Victor Eduardo Requia *Trabalho EDA - Algoritmos de ordenação*

25/09/2020 *Gilmario Barbosa dos Santos*

1-Quick-sort (QS);

O algoritmo consiste em escolher um elemento pivô “x” (podendo ser qualquer valor em qualquer posição). Assim que escolhido, x é trocado de lugar com o último elemento do vetor. Então, primeiramente, é usado o vetor da posição inicial ‘j’ até n (tamanho do vetor) -1. Com isso vamos avançando a posição j até encontrarmos um valor maior ou igual ao do pivô e regredindo a posição n-1 até cruzarmos com o vetor j ou encontrarmos um valor menor que do pivô. Caso acharmos um elemento maior pelo j e um menor pelo n-1, trocamos os valores de lugar e então fazemos o mesmo processo para o próximo elemento de j e o anterior de n-1 até x = j. Quando forem iguais, veremos que ao lado direito do elemento de encontro, só teremos valores menores ou iguais a de x, o pivô e a esquerda maiores. Com isso, iremos trocar o pivô com o elemento que está na posição x e então x estará na posição correta da ordenação. Após ordenar x, iremos dividir o vetor em outras 2 partes, uma com todos os elementos a esquerda de x e outra com todos a direita e então repetimos o processo de forma recursiva com esses subvetores até classificarmos o vetor por inteiro.

Exemplo:

Seja o vetor 10 9 8 7 6 5 4 3 2 1 e o elemento pivô escolhido inicialmente é 6.

Vamos trocar (6) de lugar com o último elemento (1) -> [10, 9, 8, 7, 1, 5, 4, 3, 2, 6]

Vamos analisar o vetor a direita de 6 -> [10, 9, 8, 7, 1, 5, 4, 3, 2]

Vamos partir da primeira posição x [0] até encontrarmos um elemento maior ou igual a 6(pivô). Neste caso x [0] = 10, é maior que 6 então encontramos. Faremos a mesma coisa com o último elemento, porém, procurando um elemento menor que 6. Partimos de y [n (número de elementos do “subvetor” analisado) -1] = y [8] = 2 e como 2 é menor que 6 então encontramos. Com isso, trocaremos x e y de lugar -> [2, 9, 8, 7, 1, 5, 4, 3, 10]

Faremos o mesmo procedimento porém, agora x será o próximo elemento de onde ocorreu a troca de x , ou sejam x [y+1] e seguindo o mesmo raciocínio, y [x-1] -> x [y+1] = x [0+1] = x [1] e y [x-1] = y [8-1] = y [7]

Partindo de x [1] = 9, o próximo maior ou igual elemento que 6(pivô) será ele próprio. Assim como partindo de y [7] = 3 será o elemento anterior menor que 6. Assim, trocamos x e y de lugar -> [2, 3, 8, 7, 1, 5, 4, 9, 10,]

Seguiremos com essa mesma logica até chegar no ponto em que x = y -> [2, 3, 4, 5, 1, 7, 8, 9, 10] (nessa ordenação, teremos x [a] = y [a], onde a = 5 e x [5] = 7)

Vamos pegar o a posição do vetor e trocar de lugar com o pivô -> [2, 3, 4, 5, 1, 6, 8, 9, 10, 7]

Repare que os elementos a direita do pivô são menores do que ele e a esquerda maiores. Assim, iremos fazer uma subdivisão (particionamento) desse vetor em outras duas outras partes. Uma será com os elementos a direita do pivô e outro a esquerda -> [2, 3, 4, 5, 1] 6 [8, 9, 10, 7]

Vamos repetir o procedimento inicial com ambos os vetores subdivididos. Primeiramente vamos desenvolver os passos com o vetor a direita do pivô, ficando

[2, 3, 4, 5, 1] -> vetor inicial. O elemento pivô será o 4

[2,3,1,5,4] -> trocando o pivô com a última posição do vetor

[2,3,1,5,4] -> procurando da esquerda para a direita um elemento maior que 4. Nesse caso, apenas o último elemento do vetor (sem contar o pivô) é o elemento maior ou igual a 4. Ou seja, a posição x é igual a de y. Iremos trocar x com o pivô -> [2, 3, 1, 4, 5]. Com isso o pivô estará na ordem correta. Iremos particionar o vetor que está sendo ordenado em outras duas partes. Uma a direta do pivô e outra à esquerda -> [1,3,1] 4 [5]

Seguiremos com a mesma lógica até ordenarmos todos os valores, no final, o vetor ordenado será -> [1, 2, 3, 4, 5]

Agora temos todo o vetor que estava à esquerda do primeiro pivô 6 ordenado. Faremos os mesmos passos com o vetor a esquerda e chegaremos em [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] que é o vetor ordenado.

