MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PIAUÍ – UFPI
CAMPUS SENADOR HELVÍDIO NUNES DE BARROS – PICOS
CURSO BACHARELADO EM SISTEMAS DE INFORMAÇÃO
PROFESSOR(A): LUANA BATISTA DA CRUZ
DISCIPLINA: ESTRUTURAS DE DADOS I

ANDRESSA CAROLINE DA ROCHA PEREIRA
ANTONIO IGOR MOREIRA PAIXAO
JOSE FILHO RIBEIRO DO NASCIMENTO
JOSE WANDERLEI FRANCISCO DE SOUSA ROCHA
PEDRO NICOLAS CAMPOS FERREIRA
VINICIUS DE SOUSA CARVALHO

RELATÓRIO SOBRE O ALGORITMO DE ORDENAÇÃO MERGE SORT

Picos
2022

1. Introdução

O presente trabalho tem como foco principal apresentar o algoritmo de ordenação Merge Sort, visando compreender seu princípio de funcionamento, processo de ordenação, bem como apresentar sua implementação na linguagem C.

2. Dividir para conquistar

A Merge Sort surge da técnica de divisão em conquista, que consiste em 3 passos:

Divisão: a divisão é o primeiro processo executado, na qual pegamos o nosso problema e dividiremos em subproblemas menores, visando conseguir uma melhor visualização do que deve ser resolvido.

Conquista: É na conquista que os subproblemas divididos na parte da divisão serão resolvidos um por um recursivamente.

Combinação: Com os resultados da conquista, é feito agora o processo de combinação, que consiste em reunir novamente os subproblemas solucionados, resultando em uma solução para o grande problema, o problema original.

Do ponto de vista da Merge Sort, no processo de divisão, nós dividimos o vetor principal em vetores menores e então, no processo de conquista, nós ordenamos os "sub-vetores", adquiridos na divisão e ordenamos um por um para que no final, possamos combiná-los todos os sub-vetores em um único vetor, solucionando o problema original, ou seja, ordenando o vetor inicial.

Ao optar por utilizar o conceito de Dividir para conquistar, é importante identificar no problema alguns aspectos essenciais para a aplicação dessa estratégia. Ao analisar o problema, caso seja possível identificar que podemos efetuar a divisão dele em subproblemas menores, em tamanhos iguais e que possamos fazer o processo de combinação desses subproblemas de forma eficiente, possivelmente a estratégia de dividir para conquistar será uma boa alternativa de solução.

3. Merge Sort

Ao analisar o processo do Merge Sort é perceptível que sua execução possui formato semelhante ao de uma árvore binária, na qual cada nó é a separação do conjunto em partes de tamanho semelhante, representa uma chamada recursiva ao

Merge sort. Nesse ponto, o nó raiz seria a primeira chamada do Merge Sort, e os nós folhas são as partes do conjunto divididas ao meio.

Teremos como exemplo um vetor desordenado de 8 posições, ao chamarmos a Merge Sort, com o vetor nesse estado, o algoritmo não consegue ordená-lo, então esse vetor será dividido em duas metades, na qual podemos chamar de vetor da esquerda e a outra de vetor da direita, ambas com tamanho semelhante. Em seguida chamaremos a Merge Sort para cada metade, de modo que esse processo se repetirá até que não seja possível gerar um vetor da esquerda e outro da direita, ou seja, quando tivermos apenas um valor.

Após encontrar esse vetor de tamanho único, ele será devolvido para seu conjunto de origem, e com o vetor da direita será feito o mesmo processo, formando o primeiro vetor de origem com duas posições. Então serão realizadas comparações com o primeiro valor do vetor da esquerda e com o primeiro valor do vetor da direita, executando comparações com todos os valores da direita antes de partir para o segundo valor da esquerda e incrementá-lo, ao encontrar um valor menor do que o da direita. Ao encontrar o valor, será feita a troca entre os valores, até ordenar por completo o último conjunto dividido.

O processo ocorrerá até que seja concluído a primeira metade do vetor original, para então fazer o mesmo processo na metade da direita. Por fim, com as duas metades do vetor original ordenadas, resta agora ordená-las, para criar o novo vetor original ordenado.

4. Análise do algoritmo

O Merge Sort possui duas chamadas recursivas, cada uma reduzindo o problema (tamanho do array) na metade, ou seja, $2 * T(n/2)$. Além disso, há também uma chamada a função intercala, que sabemos ser $O(n)$. Portanto, a relação de recorrência é: $T(n)=2 * T(N/2)+N$. Sua complexidade no tempo é $\Theta(n \log 2n)$, dado que a função intercala é $\Theta(n)$ e um vetor com 1 elemento está ordenado. É importante lembrar também que a ordenação é estável, pois mantém a ordem dos elementos iguais, isso porque decidimos que, se o elemento mais à esquerda for menor ou igual ao mais à direita, ele deve ser colocado primeiro no array ordenado.

