Os Robôs Dançam Quadrilha

Diogo Marques da Silva
Faculdade de Informática — PUCRS

09 de abril de 2025

Resumo

Este artigo aborda o desenvolvimento da primeira atividade avaliativa da disciplina Algoritmos e Estruturas de Dados II, 1ª semestre de 2025, que trata sobre a repetição de sequências numéricas seguindo uma determinada ordem de posicionamento. São apresentadas duas soluções para o problema e sua eficiência é analisada. A seguir as soluções desenvolvidas são mostradas.

Introdução

O problema proposto pela atividade, exige que os robôs sigam uma ordem de posições durante a dança, ordem esta que foi definida pelo robô mestre. Porém eles detestam repetir uma sequência de posições que já ocorreu. Para o contexto da disciplina de Algoritmos e Estrutura de Dados, o problema pode ser resumido assim: É dada uma sequência de números (ordem) baseado em quantos robôs estão participando. Cada rodada os robôs trocam de posição seguindo a ordem definida, ou seja, o robô vai olhar para a sequência e prosseguir para a sua nova posição. O nosso papel no problema é avisar ao robô mestre quando uma sequência se repetir.

Para melhor descrição do problema vamos fazer algumas considerações:

1. Um valor ***n*** que representa quantos robôs estão participando.
2. A ordem definida que será uma sequência de inteiros ***seq[n]***.

O enunciado informa que os robôs devem alternar de posição da seguinte forma:

- O robô mestre olha para a sequência e altera a posição dos demais robôs;

Exemplo:

Sequência = 2 0 1;

Robôs (sequência atual) = 1 2 0;

- O robô mestre vai indicar que o robô que está na posição 2, deve ir para a posição 0;
- O robô que está na posição 0, deve ir para a posição 1;
- O robô que está na posição 1, deve ir para a posição 2.

Ao final desta rodada os robôs estarão assim:

Robôs = 0 1 2;

Para resolver o problema proposto, analisaremos duas possíveis soluções, suas abordagens e diferenças. Optando pela alternativa mais eficiente, vamos entender a otimização utilizada. Em seguida os resultados obtidos serão apresentados, bem como as conclusões obtidas no decorrer da atividade.

Primeira solução

Para esta solução podemos pensar em anotar todas as sequências que já ocorreram e comparar todas elas com a atual sequência dos robôs. Esta abordagem vai funcionar perfeitamente, porém, apenas quando poucos robôs estão participando da dança.

Para esta solução, nós vamos precisar de uma maneira de armazenar todos estes dados, podemos utilizar, em ***C++***, ***vectors***, isso nos dá a capacidade de armazenar vetores de forma dinâmica, sem precisar se preocupar com o número de sequências possíveis para o ***n*** inserido. Agora com a ideia inicial, devemos pensar em uma maneira de efetuar as trocas dos robôs, em ambas as soluções eu utilizo da mesma, definindo um vetor ***temp[n]*** que será temporário para facilitar as trocas em cada rodada. *Uma implementação para esta ideia seria assim:*

```
enquanto(1) faça
    se i = n então   // significa que concluiu uma rodada.
        i ← 0;
        sequencias ← temp;
        para j ← 0 até n faça
            robôs[j] = temp[j];
        fim
        para k ← 0 até sequências.comprimento() faça
            se robôs = sequências[k] então
            quebra;
            fim
        fim
    fim
    para i ← 0 até n faça
        posição ← sequencia[i];
        temp[i] ← robôs[posição];
    fim;
fim;
```

Esta implementação efetua as trocas dos robôs, adiciona a nova sequência em um **vector** e compara cada sequência nova com as que já ocorreram. Este algoritmo funciona, mas é bem custoso, uma vez que nós precisamos passar por cada elemento de cada sequência que já ocorreu. Quando o ***n*** é pequeno a diferença não é tão grande, contudo, conforme o ***n*** aumenta, a diferença fica imensa. Vamos destacar alguns pontos para esclarecer:

- Para utilizarmos a ideia de “anotar” cada sequencia nós precisaríamos percorrer todas as anteriores para checar se alguma se repete;
- A implementação dessa solução necessita de adaptações para comparar e adicionar valores em ***vectors***;
- Será necessário a utilização de mais laços dependentes de ***n***.

Então, analisando como deveríamos implementar essa solução, podemos perceber que o algoritmo seria relativamente complexo e também custoso. Uma vez que laços em função do número de entradas (***n***) são muito impactantes na performance do algoritmo.

Devido esta complexidade e ineficiência nós decidimos não implementar um algoritmo para essa solução. Explicaremos de forma mais detalhada sobre os laços de comparação dos ***vectors*** adiante. Com isso, chegamos a conclusão de que a ideia de “anotar” todas as sequências não é muito promissora.

Segunda solução

Partindo de uma solução que funciona apenas para números pequenos de entrada, nós precisamos de uma abordagem mais direta e mais eficaz. Notando a ineficiência em guardar todas as sequências que já ocorreram, pensamos em uma outra solução para este problema, onde não será necessário “anotar” todas elas.