Algoritmo utilizado:

Função de particionar:

```
int partition ( int A[],int start ,int end) {

    int i = start + 1;

    int piv = A[start] ;

    for(int j =start + 1; j <= end ; j++ )   {

          if (A [j] < piv) {

                swap (A [i], A [j]);

            i += 1;

        }

  }
```

```
    swap ( A [start ] ,A[ i-1 ] ) ;
    return i-1;
}
```

Função recursiva de cada partição:

```
void quick_sort ( int A[ ] ,int start , int end ) {
    if( start < end ) {
            int piv_pos = partition (A,start , end ) ;
            quick_sort (A,start , piv_pos -1);
            quick_sort ( A,piv_pos +1 , end) ;
    }
}
```

2 – Merge-Sort (MS)

O algoritmo Merge-Sort tem o princípio parecido com o do Quick-Sort de divisão e conquista. Porém, este, tem um procedimento diferente. Primeiramente, o vetor é dividido ao meio, e o processo se repete com todas as outros subvetores até que se tenha um vetor com apenas um elemento, após isso, o vetor constrói-se novamente de forma classificada com o processo inverso. Exemplo:

Seja o vetor [7, 6, 5, 4, 3, 2, 1, 0]

Primeiramente iremos dividir o vetor em duas partes -> [7, 6, 5, 4] [3, 2, 1, 0]

Vamos dividir novamente esses vetores (Demonstração apenas com o da direita, porém, o processo é exatamente o mesmo com o da direita) -> [7, 6] [5, 4]

Novamente iremos dividir estes vetores -> [7] [6] [5][4]

Agora que temos apenas um elemento por vetor, iremos fazer o processo inverso, porém classificando os valores dos vetores unitários de dois em dois -> [6, 7] [4, 5]

Vamos fazer o mesmo processo, porém, comparando os dois vetores e colocando a ordem correta em um vetor com 4 posições -> [4, 5, 6, 7]

Faremos o mesmo processo com o vetor da esquerda e encontraremos o vetor -> [0, 1, 2, 3]

Para finalizar, comparamos o vetor da direta com o da esquerda e colocamos a ordem correta em um vetor do tamanho do origina, ficando -> [0, 1, 2, 3, 4, 5, 6, 7], ou seja, o vetor original ordenado

Algoritmo utilizado:

```
void merge(int A[ ] , int start, int mid, int end) {

int p = start ,q = mid+1;

int Arr[end-start+1] , k=0;

for(int i = start ;i <= end ;i++) {

      if(p > mid)

          Arr[ k++ ] = A[ q++] ;

    else if ( q > end)

          Arr[ k++ ] = A[ p++ ];

    else if( A[ p ] < A[ q ])

         Arr[ k++ ] = A[ p++ ];

    else

         Arr[ k++ ] = A[ q++];

 }

   for (int p=0 ; p< k ;p ++) {

       A[ start++ ] = Arr[ p ] ;

   }

}


void merge_sort (int A[ ] , int start , int end )

    {

                if( start < end ) {

                int mid = (start + end ) / 2 ;

                merge_sort (A, start , mid ) ;
```

```
            merge_sort (A,mid+1 , end ) ;


           merge(A,start , mid , end );
    }
}
```

3 - Heap-Sort

O Heap-Sort utiliza uma estrutura de dados chamada heap (Para melhor entendimento, a estrutura heap pode ser visualizada como uma espécie de árvore binária mantida na forma de um vetor). Para ordenar os elementos à medida que insere os elementos na estrutura. Assim ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada (O heap é gerado e mantido no próprio vetor a ser ordenado). Para uma ordenação crescente, deve ser construído um heap máximo (o maior elemento fica na raiz). Para uma ordenação crescente, deve ser construído um heap mínimo (o menor elemento fica na raiz).

