UNIVERSIDADE PAULISTA

Ciências da Computação

Felipe Viana Reis - F287041

Guilherme Cabral – N6654B3

Guilherme de Souza Gomes - F298990

Isaac Lopes de Carvalho – F2517G8

Ismael Soares - N6249E6

Luis Fernando Bueno Viana – F210EF7

Matheus Henrique Xavier Alves – F317BD8

Thaís da Silva Cabral - N655CE1

ATIVIDADE PRÁTICA SUPERVISIONADA - APS

Desenvolvimento de sistema para análise de performance de algoritmos e ordenação de dados

SÃO PAULO

2021

UNIVERSIDADE PAULISTA

Ciências da Computação

ATIVIDADE PRÁTICA SUPERVISIONADA - APS

Desenvolvimento de sistema para análise de performance de algoritmos e ordenação de dados

Trabalho apresentado no curso de Ciências da computação da Universidade Paulista – UNIP, como parte das Exigências para conclusão do semestre e obtenção de nota.

SÃO PAULO

2021

SUMÁRIO

1. INTRODUÇÃO

Na ciência da computação, um algoritmo de ordenação é uma "função" que reúne os elementos de uma cadeia de dados específica em uma ordem específica, que tem como objetivo facilitar a recuperação e legibilidade destes dados.

Imagine uma lista telefônica completamente desorganizada, sem padrões ou regras para seguir. Os nomes, telefones e todas as outras informações foram simplesmente jogadas nesta lista. Pesquisar os contatos certamente seria uma tarefa impossível se nem ao menos os nomes das pessoas estivessem em ordem alfabética. Para que isso não ocorra em programas de computação, existem os algoritmos de ordenação de dados, que organizam a lista telefônica seguindo regras pré-determinadas, tornando a pesquisa dos contatos muito mais fácil.

Existem vários tipos de algoritmos de ordenação, cada qual seguindo diferentes regras e especificações, mas visando sempre o mesmo resultado: organizar conjuntos de dados.

Os métodos de ordenação se classificam em: ordenação interna e ordenação externa.

- Ordenação interna: o arquivo que vai ser ordenado cabe todo na memória principal, e qualquer registro pode ser acessado imediatamente. Neste tipo de ordenação, o tempo gasto para o processo ser concluído e a economia de memória são dois pontos importantes a serem seguidos.

Podemos citar como exemplos de métodos de ordenação interna: ordenação por seleção (Selection Sort), ordenação por inserção (Insertion Sort), e ordenação por troca (Bubble Sort). Alguns outros nomes que podem ser citados: Shell Sort, Quick Sort e Heap Sort.

- Ordenação externa: o arquivo que vai ser ordenado não cabe todo na memória principal, e os registros são acessados de bloco em bloco. Neste tipo de ordenação, deve-se levar em conta que os dados estarão armazenados em unidades de memória externa (discos, fitas etc.).

Todas as linguagens de programação atuais possuem algum método capaz de ordenar dados. Neste trabalho, discutiremos os principais algoritmos de ordenação de dados existentes, como eles funcionam e como implementá-los, focando em abordar os seis algoritmos: Bubble Sort, Heap Sort, Insertion Sort, Merge Sort, Quick Sort, Selection Sort.

Estes algoritmos são os mais famosos, com eles existem diversas técnicas, abordando uma delas; dividir para conquistar, essa é uma solução de técnica onde utiliza divisão do vetor a ser ordenado em sub-vetores para os ordená-los de duas formas, com isso eles se baseiam em um valor chamado pivô. O pivô serve como um elemento base para dividir, os algoritmos Quick Sort e Merge Sort utilizam essa técnica e são bons vetores para soluções com grandes quantidades de dados a serem ordenados, eles escolhem um pivô para dividir a lista em duas; ao lado esquerdo os valores menores do que o pivô e ao lado direito os valores maiores que o pivô. Assim essa técnica os torna eficazes em ordenar grandes quantias de dados. Os sub-vetores são ordenados através de recursão, assim utilizando dividir para conquistar nos sub-vetores até que todo o vetor principal esteja ordenado.

2. REFERENCIAL TEÓRICO

2.1. Quicksort

Um dos algoritmos escolhidos foi o QuickSort, ele é um algoritmo poderoso que utiliza uma técnica chamada DC (Dividir para Conquistar), DC é uma técnica que deve seguir dois passos:

1. Descobrir o caso-base, que deve ser o caso mais simples possível.
2. Dívida ou diminua o seu problema até que ele se torne o caso base.

Dividir para conquistar não é tão simples assim, um breve exemplo de como funciona essa técnica:

Passo 1: Descobrir o caso-base. O array mais simples que você pode obter é se você tiver um array com 0 ou com 1 elemento. Assim se torna simples calcular a soma dele.

Passo 2: Você deve chegar o mais próximo possível de um array vazio a cada recursão.

QuickSort segue uma lógica: Escolha um elemento do array para ser o pivô (Geralmente o pivô é escolhido de forma aleatória). Assim ele pode fazer duas divisões, sendo elas:

1. Números menores que o pivô ao lado esquerdo.
2. Número maiores que o pivô ao lado direito.

Isso se chama *particionamento,* assim você possui:

- Um subarray que possui todos os números menores que o pivô
- O pivô
- Um subarray com todos os números maiores que o pivô

O desempenho do QuickSort depende da escolha do pivô, além que ele não checa se o array já está ordenado, ele tenta ordenar o array de qualquer forma.

Por fim, a escolha do QuickSort é por ele ser um algoritmo poderoso que possui tempo de execução $O(n \log n)$ no melhor e médio caso, e $O(n^2)$ no pior caso, esse pior caso é bom porque em casos específicos de desequilíbrio de particionamento.

