capacidades diferentes até obter o resultado esperado.

Será apresentada uma possibilidade de solução implementada na linguagem de programação Java, através de pseudocódigos e link para o repositório do trabalho no GitHub, e também serão discutidas outras opções de solução que surgiram durante o desenvolvimento do trabalho.

# 2 Introdução

O problema que será resolvido foi disponibilizado pelo professor na página da disciplina e consiste no seguinte cenário:

> Você está prestando assessoria para o sheik de um emirado distante, que deseja que você resolva um problema que atormenta sua família há gerações: segundo a lenda, um ancestral do sheik recebeu de um gênio três jarros com água e devia mover a água de um para o outro até atingir quantidades estabelecidas pelo gênio. Isto nem sempre era possível, e por isso o sheik é atormentado por dúvidas. Agora o problema foi passado para você, e você deve fazer o que puder, sempre respeitando as regras originais do problema:
>
> - É proibido jogar água fora.
> - É proibido pegar água de uma fonte.
> - Você só pode esvaziar um jarro em outro ou completar o outro jarro até a borda.

O gênio nos forneceu diversos casos de teste com todas as informações necessárias para resolver este problema: a capacidade de água suportada por cada jarro, a quantidade de água que existe atualmente em cada jarro e também o resultado esperado após as movimentações. Nos pseudocódigos apresentados nas próximas seções, serão utilizadas as palavras capacidade, contido e desejado para mencionar os valores que cada jarro possui.

Para auxiliar o sheik com este desafio, nossa missão é desenvolver um algoritmo que descubra a quantidade mínima de movimentos para obter o resultado esperado de forma automatizada. Na seção 3, será apresentada a solução encontrada para resolver o problema e também serão discutidas outras ideias que surgiram durante o desenvolvimento trabalho. Na seção 4, serão mostrados os resultados obtidos através dos casos de teste disponibilizados e na seção 5, um breve conclusão sobre o trabalho.

# 3 Solucionando o problema

Os dados fornecidos pelo gênio serão lidos pelo programa através de três linhas. Na primeira linha, são informadas as capacidades de cada jarro. Na segunda linha, é informado o valor contido atualmente e na terceira linha, é informado o valor desejado pelo gênio para cada jarro. Uma ilustração da entrada de dados pode ser vista na figura 1.

| | | | |
|---|---|---|---|
| Capacidade | 6 | 10 | 15 |
| Valor contido | 5 | 7 | 8 |
| Valor desejado | 0 | 5 | 15 |

Figura 1: Caso de teste - exemplo enunciado

Analisando o enunciado do problema, uma das primeiras conclusões que podemos obter é de que há exatamente seis movimentos possíveis para transferência de água entre os jarros:

- Jarro 1 transfere para o jarro 2
- Jarro 1 transfere para o jarro 3
- Jarro 2 transfere para o jarro 1
- Jarro 2 transfere para o jarro 3
- Jarro 3 transfere para o jarro 1
- Jarro 3 transfere para o jarro 2

Entretanto, existem alguns pontos que devem sem notados antes de transferir a água de um jarro para o outro. Por exemplo, não podemos transferir quando o jarro de destino já se encontra cheio ou quando o jarro de origem já se encontra vazio. Cuidados também devem ser tomados ao transferir para um jarro que não possui capacidade total para a quantidade de água contida no jarro de origem. Desta forma, foi desenvolvido o método que é apresentado em pseudocódigo no Algoritmo 1:

**Algorithm 1** boolean transfere(Jarro origem, Jarro destino)

```
if destino.isJarroCheio then return False
end if
if origem.contido = 0 then return False
end if
if destino.capacidade − destino.contido < origem.contido then
    sobra ← |destino.capacidade − destino.contido − origem.contido|
    destino.contido ← destino.contido + origem.contido − sobra
    origem.contido ← sobra
    return True
end if
if destino.capacidade − destino.contido >= origem.contido then
    destino.contido ← destino.contido + origem.contido
    origem.contido ← 0
    return True
end if
```

Aplicando os movimentos descritos anteriormente no caso de teste inicial, seis novos estados poderiam ser criados, cada um contendo a quantidade de água de cada jarro após os movimentos de transferência, respeitando os critérios do algoritmo 1. Um exemplo dos seis novos estados pode ser visto na Figura 2.