Algoritmo utilizado:

```
    void heap_sort(int Arr[ ])
    {
        int heap_size = N;


        build_maxheap(Arr);
        for(int i = N; i >= 2 ; i-- )
        {
            swap|(Arr[ 1 ], Arr[ i ]);
            heap_size = heap_size - 1;
            max_heapify(Arr, 1, heap_size);
        }
    }
```

4 - Bubble-Sort (BS)

O algoritmo consiste em percorrer uma sequência várias vezes, comparando cada elemento (x) com o próximo (x+1). Na primeira passagem, a ordenação é feita da seguinte maneira. O primeiro elemento é comparado com o sucessor, se o sucessor for "menor" a troca é feita com ajuda de uma variável auxiliar e então o segundo é comparado com o terceiro, o terceiro com o quarto e assim até o último elemento. Deixando o "maior" elemento na última posição. Assim que a primeira passagem termina, a segunda passagem é acionada, fazendo o mesmo procedimento. O primeiro elemento é comparado com o sucessor.... Ao final da segunda passagem, o segundo maior elemento é colocado na penúltima posição e assim por diante.

Exemplo:

Seja o vetor 55 23 48 45 26 85 92 15 20 45

Primeira passagem:

x[0] com x[1] (55 com 23) -> realiza a troca

x[1] com x[2] (55 com 48) -> realiza a troca

x[2] com x[3] (55 com 45) -> realiza a troca

x[3] com x[4] (55 com 26) -> realiza a troca

x[4] com x[5] (55 com 85) -> nenhuma troca

x[5] com x[6] (85 com 92) -> nenhuma troca

x[6] com x[7] (92 com 15) -> realiza a troca

x[7] com x[8] (92 com 20) -> realiza a troca

x[8] com x[9] (92 com 45) -> realiza a troca

Assim, após a primeira passagem, a ordem será:

23 48 45 26 55 85 15 20 45 92

Observe que o maior elemento da sequência (92) ficou na última posição. Agora será realizada a segunda iteração.

O conjunto completo das iterações ficara assim:

Iteração 0 (conjunto original) (55 23 25 48 45 26 85 92 15)

Interação 1 (23 48 45 26 55 85 15 20 45 92)

Interação 2 (23 45 26 48 55 15 20 45 85 92)

Interação 3 (23 26 45 48 15 20 45 55 85 92)

Interação 4 (23 26 45 15 20 45 48 55 85 92)

Interação 5 (23 26 15 20 45 45 48 55 85 92)

Interação 6 (23 15 20 26 45 45 48 55 85 92)

Interação 7 (15 20 23 26 45 45 48 55 85 92)

Interação 8 (15 20 23 26 45 45 48 55 85 92)

É evidente que este algoritmo tem alguns aperfeiçoamentos. O primeiro deles é que, após a primeira iteração, o maior elemento estará na última posição, como demonstrado anteriormente. Com isso, a segunda interação não precisará comparar o penúltimo elemento com o último pois esse já sabemos que é o maior e não haverá troca. A mesma acontece com a terceira iteração, o elemento n-3 não irá precisar comparar com o elemento n-2. Sendo assim, na primeira passagem, são feitas n-1 comparações, na segunda passagem n-2 e na passagem (n-1) somente uma comparação é feita (entre x [0] e x [1]). O outro aperfeiçoamento é verificar se houve trocas durante a passagem, se não houver troca, o “vetor” estará ordenado e não precisará continuar as comparações, eliminando iterações desnecessárias e tempo de processamento. Para isso pode ser usada uma variável booleana que a cada passagem recebe o valor false e se houver troca o valor true

Algoritmo utilizado com os aperfeiçoamentos para a realização das iterações:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

int main(){

    int vetor[] = {55,23,25,48,45,26,85,92,15};

    int i=0, j=0;

    bool switched = true;


    for(i; i<n && switched == true; i++){

        switched = false;

        j=0;

