RELATORIO DE ALGORITOMOS DE ORDENAÇÃO

Thiago Lima Rodrigues

Na tabela abaixo estão os resultados de tempo de execução dos algoritmos de ordenação (Bubble, Selection, Merge e Quick).

Com os valores de 100, 1000, 10000, 100000, 500000 e 1 milhão.

Com os resultados obtidos, o quick sort se mostrou o mais eficiente entre os todos.

E o bubble é o menor eficiente de todos eles.

	100	1000	10000	100000	500000	1000000
Bubble Sort	0,000s	0,000s	0,483s	51,169s	1305,246s	5389,634s
Selection Sort	0,000s	0,001s	0,109s	11,320s	285.916s	1173.555s
Merge Sort	0,000s	0,001s	0,005s	0,049s	0,256s	0,472s
Quick Sort	0,000s	0,000s	0,002s	0,016s	0,134s	0,267s

Bubble Sort é o pior dos ordenadores como podemos ver pelos números na tabela acima, e o principal motivo dele ser o pior é por conta do seu funcionamento, já ele compara os elementos de forma adjacente e quanto maior mais comparações terão e mais vezes terá que retornar ao começo do vetor fazendo o mesmo trajeto inúmeras vezes. O seu melhor caso é quando ele já está ordenado, aí nessa situação ele terá bom funcionamento, mas nosso caso onde os vetores estão desordenados o bubble se torna a pior opção para ser usado como algoritmo de ordenação.

Desempenho: O(n^2): quadratico

```c
void bubble(int *vetor, int size, FILE *arq)
{
    clock_t bInicio, bFinal;
    float bTotal;

    int troca = 0, varredura = 0, comparacao = 0;

    bInicio = clock();

      for(int t = 0; t < size; t++)
        *(vetor + t) = rand() % 100000;

    //POSICAO DE COMPARACAO COM O VETOR
    for (int i = 0; i < size - 1; i++)
    {
        //PERCORRE O VETOR SEMPRE COM UMA POSICAO A MENOS
        //ULTIMA POSICAO DE TROCA JA ESTA COM O MAIOR MAIOR VALOR
        for (int j = 0; j < size - 1; j++)
        {
            //CASO SEJA MAIOR VALOR TROCA COM O PROXIMO
            //NA ULTIMO POSICAO POSSIVEL FICARA O MAIOR NUMERO
           if (vetor[j] > vetor[j + 1])
           {
            swap(&vetor[j], &vetor[j + 1]);
            troca++;
           }
        comparacao++;
        }
    varredura++;
    }
bFinal = clock();
bTotal = ((float)(bFinal - bInicio) / CLOCKS_PER_SEC);
```

Exemplo de como funciona:

Inicial: [4] [1] [8] [2] [6] [3] [5] [7] comparar o 4 com 1 para ver quem e o maior assim:

1: [1] [4] [8] [2] [6] [3] [5] [7] comparar se o 4 e maior que 8, como é menor continua assim:

2: [1] [4] [8] [2] [6] [3] [5] [7] agora comparar o 8 com 2, assim:

3: [1] [4] [2] [8] [6] [3] [5] [7] comparar 8 com 6:

4: [1] [4] [2] [6] [8] [3] [5] [7] comparar 8 com 3:

5: [1] [4] [2] [6] [3] [8] [5] [7] comparar 8 com 5:

6: [1] [4] [2] [6] [3] [5] [8] [7] comparar 8 com 7:

7: [1] [4] [2] [6] [3] [5] [7] [8] e repetir tudo de novo com 0 comparando com 3 até ficar ordenado.

Selection Sort foi o segundo a ser testado e se mostrou muito melhor e mais eficiente que o Bubble com vetores de tamanho semelhante como mostrado na tabela. Basicamente seu funcionamento é sempre encontrar o menor elemento e colocá-lo na primeira posição e sua ordenação é fazer isso repetidas vezes para todo restante do vetor, e por isso ele perde eficiência em vetores maiores, como por exemplo o de 500000 e 1000000. Como foi dito para vetores pequenos ele é eficiente, mas já perde seu valor nos maiores vetores por isso no nosso caso fica atrás do Merge e Quick mas a frente do Bubble.