Quick Sort demonstra de fato ser um algoritmo muito útil para ocasiões com grande escala de dados a serem ordenados. Porém Quick Sort é um algoritmo não estável, que nesse caso ele não preserva a ordem dos registros de chaves, assim o Quick Sort não seria útil caso necessitássemos ordenar um dicionário, onde é uma estrutura de dados que tem relação *valor : chave.*

2.2. Mergesort

Assim como o QuickSort, o MergeSort utiliza a técnica DC (Dividir para conquistar), porém ele é um algoritmo que ordena por comparação do tipo dividir-para-conquistar.

A sua ordenação por comparação consiste em ler os elementos da lista com uma operação abstrata única (“Menor ou igual a”), com essa comparação o algoritmo decide qual elemento deve vir primeiro na ordenação. Esta técnica exige duas coisas para poder executar:

1. Se a ≤ b e b ≤ c então *a ≤ c (Isso é chamado de transitividade)*
2. para todo *a* e *b*, ou *a ≤ b* ou *b ≤ a* (totalidade ou tricotomia).

A escolha deste algoritmo não tem segredo, ele é considerado próximo a velocidade de ordenação do QuickSort, com a única diferença de ele cumprir uma ordenação que analisa os elementos. Em comparação com outros algoritmos básicos de ordenação (BubbleSort, InsertionSort e SelectionSort), o MergeSort chega a ser mais rápido e mais eficiente quando você tem que ordenar uma grande quantia de dados. O seu tempo de execução são os mesmo em todos os casos! $O(n \log^2 n)$ no melhor, médio e pior caso.

O Merge Sort é muito interessante em questão de grandes quantias de dados, mas assim como o Quick Sort ele depende da escolha do pivô para ter uma execução boa e rápida. Além que há a desvantagem do consumo de memória que o Merge Sort possui, esse algoritmo consume bastante memória durante sua execução por utilizar a recursão para isso.

O Merge Sort em comparação a outros algoritmos de comparação para ordenar demonstra maior velocidade aos algoritmos básicos (Bubble Sort, Insertion Sort e Selection Sort), Merge Sort tem mais desempenho com os dados em larga escala pelo seu uso de dividir para conquistar, mas caso houvesse necessidade de ordenar vetores com dados menores, seria mais conciliável usar vetores mais simples que se dão melhor com dados menores, que nesse caso seria por recomendação o algoritmo Insertion Sort, não apenas por ser um algoritmo bom para ordenar poucos dados, ainda mais quando o vetor a ser ordenado já está quase ordenado, assim as comparações dos elementos se tornam mais rápido para estas situações.

2.3. InsertionSort

InsertionSort é um algoritmo de ordenação que utiliza a estrutura de dado vetor ou lista constrói uma matriz com um elemento de cada para uma inserção por vez. Insertion Sort é o mais eficiente entre os algoritmos desta classificação. Quando ele recebe um elemento por vez, ele compara com os outros elementos do vetor ou lista para poder fazer a ordenação de forma correta, é similar com a técnica de dividir para conquistar, claro que ele não utiliza nenhuma recursão para executar a tarefa e nem sub-vetores com pivôs para serem executados esta função.

3. DESENVOLVIMENTO

No projeto estamos abordando um cenário de ordenação de 100 mil dados fornecidos pela NASA. Nesta ocasião, o projeto utilizou uma biblioteca padrão da linguagem C de entrada e saída dos dados para essa obtenção dos dados. Com a biblioteca de entrada e saída dos dados é possível ler arquivos externos para poder vasculhar informações no arquivo escolhido, existe uma função na biblioteca de I/O (Entrada/saída) que lê os dados fornecidos, com isso, podemos obter os dados com as seguintes etapas:

1. Inicializar uma estrutura de dados chamada ***FILE**** assim poderemos extrair os dados de uma forma correta do arquivo, exemplo:

```
FILE* f1;
FILE* f2;
```

Fonte: realizado pelos autores

2. Utilizando a função ***fopen*** para abrir o arquivo desejado passando nos parâmetros da função, e usando um argumento chave de apenas leitura do arquivo que seria *r,* assim temos:

```
f1 = fopen("100k.txt", "r");
f2 = fopen("100kdup.txt", "r");
```

Fonte: realizado pelos autores

Com isso já obtemos os dados, mas não basta apenas abrir e ler esse arquivo, precisamos focar na solução que a NASA precisa: ordenar os dados. Ao obter esses dados, precisamos de um vetor que possa armazenar uma lista de dados, a forma que precisamos fazer isso é usando outra função da biblioteca de I/O na leitura de arquivos, a linguagem C tem um padrão de entrada de dados com a função ***scanf*** e não é muito diferente para fazer isso com os arquivos, existe a função ***fscanf*** uma função que podemos entrar com os dados do arquivo para ser armazenado. Nessa função, passamos 10 argumentos no parâmetro: arquivo, tipo dado e onde esse dado será armazenado, além disso precisamos de algo para poder percorrer linha a linha do documento lendo e obtendo os dados em cada linha. Desta forma, usamos um laço de repetição para poder iterar com essas linhas e poder armazenar os dados no vetor

desejado. Assim o código ficaria com um laço de repetição ***while*** utilizando um contador para podermos ter uma condição que armazene os 100 mil dados em nosso vetor. Teremos uma sintaxe dessa forma, como mostra a figura abaixo:

```
int vetor[tamanho]; // vetor para ser ordenado
int vetor_duplicidade[tamanho];

int contador = 0;
while(contador <= tamanho){
  fscanf(f1, "%d\n", &vetor[contador]);
  fscanf(f2, "%d\n", &vetor_duplicidade[contador]);
  contador++;
}
```

Fonte: realizado pelos autores

Com isso já conseguimos os dados para ser ordenado. Uma amostra dos dados a serem são ordenados são dois vetores, um com dados sem duplicidades e outro com dados duplicados. Na figura abaixo há uma amostragem de alguns dados não ordenados:

```
Dados sem duplicidade: 2069254, 3190673, 9487999, 4564769, 3006025, 6056903, 9522690, 5969561, 1136518,
Dados com duplicidade: 4449079, 8318910, 1196550, 1881149, 3256704, 5434856, 5634817, 5001673, 6296735,
```

Fonte: realizado pelos autores

No processo precisamos organizar esses dados utilizando algoritmos de alta complexidade para poder fazer a organização. Aqui será abordado apenas os algoritmos utilizados no projeto final.