        for(j; j<n-i-1; j++){
```

```
                    if(vetor[j]>vetor[j+1]){
                        switched = true;
                        int hold = vetor[j];
                        vetor[j] = vetor[j+1];
                        vetor[j+1] = hold;
                    }
                }
            }
};
```

Obs. a variável "n" nos laços for, precisa ser substituída pela quantidade de elementos que o vetor carrega.

Vantagens e desvantagem de cada algoritmo:

Ao abordarmos algoritmos de ordenação, não temos como definir qual é o melhor ou pior deles, quem deve decidir isso, é quem está programando. Isso se deve ao fato de cada algoritmo funcionar de maneira diferente dependendo do problema e dos recursos disponíveis principalmente com hardware. Apesar de não ser possível definir um padrão a ser usado, podemos descrever as vantagem e desvantagem de cada um.

Quick-Sort:

Vantagens:

É o algoritmo de ordenação mais rápido que se conhece para diversas situações (apesar de às vezes partições desequilibradas conduzirem uma ordenação lenta)

É extremamente eficiente para ordenar grandes arquivos de dados

Necessita de pouca memória para armazenar a pilha auxiliar

Requer O (n log n) comparações em média (caso médio) para ordenar n itens

Desvantagens:

Tem o pior caso de n^2 comparações

Implementação complexa em relação a algoritmos mais simples como o Bubble-sort

Merge-Sort

Vantagens:

Merge-Sort é O (n log n)

Indicado para aplicações que possuem restrição de tempo (executa sempre em um determinado tempo para n)

Possível ser transformado em estável (tomando cuidado na implementação da intercalação)

Fácil implementação

Desvantagem:

Utiliza memória auxiliar O(n)

Na prática é mais lento que o Quick-Sort no caso médio

Heap-Sort:

Vantagens:

Recomenda-se para aplicações que não podem tolerar eventualmente um caso desfavorável

Para dados imprevisíveis, podem ser mais vantajosos por serem previsíveis em termos de tempo de execução

Exige somente O (n log n) operações independentemente da ordem de entrada

Desvantagens:

O anel interno do algoritmo é bastante complexo se comparado com o do Quick-Sort

Não é estável

Construir a árvore-heap pode consumir muita memória

Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o heap

Bubble-Sort:

Vantagens:

O algoritmo é de fácil implementação e entendimento

Os elementos são trocados de posições apenas com uma variável auxiliar, não comprometendo o espaço de memória

Estável

Desvantagens:

Não é muito eficiente em termos de processamento x tempo

Não apresenta bons resultados quando a lista contém muitos itens. Isso ocorre, pois, esse tipo de ordenação exige n^2 passos de processamento para cada número n de elementos que serão ordenados

Melhor e pior caso

Quick-Sort:

Melhor caso: Ocorre quando o vetor é sempre dividido em vetores de tamanho igual após a partição. Exemplo: Vetor de 9 posições, estando o pivô ordenado na posição 5 e o vetor sendo particionado em 2 partes iguais (4 elementos a esquerda e 4 a direita) e assim sucessivamente. Com isso O (log n)

Pior caso: O pior caso é quando o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado. Nesse caso O(n). Este problema pode ser contornado escolhendo três itens quaisquer do vetor e usar a mediana dos três como pivô

Bubble-Sort:

A eficiência do método depende de que forma é implementado. Se for implementado da maneira sem as melhorias, a análise é. Existem n-1 passagens e n-1 comparações em cada passagem. Sendo assim, o número total de comparações é (n-1) *(n-1) = n^2-2n-1 que é n^2. Ou seja, O(n^2).

Com os aprimoramentos, o número de comparações na iteração i é n - i. Sendo assim, se existirem k iterações, o número total de iterações será (n - 1) + (TI - 2) + (n - 3) + ... + (n - k), que é igual a (2kn - k^2- k) /2. Podemos provar que o número médio de iterações, k, é O(n), de modo que a fórmula inteira é ainda O(n^2), embora o fator constante

seja menor do que antes. Entretanto, ocorre uma sobrecarga adicional ao testar e inicializar a variável switched (uma vez por passagem) e ao defini-la com TRUE (uma vez para cada troca).

O único recurso vantajoso da classificação por bolha é a exigência de pouco espaço adicional (variáveis auxiliares de troca) e o fato de ela ser O(n) no caso em que o arquivo está totalmente classificado (ou quase totalmente classificado). Isso se verifica ao observar que apenas uma passagem de n - 1 comparações (e nenhuma troca) é necessária para perceber que um arquivo está classificado

Bibliografia:

Estruturas de Dados Usando C (Aaron Ai Tenenbaum, Yedidyah Langsam, Moshe J. Augenstein)

Métodos de Ordenação Eficiente (Paulo Afonso, Valéria, Jadson)

Ordenação: QuickSort (Túlio Toffolo)

Estrutura de dados e algoritmos (Ricardo Farias)