Apesar de estar na mesma classe de complexidade do Quick Sort, o Merge Sort tende a ser, na prática, um pouco menos eficiente do que o Quick Sort, pois suas constantes são maiores. Contudo, a seu favor, o Merge Sort garante $n*\log n$ para qualquer caso, ou seja, ele não é somente $\Omega(n*\log n)$, mas é $\Theta(n*\log n)$. Isso nos dá uma garantia de que, independente da disposição dos dados em um array, a ordenação será eficiente, enquanto o Quick Sort pode ter ordenação n^2 no pior caso, embora raro. Outra vantagem da Merge Sort é que a complexidade do MergeSort não depende da sequência de entrada. Embora eficiente em diversos aspectos, uma das grandes desvantagens da Merge Sort é que a função intercala requer um vetor auxiliar, o que aumenta o consumo de memória e tempo de execução.

5. Implementação recursiva da Merge Sort

A implementação da Merge Sort pode ser dividida em duas funções: a função mergesort(), responsável por dividir o vetor em partes menores, compreendendo a ideia de dividir o vetor sempre ao meio, formando os vetores da esquerda e da direita e a função intercala(), responsável pelo processo de unir dois vetores de maneira ordenada; O código a seguir, é um exemplo de algoritmo Merge Sort implementado na linguagem C. Este recebe ponto inicial e final do vetor a ser ordenado, bem como o vetor com os valores para ordenar, que são lidos na função main() pela entrada padrão (teclado).

```c
#include <stdio.h>
#include <stdlib.h>

void intercala(int inicio, int meio, int fim, int v[]){
    int vet_esquerda , vet_direita , auxiliar , *ponteiro;

    // Ponteiro auxiliar para guardar os valores ordenados
    ponteiro = malloc((fim - inicio) * sizeof(int));

    vet_esquerda = inicio; // Inicio do vetor da esquerda
    vet_direita = meio; // Inicio do vetor da direita

    auxiliar = 0; // iterador auxiliar para o ponteiro
    // Iterar sobre os dois vetores para ordenar (esquerda ->
vet_esquerda, direita = vet_direita)
    // e verificar qual valor é menor entre os todos os valores
```

```
desses vetores e
    // armazenar na ordem crescente esses valores no vetor
auxiliar.
    while(vet_esquerda < meio && vet_direita < fim){
        if(v[vet_esquerda] <= v[vet_direita]){
            ponteiro[auxiliar] = v[vet_esquerda];
            auxiliar++;
            vet_esquerda++;
        }
        else{
            ponteiro[auxiliar] = v[vet_direita];
            auxiliar++;
            vet_direita++;
        }
    }
    // vetor auxiliar recebe todos os valores restantes do vetor
da esquerda
    // (caso exista valores restantes)
    while(vet_esquerda < meio){
        ponteiro[auxiliar] = v[vet_esquerda];
        auxiliar++;
        vet_esquerda++;
    }
    // vetor auxiliar recebe todos os valores restantes do vetor
da direita
    // (caso exista valores restantes)
    while(vet_direita < fim){
        ponteiro[auxiliar] = v[vet_direita];
        auxiliar++;
        vet_direita++;
    }
    // Vetor principal receber os valores ordenados do vetor
auxiliar
    for(vet_esquerda = inicio;vet_esquerda < fim;vet_esquerda++){
        v[vet_esquerda] = ponteiro[vet_esquerda - inicio];
    }
    free(ponteiro); // Liberar espaco de memoria do ponteiro
}


void mergeSort(int inicio, int fim,int vet[]){
    /*inicio= representa a posição inicial do vetor que será
```

```
ordenado;
    fim = representa até qual posição vamos ordenar o vetor
    vet[] = vetor a ser ordenado*/
    if(inicio < fim - 1){
        int centro = (inicio + fim) / 2;
        //centro = representa a posição central do vetor;
        mergeSort(inicio, centro,vet);
        //Ordenada do início ao centro;
        mergeSort(centro,fim,vet);
        //Ordenada do centro até o fim;
        intercala(inicio,centro,fim,vet);
    }
}
void main(){
    int qnt_num;
    printf("Informe o tamanho do vetor: ");
    scanf("%d",&qnt_num);
    int vetor[qnt_num];
    for(int i = 0; i < qnt_num; i++){
        scanf("%d",&vetor[i]);
    }

    mergeSort(0,qnt_num,vetor);

    printf("Resultado mergesort recursivo :");
    for(int i = 0; i < qnt_num; i++){
        printf("%d ",vetor[i]);
    }
    printf("\n");
}
```