Além disso, houve a necessidade de introduzir uma forma para incluir o cálculo da performance em tempo pelos algoritmos, utilizando uma biblioteca da linguagem C chamada ***time*** nos possibilita criar uma estrutura de dados que é possível fazer o cálculo dos ***ticks*** do ***clock*** do processador assim sendo possível armazenar essas informações para podermos fazer a comparação. Nessa etapa devemos chamar a biblioteca:

```
#include <time.h>
```

Após chamar essa biblioteca em nosso projeto, devemos construir duas estruturas de dados que possibilitam a análise do início e o fim do algoritmo e iniciar uma variável que irá armazenar o cálculo de segundos registrados nessas estruturas de dados, desta forma temos:

```
clock_t start, end;
double cpu_time_used;
```

Com isso, já se torna possível fazer essa análise em todos os algoritmos, um exemplo é com o algoritmo QuickSort, nele devemos colocar a estrutura de dados de clock no antes da execução do algoritmo e a outra estrutura de dados clock após a execução desse algoritmo, assim ficaria:

```
start = clock();
QuickSort(vqs, tamanho);
end = clock();
```

Com isso, estamos atribuindo uma função chamada clock para poder receber os ticks do clock do processador ao executar esse algoritmo, por fim temos o seguinte cálculo para descobrir precisamente quantos segundos foram utilizados:

```
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

Assim finalmente conseguimos obter os dados a serem exibidos.

3.1. QuickSort

```
void QuickSort(int* v, int tam){
  int j = tam, k;
  if(tam <= 1)
    return;
  else {
    int x = v[0];
    int a = 1;
    int b = tam - 1;
    do{
      while ((a < tam) && (v[a] <= x))
        a++;
      while (v[b] > x)
        b--;
      if (a < b) { // faz troca
        int temp = v[a];
        v[a] = v[b];
        v[b] = temp;
        a++;
        b--;
      }
    } while(a <= b);
    //troca pivo
    v[0] = v[b];
    v[b] = x;
    //ordena sub-vetores restantes
    QuickSort(v, b);
    QuickSort(&v[a], tam - a);
  }
}
```

Fonte: realizado pelos autores

O QuickSort utiliza uma técnica chamada dividir para conquistar, ele faz a divisão com base no pivô, todos os elementos menores que o pivô será colocado a esquerda do pivô e os elementos maiores que o pivô a direita. Seus passos:

1. Escolher algum elemento da lista que será o nosso pivô;
2. Divisão: Organiza a lista onde todos os elementos anteriores ao pivô sejam menores que ele e os posteriores sejam maiores que o pivô;
3. Recursividade: ordena os sub-vetores dos elementos menores e maiores que o pivô.

3.2. MergeSort

```
void MergeSort(int* v, int inicio, int fim){
  int i, j, k, meio, * t, z;
  if (inicio == fim)
    return;
  //ordenacao recurviva das duas metades
  meio = (inicio + fim) / 2;
  MergeSort(v, inicio, meio);
  MergeSort(v, meio + 1, fim);
  //intercalacao no vetor temporario t
  i = inicio;
  j = meio + 1;
  k = 0;
  t = (int*)malloc(sizeof(int) * (fim - inicio + 1));
  while (i < meio + 1 || j < fim + 1){
    if (i == meio + 1){
      t[k] = v[j];
      j++; k++;
    }
    else if (j == fim + 1){
      t[k] = v[i];
      i++; k++;
    }
    else if (v[i] < v[j]){
      t[k] = v[i];
      i++; k++;
    }
    else {
      t[k] = v[j];
      j++; k++;
      }
    }

  //copia vetor intercalado para o vetor original
  for (i = inicio; i <= fim; i++)
    v[i] = t[i - inicio];
  free(t);
  }
```

Fonte: realizado pelos autores

Da mesma forma que o QuickSort, o MergeSort utiliza a técnica de dividir para conquistar, utilizando os mesmos passos que o QuickSort: Escolhendo o pivô, dividindo em duas listas e as ordenando para obter o resultado. Com exceção que ele demanda mais processamento da memória.

3.3. InsertionSort

```
void InsertionSort(int* v, int tam){
  int i, j, k, chave;
  for (j = 1; j < tam; j++) {
    chave = v[j];
    i = j - 1;
    while ((i >= 0) && (v[i] > chave)) {
      v[i + 1] = v[i];
      i--;
    }
    v[i + 1] = chave;
  }
}
```

Fonte: realizado pelos autores

InsertionSort é um algoritmo de ordenação que utilizam dados no como lista (vetor, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

Percebemos que ao criar os algoritmos seria necessário saber o tamanho do vetor a ser utilizado nesses algoritmos, com isso criamos uma constante com o valor de 100 mil:

```
#define tamanho 100000
```

Agora indo ao projeto final para poder executar todas essas funções, criamos um laço de repetição para poder exibir os dados ordenados e duas funções de exibir na tela para cada algoritmo, assim podemos comparar todos os dados dos três algoritmos escolhidos, assim temos o seguinte código:

```
int contador2 = 0;
  while(contador2 < tamanho){
      printf("%d, ", vqs[contador2]);
      contador2++;
  }
  printf("Acima voce pode ver o vetor ordenado.\n");
printf("QuickSort tempo: %f\n", cpu_time_used_qs);
```

Um detalhe que incluímos durante a execução foi a função ***sleep*** essa função permitir o código dar um tempo para o código “Dormir”, nesse caso queríamos que ele tivesse

uma parada para melhorar a performance dos algoritmos, mas isso teve influência nenhuma na execução, a função:

```
sleep(3);
```

Com a montagem do código podemos ter todos os resultados de execução ao mesmo tempo:

```
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706, 1763, 1868, 1997,
QuickSort tempo: 0.016747
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706, 1763, 1868, 1997,
InsertionSort tempo: 8.348128
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706, 1763, 1868, 1997,
MergeSort tempo: 0.026355
```

Cada algoritmo teve um vetor diferente declarado para não ocorrer o problema de tentarem ordenar um vetor já ordenado.

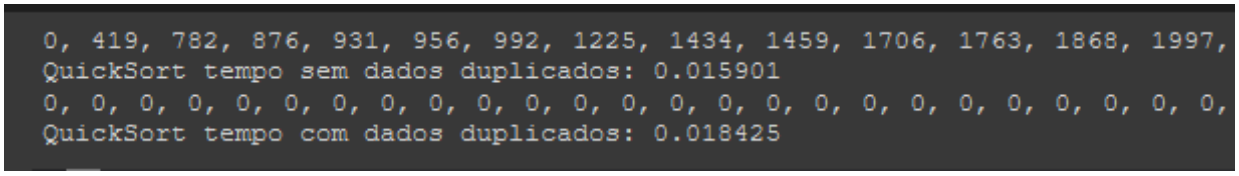
```
int vqs[tamanho];
int vis[tamanho];
int vms[tamanho];
```

O vetor denominado como *vqs* é o vetor que o algoritmo Quick Sort irá utilizar, *vis é o vetor que o Insertion Sort* irá utilizar e *vms* é o vetor para o Merge Sort ordenar.

Com todas as ferramentas organização prontos podemos finalmente ver os resultados de performance e ordenação do algoritmo.

Antes de abordar de fato sobre a saída das execuções é importante informar o leitor que todos os testes e o código foi executado em um Notebook no Google Colab, com todos os testes e algoritmos (Colab), assim sendo possível ver o código utilizado para cada algoritmo proposto (6), e os testes com os dados duplicados e não duplicados, o Google Colab foi uma excelente ferramenta nesse projeto para o desenvolvimento, levando em consideração que o hardware poderia afetar o desempenho de cada algoritmo, utilizar o Google Colab foi uma ideia estratégica para executar os testes dos algoritmos sem ter interferência (A não ser que você pague a mais pelo poder de processamento no ambiente virtual que o Google Colab propõe).

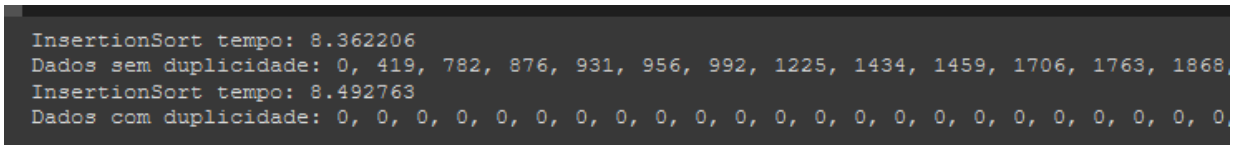
- QuickSort:



Fonte: realizado pelos autores

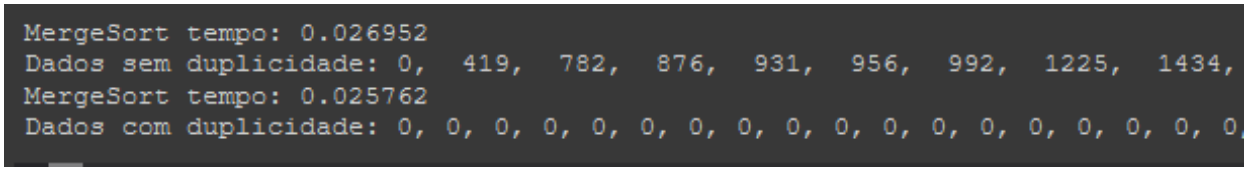
Aqui QuickSort mostrou eficiência nas ordenações e sem a perca de dados com duplicidade e mostrou também alto desempenho na ordenação dos 100 mil dados apesar de a amostragem desses dados estarem encurtados por questões didáticas.

- InsertionSort:



Fonte: realizado pelos autores

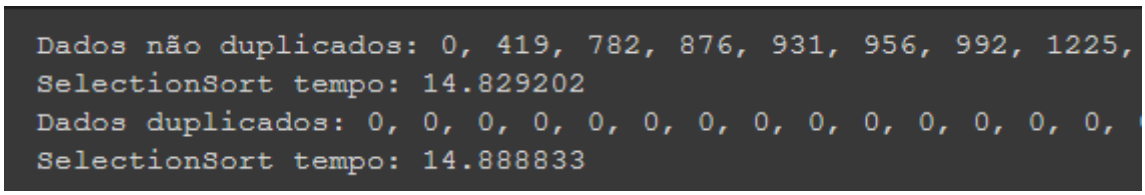
- MergeSort:



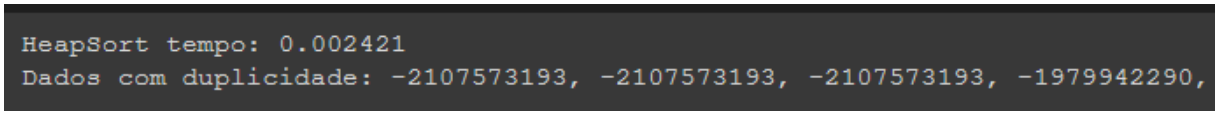
Fonte: realizado pelos autores

MergeSort também garantiu a integridade dos dados e com uma velocidade ótima.

- SelectionSort:



- HeapSort:



- BubbleSort:

```
BubbleSort tempo: 38.094349
Dados sem duplicidade: 0, 419, 782, 876, 931, 956, 992, 1225, 1434,
BubbleSort tempo: 38.210616
Dados com duplicidade: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Cada Código teve sua própria célula no Google Colab e poder analisar eles separadamente sem a necessidade de fazer um código maior no fim do projeto. Os códigos em cada célula:

```c
void BubbleSort(int *v, int tam) {
  int i, j = tam, k;
  int trocou;
  do {
    tam--;
    trocou = 0;
    for (i = 0; i < tam; i++) {
      if (v[i] > v[i + 1]) {
        int aux = 0;
        aux = v[i];
        v[i] = v[i + 1];
        v[i + 1] = aux;
        trocou = 1;
      }
    }
  } while (trocou);
}

int main(){
    FILE* f1;
    FILE* f2;
    f1 = fopen("100k.txt", "r");
    f2 = fopen("100kdup.txt", "r");
    int vetor[tamanho]; // vetor para o bubble
    int vetor2[tamanho];
    int contador = 0;
    while(contador <= tamanho){
      fscanf(f1, "%d\n", &vetor[contador]); // vetoriza os 100k de dados sem duplicidades
      fscanf(f2, "%d\n", &vetor2[contador]);
      contador++;
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    BubbleSort(vetor,  tamanho);
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("BubbleSort tempo: %f\n", cpu_time_used);
    printf("Dados sem duplicidade: ");
    int contador2 = 0;
    while(contador2 < tamanho){
        printf("%d, ", vetor[contador2]);
        contador2++;
    }

    start = clock();
    BubbleSort(vetor2,  tamanho);
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("\nBubbleSort tempo: %f\n", cpu_time_used);
    printf("Dados com duplicidade: ");
    contador2 = 0;
    while(contador2 < tamanho){
        printf("%d, ", vetor2[contador2]);
        contador2++;
    }
}
```

Fonte: realizado pelos autores

```
void PercorreArvore(int *v, int raiz, int folha) {
  int percorreu, maxfolhas, temp;

  percorreu = 0;
  while ((raiz*2 <= folha) && (!percorreu)) {
    if (raiz*2 == folha) {
      maxfolhas = raiz * 2;
    }
    else if (v[raiz * 2] > v[raiz * 2 + 1]) {
      maxfolhas = raiz * 2;
    }
    else {
      maxfolhas = raiz * 2 + 1;
    }

  if (v[raiz] < v[maxfolhas]) {
    temp = v[raiz];
    v[raiz] = v[maxfolhas];
    v[maxfolhas] = temp;
    raiz = maxfolhas;
    }
  else {
      percorreu = 1;
  }
  }
}

void HeapSort(int *v, int tam) {


  int i, temp;

  for (i = (tam / 2); i >= 0; i--) {
    PercorreArvore(v, i, tam - 1);
  }

  for (i = tam-1; i >= 1; i--) {
    temp = v[0];
    v[0] = v[i];
    v[i] = temp;
    PercorreArvore(v, 0, i-1);
  }


}
```

Fonte: realizado pelos autores

```c
void SelectionSort(int* v, int tam){
  int i, j, k, min;
  for(i = 0; i < (tam - 1); i++){
      min = i;
      for(j = (i + 1); j < tam; j++) {
          if(v[j] < v[min]){
              min = j;
          }
      }
      if (i != min){
          int swap = v[i];
          v[i] = v[min];
          v[min] = swap;
      }
  }
}
int main(){
    FILE* f1;
    FILE* f2;

    f1 = fopen("100k.txt", "r");
    f2 = fopen("100kdup.txt", "r");
    int vss[tamanho]; // vetor para o SelectionSort
    int vss2[tamanho]; // vetor para dados duplicados

    int contador = 0;
    while(contador <= tamanho){
      fscanf(f1, "%d\n", &vss[contador]); // vetoriza os 100k de dados sem duplicidades
      fscanf(f2, "%d\n", &vss2[contador]);
      contador++;
    }

    clock_t start, end;
    double cpu_time_used;

    start = clock();
    SelectionSort(vss, tamanho);
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Dados não duplicados: ");
    int contador2 = 0;
    while(contador2 < tamanho){
        printf("%d, ", vss[contador2]);
        contador2++;
    }

    printf("\nSelectionSort tempo: %f\n", cpu_time_used);

    //-----------//
    start = clock();
    SelectionSort(vss2, tamanho);
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Dados duplicados: ");
    contador2 = 0;
    while(contador2 < tamanho){
        printf("%d, ", vss2[contador2]);
```

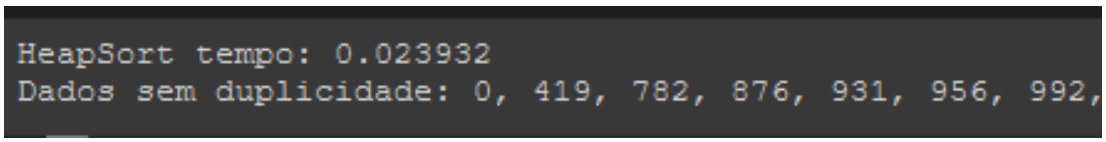
Fonte: realizado pelos autores

4. RESULTADO E DISCUSSÃO

Os testes dos algoritmos foram executados em um Notebook do Google Colab, uma plataforma que roda em uma máquina virtual em Cloud, assim podendo ter os mesmos resultados em quaisquer computadores sem a interferência de diferentes Hardwares, assim dependendo do computador da pessoa que está executando o teste não há diferença em desempenho, apenas o desempenho da plataforma. Vamos fazer a magia acontecer!