Durante os testes em papel, percebemos que a sequência que se repetia era sempre a própria sequência dada pelo robô mestre, ou seja, não importa quantos robôs estão participando, eles somente irão repetir quando eles estiverem nas posições iguais às definidas pelo robô mestre. Exemplificando:

- Sequência = 2 0 1
- Próxima sequência que se repete é: 2 0 1

Então, dessa forma nós concluímos que não é necessário guardar todas as sequências se já sabemos qual é a próxima que se repete. Com isso nós podemos deixar de lado a ideia de ***vectors*** e partir para uma implementação muito mais simples, apenas comparando a sequência atual com a sequência dada pelo robô mestre. Agora tendo uma implementação assim:

```
se i = n então
        i ← 0;
        para j ← 0 até n faça
                robôs[j] = temp[j];
        fim
        se robôs = sequência então
                quebra;
        fim
fim
```

Como mostrado no trecho de código acima, simplificamos a comparação, tirando o laço que repetia até ***sequencias.comprimento()***. Mesmo que o algoritmo não passe por todas as sequências possíveis de uma entrada ***n***, que no caso de ***n*** = 74 seria 74^{74} que dá um valor colossal, podemos perceber que cada rodada gera uma nova sequência e sabendo que ***sequencias.comprimento()*** se torna muito maior que ***n***, nós precisaríamos efetuar muitas operações e é aqui que a solução 2 tem vantagem.

Para esta solução funcionar, utilizamos uma função chamada ***veteq()*** que compara vetores e então retorna um bool (verdadeiro ou falso). Então, esta função utiliza de um laço até ***n*** para comparar as sequências informadas, retornando falso assim que algum número divergir e verdadeiro caso todos sejam iguais. A implementação seria assim:

```
se robôs = sequência então          →     se veteq(robôs, sequência, n) então
        quebra;                                   quebra;
    fim;                                      fim;
bool veteq(vet1[], vet2[], n)
    para i ← 0 até n faça
        se vet1[i] != vet2[i] então
            retorna falso;
        fim;
    fim;
retorna verdadeiro;
```

Destacamos esta função porque nela ocorre uma grande diferença de performance, nesta implementação ela vai ser chamada uma vez por rodada. Já para a solução anterior esta função seria chamada para cada sequência já ocorrida, a cada rodada. *Observe a tabela a seguir:*

Número de entradas (n)	74
Número de rodadas	3.233.232
Chamadas para solução 1	$\frac{(3.233.232 \times (3.233.232 + 1)}{2}$
Chamadas para solução 2	3.233.232

Agora pensando em operações da função **veteq()**:

- Executa um laço até ***n***, no pior caso;
- As operações que ela realiza serão: ***n x chamadas***;

Sendo assim:

	Solução 1	Solução 2
Número de entradas (n)	74	74
Operações	$74 \times \frac{3.233.232 \times (3.233.232 + 1)}{2}$ **= 3.867.903.188×10¹⁴**	74 x 3.233.232 **= 239.259.168**

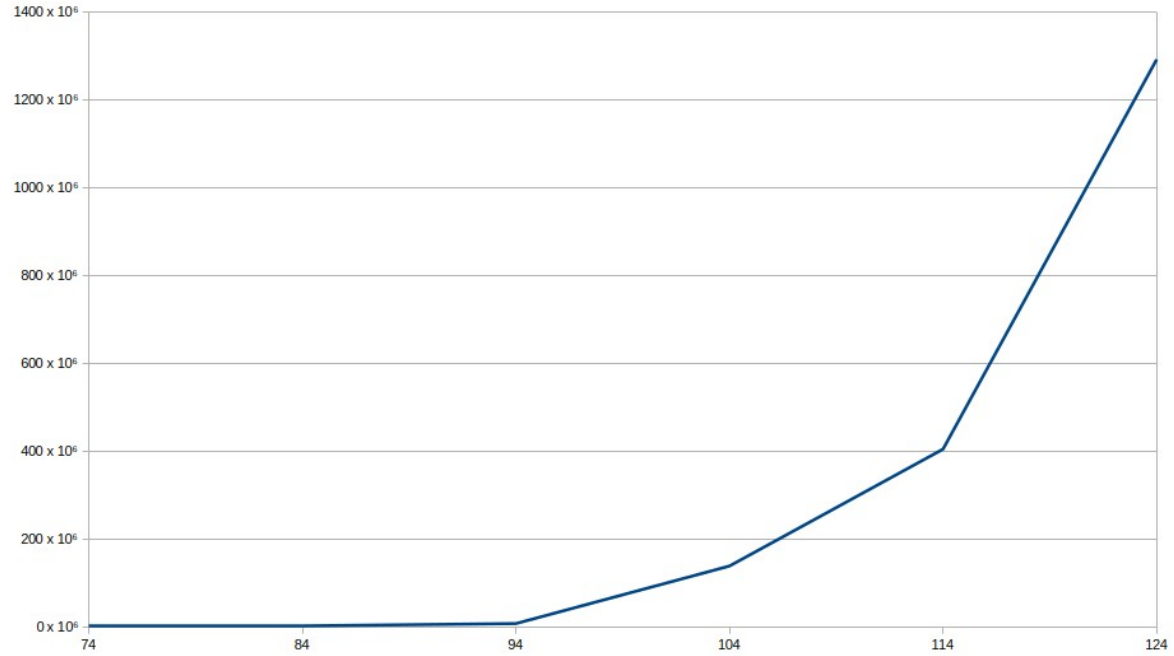
Então, concluímos que há uma grande diferença de operações da função **veteq()** *entre as duas soluções, tornando o algoritmo muito mais eficiente na solução 2, esta sendo a implementação que nós adotamos para realizar a atividade.*

