Relatório do EP3

Juliano Garcia de Oliveira Nº USP: 9277086

17 de Outubro, 2016

1. Especificação do EP

O EP3 possui foco em algoritmos de ordenação. O propósito deste EP é: ler um vetor de tamanho n da entrada padrão, que é considerado um vetor circular, e verificar se ele é ordenável através de sequências de 3-reversão. Se for possível ordenar, então o programa terá que imprimir a sequência de 3-reversões que ordena o vetor. Deve ser impresso "Nao e possivel" caso contrário.

2. Algoritmo

O algoritmo implementado separa em vários casos de ordenação possíveis. A sequência das instruções (bem simplificada) é a seguinte:

- Se o tamanho do vetor original for **par**:
  - Faz um *bubbleSort* nas posições pares e ímpares do vetor original e verifica se foi possível ordenar.
    - Se é possível, ordena novamente e vai imprimindo as posições;
    - Senão, imprime "Nao e possivel" e finaliza a execução do programa;
- Senão o vetor é **ímpar**, então:
  - Ordena o uma cópia do vetor usando um ***heapSort***
  - Ordena o vetor original e imprime os movimentos, já que é possível a ordenação.

3. Ordenação

O algoritmo usa 2 matrizes para fazer a ordenação. As duas matrizes são a matriz ***v*** e a matriz ***s***. Ambas são matrizes ***2 x n***, sendo que a primeira linha de cada matriz guarda os números lidos na entrada padrão, e a segunda linha é auxiliar para a ordenação. A matriz ***v*** é a matriz do vetor **original**, i.e. o vetor lido na entrada padrão. A matriz ***s*** é a matriz do vetor **ordenado**.

- O vetor ***v[0]*** é o vetor lido na entrada padrão.
- O vetor ***v[1]*** representa a posição onde o elemento **i** do vetor ***v[0]*** está no vetor ordenado.
- O vetor ***s[0]*** é o vetor já ordenado.
- O vetor ***s[1]*** é o vetor que armazena a posição onde o elemento **i** do vetor ordenado está no vetor original. Darei exemplos para melhor clarificação abaixo.

- Vetor de tamanho **par**:

  Em um vetor de tamanho par, ao efetuar movimentos de 3-reversão, um elemento da posição **i** só pode chegar em posições de mesma paridade que **i**. Na prática, isto quer dizer que elementos que começaram em posição par no vetor original só podem chegar (através de 3-reversão) em posições pares. E a mesma coisa para os ímpares, que só podem chegar em posições ímpares. Por exemplo, considere o vetor:

  $n = 6$

  1 2 3 4 6 5

  Para ordená-lo seria necessário trocar o número *5* com o *6*, mas como a posição de paridade deles são diferentes, é impossível ordenar este vetor usando 3-Reversão. É fácil perceber, por exemplo, que o *5* só pode trocar com o *2* e *4*, ou seja, só com posições ímpares (considerando que o vetor começa a ser contado do *0*);

  Ou seja, na prática um vetor par só será possível de ordenar usando uma 3-reversão se e somente se, ao ordenar somente as posições ímpares do vetor, e depois as posições pares, o vetor final estiver ordenado. Inicialmente, meu teste para verificar se um vetor é ordenado era somente ordená-lo usando um *heapSort*, e verificar se as paridades das posições após a ordenação são as mesmas do vetor original. Porém, esse algoritmo tinha um problema quando havia números iguais, como por exemplo o vetor abaixo: $n = 4$

  4 5 4 4

  Em que após o *heapSort* a paridade dos índices podia dar diferente, embora esse vetor é ordenável usando 3-Rotações. Então, para contornar esse erro, poderia fazer de dois modos: Ordenar as posições pares e ímpares do vetor usando um *bubbleSort*, colocando os movimentos feitos em uma pilha. Ao final, verificar se o vetor final está ordenado. Se estivesse ordenado, imprimir todos os passos da pilha / fila. A segunda alternativa foi a que eu escolhi, que é ordenar o vetor usando o *bubbleSort*, verificar se ele está ordenado, se estiver, simplesmente rodar novamente um *bubbleSort* porém agora imprimindo os movimentos, assim não tenho que alocar mais um vetor com um tamanho grande para armazenar os movimentos feitos.

  Portanto, a complexidade neste caso é $\mathcal{O}(n^2)$, pois seriam dois *bubbleSort* no vetor inteiro de tamanho n.