4.1. Análise do algoritmo HeapSort

O algoritmo HeapSort é bem rápido em questão de processamento, ao rodar o teste de desempenho e ordenação obtivemos o seguinte resultado:



Fonte: realizado pelos autores

O algoritmo mostrou que seu desempenho é bem alto, principalmente com uma lista de 100 mil dados a serem ordenados, porém houve o levantamento se ele mantinha a integridade desses dados. Então executamos o algoritmo com dados duplicados para poder analisar o resultado.

Eis o seguinte resultado:

```
HeapSort tempo: 0.002421
Dados com duplicidade: -2107573193, -2107573193, -2107573193,
```

Fonte: realizado pelos autores

O algoritmo manteve o desempenho rápido na execução, mas mostrou que a integridade dos dados não foi mantida. Mostrando que é um algoritmo ineficiente caso há dados duplicados. Então resolvemos descartar esse algoritmo por demonstrar essa instabilidade.

4.2. Análise do algoritmo SelectionSort

SelectionSort por si há uma desvantagem por ser lento com entradas de grandes dados, esse algoritmo mostra grande desempenho ao ordenar poucos dados, mas em uma situação que deva ordenar uma quantidade grande de dados pode ser que ele demostre lentidão, vamos pôr a prova com o caso de uso.

Ao executar o SelectionSort com uma lista de 100 mil dados, obtivemos o seguinte resultado:

```
Dados não duplicados: 0, 419, 782, 876, 931, 956, 992, 1225, 1434,
SelectionSort tempo: 14.829202
Dados duplicados: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
SelectionSort tempo: 14.888833
```

Fonte: realizado pelos autores

Ele mostrou que mantém a integridade dos dados, mas não se deu bem com o tamanho dos dados a serem ordenados, assim fazendo que ele seja um algoritmo de fato lento para ordenar nossos dados nesse caso de uso. Por ele ter demonstrado essa lentidão ao ordenar os dados, decidimos descartá-lo.

4.3. Análise do algoritmo BubbleSort

BubbleSort é um algoritmo muito famoso por mostrar que não se dá bem quando ***Softwares*** necessitam de alto desempenho e mexer com larga escala de dados, assim, sabemos que BubbleSort será lento ao executar os dados necessários para ordenação, ao executar o código:

```
BubbleSort tempo: 38.094349
Dados sem duplicidade: 0, 419, 782, 876, 931, 956, 992, 1225,
BubbleSort tempo: 38.210616
Dados com duplicidade: 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Fonte: realizado pelos autores

O algoritmo mostrou que mantém a integridade dos dados, mas por outro lado mostrou alta latência ao executar o algoritmo, por esse motivo o descartamos.

4.4. Análise do algoritmo InsertionSort

O algoritmo InsertionSort mostra que é um algoritmo instável e mostra uma desvantagem que é mover muito os dados para pode-los ordenar. Ele é bom em situações que os dados a serem ordenador não tem muita discrepância na ordem em que estão para serem ordenados. É um ótimo algoritmo.

Ao executar o teste, sua saída:

```
InsertionSort tempo: 8.362206
Dados sem duplicidade: 0, 419, 782, 876, 931,
InsertionSort tempo: 8.492763
Dados com duplicidade: 0, 0, 0, 0, 0, 0, 0, 0,
```

Fonte: realizado pelos autores

Ele mostra uma certa latência ao ordenar a lista, porém comparamos eles aos algoritmos Heap, Bubble e Selection, ao deparar que ele foi o quarto algoritmo testado e outros dois algoritmos tem a latência maior que a dele e o algoritmo HeapSort demonstrou falta de integridade dos dados; decidimos adotar esse algoritmo como um dos três escolhidos.

4.5. Análise do algoritmo MergeSort

MergeSort é um dos algoritmos de ordenação mais poderosos, sua técnica de Dividir para Conquistar o torna rápido em ordenar grandes vetores, porém, ao utilizar este algoritmo poderoso ele exige muita demanda na memória do computador utilizado. Como o caso de uso é apenas ordenar os seus valores, ele é um algoritmo altamente qualificado. Ao executar:

```
MergeSort tempo: 0.026952
Dados sem duplicidade: 0,  419,  782,  876,  931,
MergeSort tempo: 0.025762
Dados com duplicidade: 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Fonte: realizado pelos autores

MergeSort demonstrou que há uma grande velocidade em ordenar todos esses dados e manteve sua integridade, isso o tornou segundo algoritmo elegível para nós.

4.6. Análise do algoritmo QuickSort

O QuickSort é um dos algoritmos mais famosos de ordenação (Senão O algoritmo mais famoso), ele é bem similar ao MergeSort, porém a escolha do seu pivô que faz toda diferença na execução. Ele é um algoritmo poderoso que mostra grande desempenho com esse problema de ordenar grandes quantias de dados. Vamos fazer a mágica acontecer:

```
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706,
QuickSort tempo sem dados duplicados: 0.015901
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
QuickSort tempo com dados duplicados: 0.018425
```

Fonte: realizado pelos autores

O QuickSort mostrou ser o algoritmo mais rápido em ordenar todos os dados além disso manteve a integridade dos dados. Assim sendo o último algoritmo escolhido e o melhor algoritmo para o uso.