Desempenho: O(n^2): quadratico

```c
void selection (int *vetor, int size, FILE *arq)
{
  clock_t sInicio, sFinal;
  double sTotal;
  int troca = 0, varredura = 0, comparacao = 0;

  sInicio = clock();
  for(int t = 0; t < size; t++)
    *(vetor + t) = rand() % 100000;

    for (int i = 0; i < size; i++)
    {
        //armazena a posicao que sera comparada com o vetor
        int menor = i;
        //faz a comparacao com a posicao armazenada no menor
        for (int j = i; j < size; j++)
        {
            //caso o valor do vetor que percorre seja menor que o armazenado anteriormente a menor posicao já armazenada atualiza com a posicao
            if(vetor[j] < vetor [menor])
              {
                menor = j;
              }
            comparacao++;
        }
        //faz a troca dos valores "menor" atualizado e da posicao(em ordem 'i') que sera trocada caso a referencia('i') seja a mesma que o menor
        if(i != menor)
        {
          swap(&vetor[i], &vetor [menor]);
          troca++;
        }
    varredura++;
}
```

Exemplo do Selection:

Inicio: [3] [0] [7] [1] [5] [2] [4] [6] compara o menor valor com a primeira posição que é 3:

1: [0] [3] [7] [1] [5] [2] [4] [6] pegar o segundo menor valor que e 1 e trocar com 3:

2: [0] [1] [7] [3] [5] [2] [4] [6] pegar o terceiro menor valor que e 2 e trocar com 7:

3: [0] [1] [2] [3] [5] [7] [4] [6] pegar o quarto menor valor que e 3 e trocar com próprio 3:

4: [0] [1] [2] [3] [5] [7] [4] [6] pegar o quinto menor valor que e 4 e trocar com 5:

5: [0] [1] [2] [3] [4] [7] [5] [6] pegar o sexto menor valor que e 5 e trocar com 7:

6: [0] [1] [2] [3] [4] [5] [7] [6] pegar o sétimo menor valor que e 6 e trocar com 7:

7: [0] [1] [2] [3] [4] [5] [6] [7] até ordenar dessa forma.

Merge Sort é um algoritmo de ordenação de grande eficiência, ele funciona com um sistema de 'Divisão e Conquista', ele tem uma eficiência igual para todos os casos (melhor, médio e pior). Seu funcionamento é na base da divisão do vetor principal "quebrando" ele em partes menores, depois ele ordena essas partes menores e vai juntando as partes ordenadas, até que o vetor "quebrado" estará inteiro. Ele é estável e mais eficiente que os outros dois apresentados anteriormente.

Desempenho 0(n log n) quasilinear

```c
int mergesort(int *vetor, int inicio, int fim, dados *dado)
{
  int meio;

    if(inicio < fim)
    {
        //para toda vez que uma instancia dessa funcao e chamada
        //ele gera um novo meio e inicia novamente a recursividade
        meio = inicio + (fim - inicio)/2;
        mergesort(vetor, inicio, meio, dado);
        mergesort(vetor, meio+1, fim, dado);

        //quando inicio = fim a funcao termina sua instancia e retorna para instancia anterior
        //nesse retorno o segundo mergesort e chamado realizando o mesmo caminho da linha acima
        //terminando as duas metades o merge e chamado realizando assim a ordenacao entre as duas mergesort dessa instancia
        //o processo se repete ate todas as instancias serem realizadas e terminar o primeiro mergesort chamado, com o vetor ordenado

        merge(vetor, inicio, meio, fim, dado);
    }
}
```

Exemplo da divisão:

DIVIDIR

Esquerda m+1 direita

(a) inicio[3 0 7 1] [5 2 4 6]fim

(b) inicio[3 0][7 1]fim início[5 2][4 6]fim

Exemplo da Conquista:

CONQUISTA

(c) (pegar o (b) e comparar quem é o maior e coloca o menor sempre na esquerda)

[0 3] [1 7] [2 5] [4 6]

(d) (pegar o (c) e compara para ver quem é o menor)

[0 1 3 7] [2 4 5 6]

(e) (ordenar o (d) e ter o vetor todo ordenado)

[0 1 2 3 4 5 6 7]

Quick Sort: é o algoritmo mais eficiente de ordenação. No seu funcionamento há um pivô que se posiciona no array de uma maneira em que todos elementos menores ou igual a do pivô fique a sua esquerda e todos os maiores ou igual na sua direita. Aqui um exemplo na pratica de como funciona o algoritmo mais eficiente feito por nos nesse trabalho como mostra os resultados na tabela:

Desempenho 0(n log n) quasilinear

```
void quicksort(int *vetor, int inicio, int fim, dados *dado)
  {
  if(inicio < fim)
    {
    int pivo = quick(vetor, inicio, fim, dado);
    quicksort(vetor, inicio, pivo - 1, dado);
    quicksort(vetor, pivo + 1, fim, dado);
    }
  }
```

[3] [0] [7] [1] [5] [2] [4] [6] o pivô principal é o [3], assim:

(a) [0] [1] [2] [3] (b) [7] [5] [4] [6]

No (a) o pivô é o [0], mas como já está ordenado não existe necessidade de mudança. Já no (b) o pivô é o [7], assim:

(c) [5] [4] [6] [7]

No (c) o pivô é o [5], assim:

[4] [5] [6] [7]

Com isso juntando o (a) e o (c) fica:

[0] [1] [2] [3] [4] [5] [6] [7]