Resultados

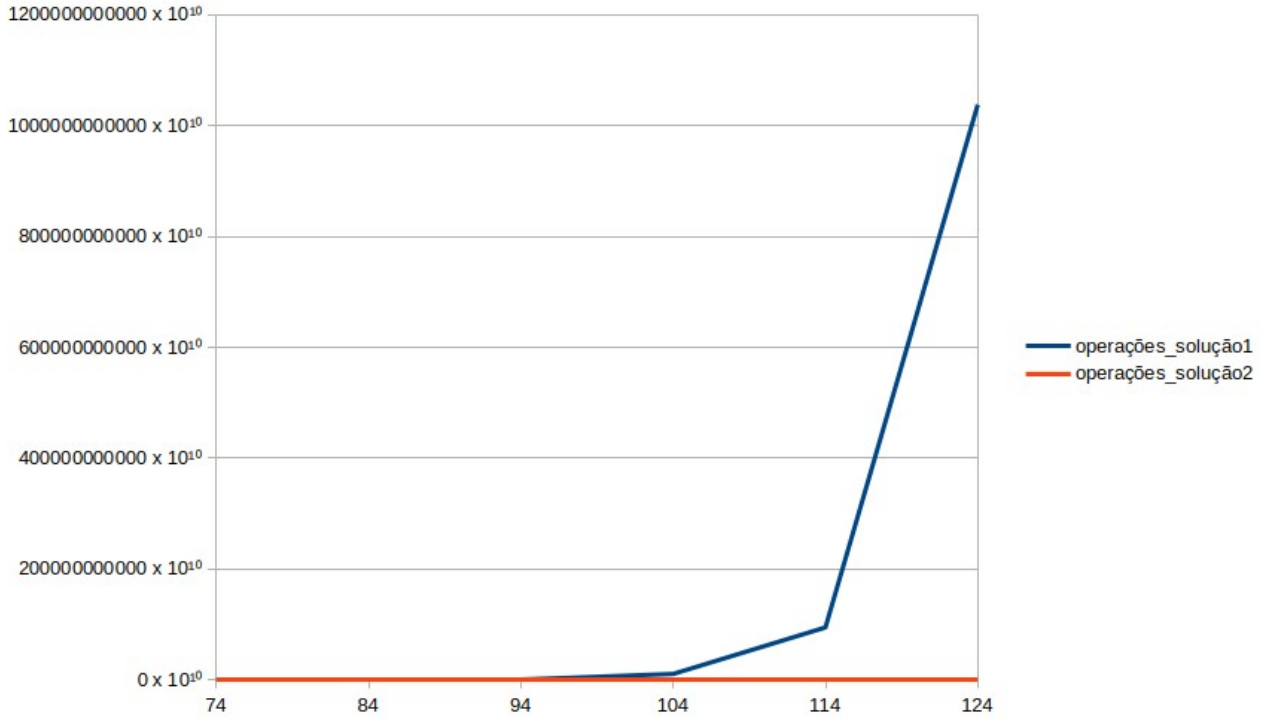
Depois de implementar o algoritmo acima em linguagem C++ e executá-lo, obtivemos os seguintes resultados:

Numero de entradas (**n**)	Tempo de Execução (segundos)	Rodadas
74	1.4	3.233.232
84	2.7	5.720.332
94	5	9.111.832
104	86	140.645.507
114	270	406.816.412
124	930	1.293.938.648

Podemos ver como esse crescimento das rodadas se comportam a partir deste gráfico:



*A*gora vamos tentar enxergar a diferença das operações realizadas em cada solução através do seu comportamento em gráfico:



Conclusões

A solução inicial, apesar de não parecer interessante nem mesmo de ser implementada, contribuiu para o entendimento do problema e para sua resolução de forma geral. Embora a solução adotada para a atividade ainda possa melhorar, já obtivemos ganhos consideráveis em relação à primeira solução proposta, fazendo com que a análise de ambas ajude a enxergar otimizações para este determinado problema. Também é destacável a simplicidade da mesma para implementação, fazendo com que o algoritmo se torne mais fácil de ser escrito e até mesmo lido.

Acreditamos ter desenvolvido uma solução com um simples implementação e de fácil entendimento para um problema que envolve grandes números de entrada e exige uma boa capacidade de desempenho, sendo assim, conseguimos entregar bons resultados com tempos de execução do algoritmo.

Referências

[1] https://cplusplus.com/reference/vector/vector/

[2] https://cplusplus.com/reference/fstream/fstream/

[3] https://cplusplus.com/reference/chrono/