A função mergeSort() executa o processo de recursividade, que chama a si mesmo, enquanto a condição especificada for verdadeira. Esta condição, por sua vez, verifica se a variável inicio é menor que a variável fim - 1, o que significa que o programa está verificando se existe mais de um elemento no nosso vetor. Caso a condição seja verdadeira, o algoritmo encontrará o meio do vetor e o armazenará na variável centro. Em seguida, é feita a recursividade duas vezes, a primeira para a análise do vetor da esquerda e a segunda para o vetor da direita. Caso a condição seja falsa, a função apenas retornará (processo de retornar para as chamadas da função que ficaram em espera para serem executadas). Após passado pelas duas

chamadas recursivas e retornadas, é chamada a função intercala, que será responsável por ordenar o vetor passado por parâmetro. Note que são passadas 4 informações: a variável início, centro, fim e vet. Início se refere ao ponto inicial do vetor onde desejo ordenar (início do vetor da esquerda); a variável centro é o ponto médio, que divide o vetor da esquerda e o vetor da direita que serão ordenados e unidos; a variável fim marcará o ponto final do vetor onde desejo ordenar (fim do vetor da direita).

Na função intercala, será declarado variáveis auxiliares, úteis para o processo de ordenação. Em seguida, o algoritmo verificará, entre os valores do vetor da esquerda e da direita, qual valor é menor. Para cada caso, o valor menor é armazenado em um vetor auxiliar (variável ponteiro) até que os valores de um dos vetores acabe. Dessa forma, para que não haja perda de dados, os valores restantes do outro vetor é enviado para o final do vetor auxiliar. Como foi verificado na condição quais valores eram menores, significa que o vetor auxiliar contém agora todos os valores dos vetores em ordem crescente (com base na comparação e movimentação dos dados). Por fim, esses valores são enviados em ordem, para o vetor principal, mas dessa vez ordenados. Esse processo irá se repetir até que o algoritmo trabalhe sob todo o vetor principal, finalizando com o mesmo totalmente ordenado.

6. Merge Sort iterativa

O algoritmo Merge Sort na forma iterativa também faz o processo de divisão, semelhante à forma recursiva, entretanto, seu comportamento perante o vetor possui algumas diferenças que podem ser identificadas. Diferente da forma recursiva, que divide o vetor ao meio, até que não seja mais possível dividir, e nesse momento, executa o processo de combinação, na forma iterativa, o processo de ordenação das subpartes ocorre de forma diferente. O processo de ordenação ocorre inicialmente a cada dois valores, ordenando os dois primeiros valores pela função intercala. Esse processo será feito por todo o vetor, ordenando todos os valores a cada dois valores. Em seguida, após ordenar a cada dois valores, o processo será feito novamente, mais dessa vez ordenando a cada 4 valores. e assim sucessivamente, sempre dobrando a quantidade de valores que serão ordenados (2, 4, 8, 16, 32, 64), até que seja ordenado todo o vetor.