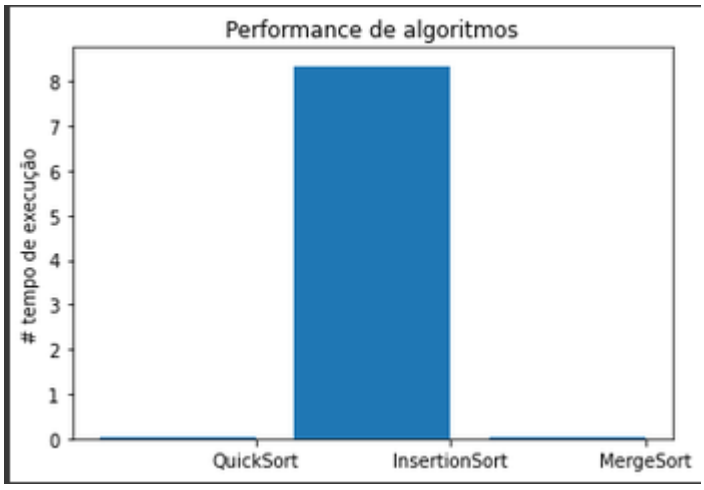
Com todas as análises desses códigos e decidindo de uma forma por eficiência de velocidade do código e por manter os dados a serem inseridos, assim facilitando a escolha dos algoritmos além da parte teórica que todos havia estudado sobre cada algoritmo e podendo levantar discussões sobre quais algoritmos pareciam ser os mais promissores, porém houve um engano com o algoritmo do HeapSort, por mais que ele seja um algoritmo houve o problema com valores duplicados e decidimos descarta-lo, caso tivéssemos um cenário mais simples para os algoritmos descartados eles seriam mais apropriados na escolha, já que a grande quantia de dados a ser armazenada é importante para esse projeto.

Com isso classificamos do melhor e pior algoritmo nessa situação:

Algoritmos	Tempo de Execução	Classificação
QuickSort	0.01	1º
MergeSort	0.02	2º
InsertionSort	8.34	3º
SelectionSort	14.82	4º
BubbleSort	38.09	5º
HeapSort	0.02	6º

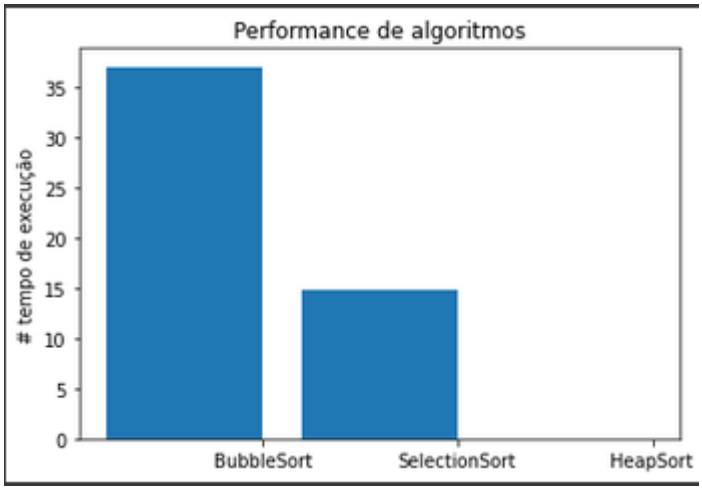
Fonte: realizado pelos autores

Também podemos ver o desempenho da execução dos algoritmos escolhidos:



Fonte: realizado pelos autores

E dos algoritmos descartados:



Fonte: realizado pelos autores

Então com os últimos dados considerados, temos aqui uma imagem dos três algoritmos escolhidos e seus desempenho juntos com dados a serem ordenados:

```
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706, 1763, 1868, 1997,
QuickSort tempo: 0.016747
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706, 1763, 1868, 1997,
InsertionSort tempo: 8.348128
0, 419, 782, 876, 931, 956, 992, 1225, 1434, 1459, 1706, 1763, 1868, 1997,
MergeSort tempo: 0.026355
```

5. CONSIDERAÇÕES FINAIS

Com o projeto utilizamos as técnicas mais eficazes com o desempenho, as técnicas para analisar o algoritmo foram utilizando bibliotecas da linguagem C e também utilizando outros dados em um cenário possível que seria os dados duplicados, muitas das vezes não necessitamos de excluir os dados duplicados por algum motivo, consideramos não só o desempenho, mas também a ordenação desses dados com duplicidade devidamente para quando alguém possa querer analisar os datasets e usar um mascaramento booleano caso necessitasse fazer uma análise de dados mais precisa para tirar insights, então decidimos viabilizar a integridade desses dados. Ficamos contentes com a forma que desempenhamos o teste e não utilizamos tanta complexidade. Com isso, fica a recomendação do grupo caso haja um cenário onde deva utilizar um único algoritmo para ordenar grandes dados e exigindo desempenho: Quick Sort.

Os resultados obtidos mostram como não é simplesmente utilizar qualquer algoritmo para fazer essa ordenação, deve ser considerado diversos fatores antes de escolher alguma dessas técnicas, a ciência da computação é muito mais que escrever código e desenvolver coisas, analisar esses resultados e a prática de testar, fazer hipóteses e teorizar as coisas para poder ter um resultado concreto dessas informações obtidas foi algo essencial nesse processo do projeto.

Nossa grande paixão a Alan Turing por contribuir com essa ciência maravilhosa que é a ciência da computação!

6. REFERÊNCIAS BIBLIOGRÁFICAS

BHARGAVA, A. Y. **Entendendo algoritmos:** Um guia ilustrado para programadores e curiosos. Novatec, 2017.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. **Algoritmos**: Teoria e Prática. 3a edição. Elsevier, 2012

DONALD KNUTH, The Art of Computer Programming Vol. 3. Sorting and Searching Second Edition, 1973.

MARCELLO LA ROCCA, Advanced Algorithms and Data Structures, Manning.