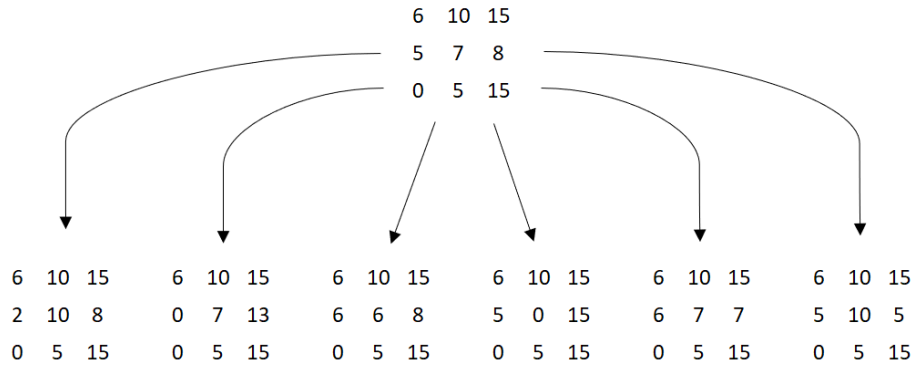


Figura 2: Caso de teste - novos estados

Para cada novo estado, os seis movimentos de transferência podem ser aplicados novamente, e assim sucessivamente, até encontrar a quantidade de água desejada pelo gênio ou até que não seja possível transferir água para outro jarro gerando um novo estado. Caso isso aconteça, seria impossível encontrar a quantidade de água desejada e o caso de teste tem como resultado “Impossível!”

A partir deste momento, surgiram três ideias de solução para o desafio.

## 3.1 Primeira solução

Inicialmente, foi pensado em resolver o problema utilizando algum tipo de estrutura de lista, como por exemplo ArrayList (java.util.ArrayList). A ideia era armazenar os estados obtidos a partir dos movimentos em uma lista, até chegar no resultado esperado.

Para não repetir nenhum estado, seria necessário sempre consultar a lista, e caso este estado já estivesse salvo, deveria ser ignorado. Seria também necessário um contador para verificar quantos movimentos de transferência já foram realizados e assim retornar a menor quantidade possível de movimentos, conforme o gênio desejava.

Depois de pensar um pouco mais sobre essa solução, percebe-se que ao invés de utilizar a estrutura de listas, uma estrutura de árvores faria mais sentido. E assim vamos para a segunda solução.

## 3.2 Segunda solução

Como descrito anteriormente, ao aplicar os seis movimentos de transferência no caso de teste, obteremos no máximo seis novos estados. A partir de agora, vamos imaginar que o caso de teste é o nodo 0 de uma árvore. Este nodo teria seis filhos, os nodos 1, 2, 3, 4, 5 e 6. Seguindo este raciocínio, o nodo 1 teria como filhos os nodos de 7 a 12, o nodo 2 teria como filhos os nodos de 13 a 18 e assim sucessivamente. Uma ilustração desta árvore pode ser vista na Figura 3. Em cada nodo é armazenada uma lista com os valores atuais de água em cada jarro.

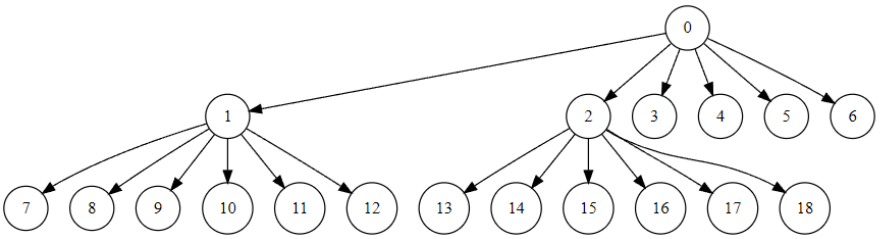


Figura 3: Exemplo de árvore que ilustra o problema dos jarros do sheik

Para cada novo nodo criado em um nível, devem ser aplicados os seis movimentos. Desta forma, só poderá passar para o próximo nível da árvore quando não há mais nada para fazer no nível atual, garantindo que seja encontrada a menor quantidade de movimentos necessários para obter as quantidades de água desejadas pelo gênio.

Após realizar todos os movimentos necessários e encontrar algum nodo que possua a quantidade desejada pelo gênio, é necessário apenas calcular em qual nível o nodo se encontra para saber quantos movimentos foram necessários. Isso é possível através do método apresentado no Algoritmo 2.

**Algorithm 2** int nivel(Integer nodo)

```
nodoAux ← encontraNodo(nodo, root)   ▷ Método para percorrer a árvore até encontrar o nodo
if nodoAux = null then return -1     ▷ Não achou o elemento
end if
cont ← 0
while nodoAux! = root do
    nodoAux ← nodoAux.pai
    cont++                           ▷ Incrementa o contador toda vez que sobe de nível e vai para o nodo pai
end while
return cont                          ▷ cont possui a quantidade de movimentos necessários para obter a combinação
```

Se essa solução fosse implementada, já seria possível ajudar o sheik a resolver os desafios propostos pelo gênio, porém após a aula em que o professor ensinou a estrutura de Heaps, outra ideia surgiu. E assim, vamos para a terceira e última solução.