7. Implementação iterativa da Merge Sort

O código a seguir, é um exemplo de algoritmo Merge Sort implementado na linguagem C.

```
#include <stdio.h>
#include <stdlib.h>

void intercala(int inicio, int meio, int fim, int v[]){
    int vet_esquerda , vet_direita , auxiliar , *ponteiro;

    // Ponteiro auxiliar para guardar os valores ordenados
    ponteiro = (int *) malloc((fim - inicio) * sizeof(int));

    vet_esquerda = inicio; // Inicio do vetor da esquerda
    vet_direita = meio; // Inicio do vetor da direita

    auxiliar = 0; // iterador auxiliar para o ponteiro

    // Iterar sobre os dois vetores para ordenar (esquerda ->
vet_esquerda, direita = vet_direita)
    // e verificar qual valor é menor entre os todos os valores desses
vetores e
    // armazenar na ordem crescente esses valores no vetor auxiliar.
    while(vet_esquerda < meio && vet_direita < fim){
        if(v[vet_esquerda] <= v[vet_direita]){
            ponteiro[auxiliar] = v[vet_esquerda];
            auxiliar++;
            vet_esquerda++;
        }
        else{
            ponteiro[auxiliar] = v[vet_direita];
            auxiliar++;
            vet_direita++;
        }
    }



    // vetor auxiliar recebe todos os valores restantes do vetor da
esquerda
    // (caso exista valores restantes)
    while(vet_esquerda < meio){
```

```
        ponteiro[auxiliar] = v[vet_esquerda];
        auxiliar++;
        vet_esquerda++;
    }

    // vetor auxiliar recebe todos os valores restantes do vetor da
direita
    // (caso exista valores restantes)
    while(vet_direita < fim){
        ponteiro[auxiliar] = v[vet_direita];
        auxiliar++;
        vet_direita++;
    }

    // Vetor principal receber os valores ordenados do vetor auxiliar
    for(vet_esquerda = inicio;vet_esquerda < fim;vet_esquerda++){
        v[vet_esquerda] = ponteiro[vet_esquerda - inicio];
    }
    free(ponteiro); // Liberar espaco de memoria do ponteiro
}

void mergeSortIterativo(int vet[], int tam)
{
   int t_bloco = 1; // quantidade de valores que serão ordenados por vez

   // while responsável pelas repeticoes onde a quantidade de valores a
   // cada intercalacao muda (de 2 valores para 4; 4 para 8)
   while (t_bloco < tam) {
      int inicio = 0; // ponto inicial do vetor para ordenar

      // O while responsavel por intercalar todos os valores do vetor
      // iniciando a cada dois valores,
      while (inicio + t_bloco < tam) {
         int fim = inicio + 2*t_bloco; // Ponto final onde será ordenado
         if (fim > tam) fim = tam; // garantir que não vai ultrapassar o
tamanho do vetor

         intercala(inicio, inicio+t_bloco, fim, vet);
         inicio = inicio + 2*t_bloco;
      }
      t_bloco = 2*t_bloco;// quantidade de valores que serão ordenados
duplica
```

```
    }
}

int main(){
    int qnt_num;

    scanf("%d",&qnt_num);

    int vetor[qnt_num];

    for(int i = 0; i < qnt_num; i++){
        scanf("%d",&vetor[i]);
    }

    mergeSortIterativo(vetor,qnt_num);

  return 0;
}
```

A função iterativa, como podem observar também opera com a mesma função intercala utilizada na recursiva, passando o início do nosso vetor, o inicio + t_bloco que seria o meio do nosso vetor e o fim do vetor. A função mergeSortIterativo é do tipo void, não vai retornar nada, e temos como parâmetros o vetor que será ordenado e o seu tamanho.

De início iremos já declarar a váriavel t_bloco e já atribuímos o valor 1 a t_bloco que assume a quantidade de valores que serão ordenados por vez, ou seja, inicialmente será ordenado de 2 em 2 valores, os dois primeiros, depois os 2 próximos até o fim do veto; após isso, será ordenado de 4 em 4 valores, 4 primeiros os 4 próximos e após isso os 8 valores. Vale ressaltar que esse processo só será feito caso o tamanho do vetor permitir, portanto, se o vetor tiver tamanho 7, será ordenado de 4 em 4 valores e na próxima tentativa não será possível ordenar com 8 valores, já que o vetor não possui essa quantidade de valores, então será feito a ordenação com 7 valores (o vetor completo). A t_bloco é exatamente essa informação, iniciando-se com 2 valores, apesar de ter sido atribuído 1, ela vai ser representada por 0 e 1, do qual são dois valores, depois 4 valores, 8 valores e assim sucessivamente.

Seguindo podemos observar o nosso 1° while, do qual ele é responsável pelas repetições onde a quantidade de valores a cada intercalação muda (de 2 para 4 valores; 4 para 8 valores) e temos sua condição de parada como t_bloco < tam. Isso significa que ele parará de executar quando for feita a ordenação com todo tamanho

do vetor. Em seguida, declaramos a variável ínicio do qual recebe o valor 0 que é o nosso ponto inicial do vetor para ordenar.

O while interno será responsável por atualizar o valor dos intervalos que desejamos ordenar. A exemplo, inicialmente desejamos ordenar os dois primeiros valores. Após ordenado pela função intercala, a variável inicio é atualizada. Na repetição seguinte, será atualizada também o valor da variável fim e dessa forma, o próximo intervalo que será ordenado será o dois próximos valores do nosso vetor (neste caso, o valor da posição 02 e 03 do vetor). Para todos os casos, esses intervalos serão enviados para função intercala, já citada tópico 5, que ordenará esses valores. Esse processo será feito por todo o vetor até que ele esteja totalmente ordenado.