MAC0121 relatório EP3

Victor Seiji Hariki

1 Ideias pré-código

1.1 Possibilidade de resolução

Primeiro foi necessário saber se um vetor poderia ou não ser ordenado. Isso foi simples, pois com três-rotações, apenas as peças em posições de mesma paridade poderiam ser trocadas entre si (Ímpares com ímpares e pares com pares). Como o vetor é circular, no caso do tamanho do vetor ser ímpar, ao fazer a rotação do fim para o começo do vetor, é possível a troca de um número em posição ímpar com um em posição par, e vice-versa. Assim, é possível ordená-lo em todo caso. Se for de tamanho par, existe a possibilidade de podermos ordená-lo, mas caso haja algum número que só poderia estar em sua ordem em uma posição de paridade diferente da em que está atualmente, é ímpossível a ordenação. Assim, o código de ordenação foi dividido para casos pares e ímpares, cada um com sua ideia e implementação escritas abaixo.

1.2 Algoritmo base de resolução

Para não ter que criar um algoritmo desde o 0, foi necessária a escolha de ul algoritmo base para a ordenação. Como teríamos que utilizar três-rotações para tal, escolhi o *bubblesort*, pelo fato dele executar apenas trocas com o próximo elemento, enquanto outros acabam por trocar elementos de locais 'distantes' no vetor.

1.3 Resolução com tamanho par

Como só é possível a troca com índices de mesma paridade, podemos apenas ordenar os pares e ímpares separadamente por *bubblesort*, ou, no caso, utilizar de uma única passada de um *bubblesort* modificado. Isso, claro, armazenando os movimentos feitos. Após isso, só é necessário verificar se o vetor está ordenado. Se não estiver, é ímpossível ordenar por três-rotações. Se estiver, já armazenei os movimentos que fiz para ordená-lo. A complexidade do algoritmo, nesse caso, é a mesma de dois *bubblesorts* de tamanho $\frac{n}{2}$, ou seja, $O(\frac{n^2}{2})$. No fim, simplificando, ainda sendo $O(n^2)$.

1.4 Resolução com tamanho ímpar

Nesse caso, é sempre possível a ordenação. Assim, usei a ordenação que user para os de tamanho par, para ordenar os índices ímpares e pares, respectivamente, E então usei mais um *bubblesort*, otimizado pra três-rotações, que pode trocar tanto números separados por uma posição quanto números adjacentes (requere exatamente n movimentos, $floor(\frac{n+1}{2})$ para a ida, e resto para a volta). A complexidade tem de considerar, assim, o *bubblesort* feito no começo, e, em média, n movimentos, multiplicado pela complexidade do *bubblesort* em questão, $O(\frac{n^2}{2})$. Assim, fica $O(\frac{n^2}{2} + n \cdot \frac{n^2}{2})$, que, simplificando, cresce em $O(n^3)$.

2 Implementação

Não há muito o que falar da implementação, que foi feita de acordo com as ideias descritas anteriormente. Gostaria apenas de destacar sobre a grande modularidade do código, com a stack sendo quase completamente reutilizada do EP2, facilitando o desenvolvimento apenas da lógica específica para esse EP, e o vetor circular.

3 Conclusões

Por testes, apesar do vetor ser pequeno, ainda foi encontrado um grande número de passos para sua resolução, talvez por não-otimidade do algoritmo escolhido, mas ainda assim bem considerável. Apesar disso, a complexidade não foi tão grande quanto esperava, quase rivalizando outros algoritmos bem conhecidos de ordenação. O algoritmo implementado, foi, por fim, não estável, com complexidade média e de pior caso de $O(n^3)$. Apesar disso, é difícil estimar o número de movimentos, visto que uma pequena mudança no vetor pode adicionar ou remover n movimentos a serem feitos em sua resolução.

4 Arquivos enviados

Foram enviados 3 arquivos compactados:

- *tresReversao.c* : Arquivo para avaliação. Compile-o por uso do *Makefile*.
- *types.h* : Arquivo contendo as *structs* e *typedefs*
- *cvector.** : Arquivos do código de manipulação do vetor circular.
- *stack.** : Arquivos do código de manipulação da pilha.
- *Makefile* : *Makefile* para a compilação da fonte e vinculação dos objetos. Use *make clean* para excluir os arquivos de objeto (**.o*)
- *Relatorio.pdf* : Este relatório.