  O número de comparações no pior caso de um *bubbleSort* é $\binom{n}{2}$. No caso do tamanho do vetor ser par, é necessário simplesmente ordenar as posições pares e as posições ímpares, ou seja, no pior caso, o número de movimentos será:

$$4 * \binom{\frac{n}{2}}{2} = 4 * \frac{\frac{n}{2}!}{2! * (\frac{n}{2} - 2)!} = 2 * \frac{\frac{n}{2} * (\frac{n}{2} - 1) * (\frac{n}{2} - 2)!}{(\frac{n}{2} - 2)!} =$$

$$= 2 * \frac{n}{2} * (\frac{n}{2} - 1) = 2 * \frac{n^2 - 2n}{4}$$

  Portanto o número máximo de movimentos será $\frac{n^2-2n}{2}$ no caso par.

- Vetor de tamanho **ímpar**:

  Em um vetor de tamanho ímpar, um elemento qualquer pode chegar em qualquer posição. Assim, usando um algoritmo correto, é fácil de perceber que qualquer vetor ímpar pode ser ordenado. usando 3-reversão.

  Nesse caso, ao invés de usar um *bubbleSort* do estilo de quando o vetor é par, para fazer uma melhor otimização do algoritmo, utilizo os endereços dos elementos para ordenar diretamente um elemento para sua posição ordenada. O vetor ***v[1]*** agora possui a posição onde o elemento i de ***v[0]*** está no vetor ordenado ***s[0]***. Isso quer dizer que eu sei exatamente onde um determinado elemento i deve estar após estar ordenado. Como exemplo, considere o seguinte exemplo:

$n = 5$

v[0] = 5 1 0 9 4

v[1] = 0 1 2 3 4 ***OBS***: Antes de ordenar, ***v[1]*** é igual ao caso quando o vetor tem tamanho par.

Copia-se este conteúdo para ***s***, e ele é ordenado usando um *heapSort*. Após a ordenação, temos:

s[0] = 0 1 4 5 9

s[1] = 2 1 4 0 3

Nesse momento, atualizamos o vetor ***v[1]***. Agora, este vetor armazenará a posição de um dado elemento i de ***v[0]*** no vetor ordenado ***s[0]***. A matriz ***v*** fica assim:

v[0] = 5 1 0 9 4

v[1] = 3 1 0 4 2

A ordenação no caso ímpar começa a preencher os elementos na última posição possível usando 3-Reversões a partir da posição do elemento que deve ficar em i após a ordenação. Na prática, isso significa que eu começo a preencher a posição $i = n - 2$ do vetor ***v[0]***, e em seguida preencho a posição $i = (i - 2) \mod 2$ do vetor, até que todas as posições sejam preenchidas com os números corretos.

Então , para cada posição i que precisa ser preenchida, obtenho a posição *pos* do elemento que deve ir para a posição i, usando vetor ***s[1]*** eu obtenho *pos*. Enquanto o elemento ***v[0][pos]*** não está na posição i, ele vai movendo usando a 3-Reversão (e o movimento é impresso ao mesmo tempo).Todas as vezes que é feita uma 3-Reversão, os valores de ***s[1]*** e ***v[1]*** são trocados também, para manter a corretude do algoritmo. A 3-Reversão é feita apenas para "frente".

OBS: Há um pequeno ajuste quando há números iguais e eles já estão na posição correta. Neste caso, eu simplesmente troco os valores de ***v[1]*** e ***s[1]***, não preciso trocar os elementos.

```
/* Posição a ser preenchida na iteração */
i = mod(n - 2 , n);

/* Preciso rodar no máximo n-1 vezes */
while(total > 1) {
/* Posição do elemento que deve ir para posição i */
    pos = s[1][i];
    flag = false;
    /* Enquanto não houver o número correto na posição i*/
    while (v[0][i] != s[0][i]) {
        flag = true;
        /* Faz a 3-rotação */
        swapElements(v, s, pos, mod(pos + 2, n));
        printf("%d\n", pos);
        pos = mod(pos + 2, n);
    }
    /* Caso seja necessário, faz a correção os índices */
     if(!flag)
        fixPos(v[1], s[1], i);
    i = mod(i - 2, n);
    total--;
}
```

Como o vetor tem n posições que precisam ser preenchidas, o algoritmo precisa olhar para todos os n elementos pelo menos uma vez (para colocá-lo no lugar certo). Diferentemente do *bubbleSort*,

este algoritmo move somente o elemento correto até sua posição, não precisa fazer comparações desde o primeiro elemento até o final.