## 3.3 Terceira solução

Na estrutura de Heaps, utilizamos uma lista para armazenar a ideia de uma árvore, acessando os filhos e o pai de um elemento através dos seguintes métodos:

- private int left (int i) { return 2 * i + 1; }
- private int right (int i) { return 2 * i + 2; }
- private int parent (int i) { return (i-1) / 2; }

Observe a figura 4. Para encontrar o filho à esquerda do nodo 6, chamamos o método left(1). O valor 1 passado por parâmetro corresponde à posição do nodo 6 na lista.

A chamada do método retorna o valor 2 * 1 + 1 = 3, que corresponde à posição do filho à esquerda na lista. O elemento que está na posição 3 da lista é o 4. Ao observar a ilustração da árvore, comprovamos que 4 é o filho à esquerda de 6. O mesmo vale para os métodos right e parent.

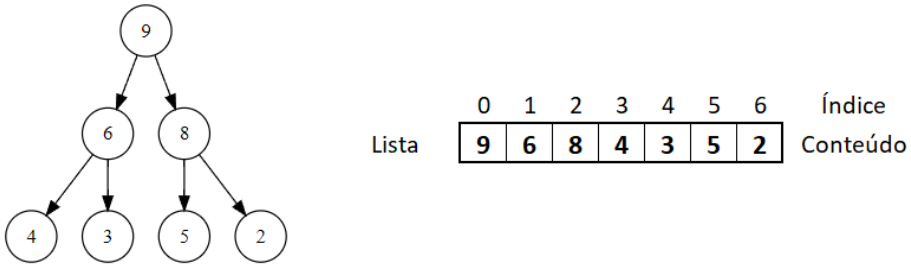


Figura 4: Exemplo de estrutura de árvore armazenada em lista

Em uma árvore com seis nodos, precisamos de mais métodos, porém poucas alterações são necessárias:

- private int filho1 (int i) { return 6 * i + 1; }
- private int filho2 (int i) { return 6 * i + 2; }
- private int filho3 (int i) { return 6 * i + 3; }
- private int filho4 (int i) { return 6 * i + 4; }
- private int filho5 (int i) { return 6 * i + 5; }
- private int filho6 (int i) { return 6 * i + 6; }
- private int pai (int i) { return (i-1) / 6; }

A principal diferença entre a implementação de lista para armazenar a ideia de uma árvore utilizada na estrutura de Heaps e a implementação que será sugerida aqui, é que ao invés de armazenar a árvore em uma lista, ela será armazenada em um dicionário.

A justificativa para tal alteração é a economia de memória. Toda vez que encontramos um estado repetido durante os movimentos de transferência descritos no Algoritmo 1, não é necessário armazená-lo. Isso implicaria em uma lista com inúmeros espaços em branco que estariam apenas ocupando memória. Desta forma, utilizando uma estrutura de dicionário chave-valor, guardamos a posição do elemento da árvore como chave e o estado da quantidade de água em cada jarro como valor.

Mas em relação à segunda solução proposta anteriormente, quais são as melhorias mais significativas? Não será mais necessário percorrer a árvore toda para encontrar a referência do nodo que contém a quantidade mínima de movimentos, como foi visto na primeira linha do Algoritmo 2!

Para descobrir quantos movimentos foram necessários até encontrar o resultado desejado pelo gênio, é necessário chamar o método pai(i), passando no parâmetro i a chave/posição do nodo que contém o resultado esperado, e repetir a chamada do método até que o valor retornado seja 0. A quantidade de vezes que o método foi chamado é o nível em que aquele nodo está localizado, ou seja, quantidade de movimentos que foram necessários. Este processo pode ser observado no método do Algoritmo 3.

**Algorithm 3** int nivel(long posicao)

```
cont ← 0
i ← posicao
while i > 0 do
    i ← pai(i)
    cont++
end while
return cont
```

Para visualizar a implementação completa desta solução, acesse o repositório do GitHub no link Problema dos Três Jarros - Luana Thomas[1].

[1] https://github.com/luanatthomas/tresJarros

# 4 Resultados

Após a implementação do algoritmo descrito na solução 3 em linguagem de programação Java, os casos de teste fornecidos pelo professor foram executados e a quantidade mínima de movimentos para atingir o resultado desejado pode ser vistos na Figura 5. Quando não é possível resolver o problema, aparece uma mensagem informando que ele é impossível.

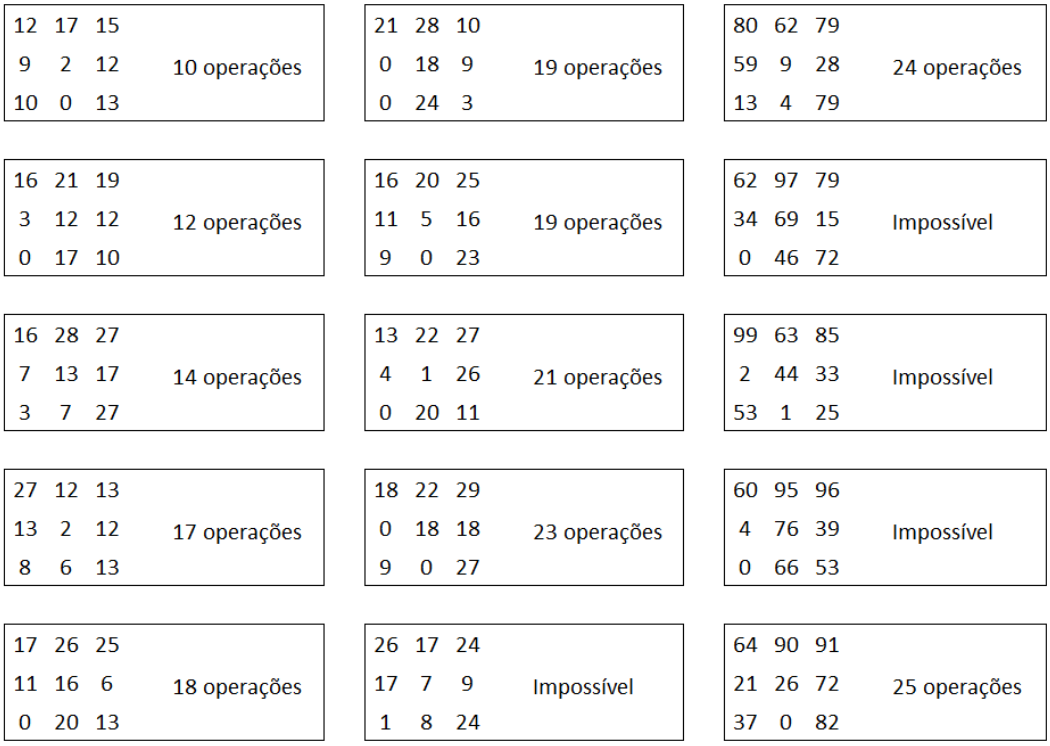


Figura 5: Casos de teste

Depois de executados os casos de teste, os resultados também foram conferidos com outros colegas que desenvolveram soluções diferentes. Como os resultados obtidos foram os mesmos, assume-se que estejam corretos e o algoritmo implementado funciona conforme o esperado.

# 5 Conclusões

O desenvolvimento do trabalho gerou resultados muito satisfatórios do ponto de vista de aprendizado. Relatar as soluções iniciais foi de grande ajuda para justificar a implementação da terceira solução, porém acredita-se que ainda há aspectos que possam ser melhorados.

Em alguns casos de teste, obtivemos como resultado “Impossível”, mas não há como ter certeza de que aquele problema não tem solução. O que ocorre é que os números que representam as posições de cada nodo crescem de forma muito rápida, até chegar o momento em que não é possível representá-los, mesmo utilizando o tipo primitivo long, que possui representação de 8 bytes. Uma solução poderia ser utilizar BigInteger para representar as posições dos nodos.

Outra alternativa que pode ser considerada é utilizar a segunda solução que foi apresentada, envolvendo a estrutura de árvores. Como foi justificado, a principal diferença de tempo de execução entre as duas soluções está na busca pelo nodo que possui a menor quantidade de movimentos e corresponde ao resultado desejado. A implementação que utiliza a lógica de árvores armazenadas como listas (terceira solução) é mais eficiente nesta busca, porém se utilizarmos a implementação de árvore, os valores dos nodos podem ser identificadores únicos que seguem uma sequência e não precisam necessariamente conter sua posição, tornando o algoritmo capaz de processar casos de teste mais complexos. Como a busca é algo que ocorre apenas uma vez no algoritmo, não é algo que causaria tanto impacto de maneira geral. Entretanto, se o gênio desejasse executar milhares de casos de teste em sequência, seria necessário decidir o que é mais importante para o algoritmo: rapidez ou capacidade de processar casos de teste mais complexos.

Acredita-se que a solução desenvolvida atende os requisitos necessários e explora a estrutura utilizada para Heaps de uma maneira não tão óbvia, exercitando o lado criativo na resolução deste problema.