Por exemplo, no pior caso, o primeiro elemento a ser verificado precisar percorrer todos os outros elementos, ou seja, precisa dar $n-1$ 3-Reversões para chegar no lugar correto. Porém, como faço os pulos diretamente com o elemento que deve ir para a posição i, o algoritmo fica mais otimizado. Nos testes que fiz, o número de movimentos sempre esteve abaixo de $\frac{n^2}{2}$.

3. Funções e estruturas de dados

O EP consiste de 2 arquivos principais (excluindo o(s) *header(s)*):

- *arrayOp.c*
- *tresReversao.c*

Como estrutura de dados, só utiliza-se matrizes simples.

As funções possuem nomes bem autoexplicativos e são divididas em escopos distintos. A explicação de cada função está mais detalhada nos comentários do próprio código *tresReversao.c*.

> tresReversao.c:

- **Sorting**
  - *sortArray( )* : Verifica se um dado vetor pode ser ordenado. Se puder, o ordena, senão retorna falso;
  - *bubble3( )* : Versão modificada do *bubbleSort*, que compara e faz movimentos de 3 em 3;
  - *bSortOdd( )*: Ordena um vetor de tamanho ímpar usando 3-reversão;
- **Movimentos no vetor**
  - *swapElements( )*: Troca dois elementos e todos os endereços que eles apontam (em ***v[1]*** e ***s[1]***);
  - *swapPos( )* : Apenas troca duas posições de elementos. É usado pelo função anterior;
  - *fixPos( )*: Conserta os endereços quando não houve troca de elementos;
  - *mod( )*: Implementa uma operação de *mod* similar a do Python;

As funções do arquivo *arrayOp.c* são funções para ordenar o vetor usando o *heapSort*, verificar se um vetor é igual a outro, verificar se um vetor foi alocado sem problema, e outras funções básicas que estão melhor explicadas no próprio código.

4. Otimização e resultados

A otimização foi descrita com mais detalhes na seção anterior. Resumidamente, é feito otimização no algoritmo de ordenação para vetores de tamanho ímpar, e detecta se o algoritmo já está ordenado nos casos de vetores de tamanho par, reduzindo a complexidade pela metade quando for possível.

É possível ordenar um vetor de qualquer tamanho com 3-rotações? R: Não. Um vetor ímpar sempre pode ser ordenado, mas um vetor par só pode ser ordenado se a paridade de seus elementos após a ordenação for a mesma de quando está na disposição original (Como a verificação que é feita com o *bubbleSort*).

Dados um vetor de tamanho n é possível ordenar qualquer instância com 3 rotações? R: Se n for ímpar, sim. Se n for par, não.

Qual o número máximo de 3-Rotações que seu algoritmo executa para ordenar um vetor? R: Se n for par, foi demonstrado acima que são $\frac{n^2-n}{2}$ trocas. Se n é ímpar, o número de comparações é (nos testes realizados e pela análise do algoritmo) menor que $\frac{n^2}{2}$, e a quantidade de trocas é na maioria dos casos próxima de $\frac{\binom{n}{2}}{2}$, mas no pior caso é um *bubbleSort* comum ($\binom{n}{2}$), mesmo que a frequência do pior caso seja bastante reduzida devido a otimização deste algoritmo.

Os testes realizados funcionaram bem, sendo que o maior teste que fiz foi com um vetor de 60001 elementos. Omitindo as funções $printf()$ do código, ele levou aproximadamente 11 segundos para ordenar o vetor, o que é um resultado muito bom, já que nesses 11 segundos foi feito o *heapSort* ***E*** a ordenação de 3-Reversão! Para ter uma melhor ideia, para ordenar esse mesmo vetor usando apenas um *bubbleSort*, levou 14 segundos e mais movimentos para executar. Ou seja, foi mais rápido executar 2 algoritmos de ordenação do que o *bubbleSort* comum.

Fontes: Bubble Sort, Wikipedia. Disponível em https://en.wikipedia.org/wiki/Bubble_sort

Sorting and Algorithm Analysis, Harvard Extension School. Disponível em

http://www.fas.harvard.edu/~cscie119/lectures/sorting.pdf