ALFRED AHO and JEFFREY ULLMAN, Foundations Of Computer Science, JEFFREY ULLMAN, 1992.

Fontes:

Documentação da linguagem C: https://devdocs.io/c/

7. CÓDIGO FONTE

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <time.h>
#include <unistd.h>

#define tamanho 100000

void QuickSort(int* v, int tam){
        int j = tam, k;
        if(tam <= 1)
                return;
```

```
        else {
                int x = v[0];
                int a = 1;
                int b = tam - 1;
                do{
                        while ((a < tam) && (v[a] <= x))
                                a++;
                        while (v[b] > x)
                                b--;
                        if (a < b) { // faz troca
                                int temp = v[a];
                                v[a] = v[b];
                                v[b] = temp;
                                a++;
                                b--;
                        }
                } while(a <= b);
                //troca pivo
                v[0] = v[b];
                v[b] = x;
                //ordena sub-vetores restantes
                QuickSort(v, b);
                QuickSort(&v[a], tam - a);
        }
}

void InsertionSort(int* v, int tam){
        int i, j, k, chave;
        for (j = 1; j < tam; j++) {
                chave = v[j];
```

```
            i = j - 1;
            while ((i >= 0) && (v[i] > chave)) {
                v[i + 1] = v[i];
                i--;
            }
            v[i + 1] = chave;
        }
}


void MergeSort(int* v, int inicio, int fim){
        int i, j, k, meio, * t, z;
        if (inicio == fim)
            return;
        //ordenacao recurviva das duas metades
        meio = (inicio + fim) / 2;
        MergeSort(v, inicio, meio);
        MergeSort(v, meio + 1, fim);
        //intercalacao no vetor temporario t
        i = inicio;
        j = meio + 1;
        k = 0;
        t = (int*)malloc(sizeof(int) * (fim - inicio + 1));
        while (i < meio + 1 || j < fim + 1){
            if (i == meio + 1){
                t[k] = v[j];
                j++; k++;
            }
            else if (j == fim + 1){
                t[k] = v[i];
```

```
                i++; k++;
            }
            else if (v[i] < v[j]){
                t[k] = v[i];
                i++; k++;
            }
            else {
                t[k] = v[j];
                j++; k++;
                }
            }

        //copia vetor intercalado para o vetor original
        for (i = inicio; i <= fim; i++)
            v[i] = t[i - inicio];
        free(t);
        }

int main(){

        int vqs[tamanho]; // vetor para o QuickSort
        int vis[tamanho]; // vetor para o HeapSort
        int vms[tamanho]; // vetor para o MergeSort

        FILE* f1;
        FILE* f2;
        FILE* f3;
        f1 = fopen("100k.txt", "r");
        f2 = fopen("100k.txt", "r");
        f3 = fopen("100k.txt", "r");
```

```
        clock_t start_qs, end_qs, start_hs, end_hs, start_ms, end_ms;
        double cpu_time_used_qs, cpu_time_used_hs, cpu_time_used_ms;

        int c1 = 0;
        while(c1 <= tamanho){
                fscanf(f1, "%d\n", &vqs[c1]); // vetoriza os 100k de dados sem
duplicidades
                c1++;
        }



        start_qs = clock();
        QuickSort(vqs, tamanho);
        end_qs = clock();

        cpu_time_used_qs = ((double) (end_qs - start_qs)) / CLOCKS_PER_SEC;



        int contador2 = 0;
   while(contador2 < tamanho){
      printf("%d, ", vqs[contador2]);
      contador2++;
   }
   printf("Acima voce pode ver o vetor ordenado.\n");
        printf("QuickSort tempo: %f\n", cpu_time_used_qs);
        sleep(3);
        //-------------------------------------------------//
```

```
        int c2 = 0;
        while(c2 <= tamanho){
                fscanf(f2, "%d\n", &vis[c2]); // vetoriza os 100k de dados sem
duplicidades
                c2++;
        }

        start_hs = clock();
        InsertionSort(vis, tamanho);
        end_hs = clock();
        ;
        contador2 = 0;
   while(contador2 < tamanho){
      printf("%d, ", vis[contador2]);
      contador2++;
   }

        cpu_time_used_hs = ((double) (end_hs - start_hs)) / CLOCKS_PER_SEC;
   printf("Acima voce pode ver o vetor ordenado.\n");
        printf("InsertionSort tempo: %f\n", cpu_time_used_hs);
        sleep(3);
        //-----------------------------------------------------//



        // adiciona os dados no vetor do mergesort
        int c3 = 0;
        while(c3 <= tamanho){
                fscanf(f3, "%d\n", &vms[c3]); // vetoriza os 100k de dados sem
duplicidades
                c3++;
```

```
        }

        start_ms = clock();
        MergeSort(vms,  0, tamanho-1);
        end_ms = clock();

        contador2 = 0;
    while(contador2 < tamanho){
       printf("%d, ", vms[contador2]);
       contador2++;
    }

        cpu_time_used_ms = ((double) (end_ms - start_ms)) / CLOCKS_PER_SEC;
    printf("Acima voce pode ver o vetor ordenado.\n");
        printf("MergeSort tempo: %f\n", cpu_time_used_ms);
        sleep(3);

        fclose(f1);
        fclose(f2);
        fclose(f3);

}
```

8. Ficha APS

UNIVERSIDADE PAULISTA

FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME: Guilherme Cabral Procopio TURMA: CC4P04 RA: N6654B3

CURSO: Ciências da Computação CAMPUS: Paulista SEMESTRE: 4º Semestre TURNO: Noturno

CÓDIGO DA ATIVIDADE: 77B1 SEMESTRE: 4º SEMESTRE ANO GRADE: 2021

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
13/10/2021	Estudo de algoritmo – Estudo feito por todos os participantes	8	Guilherme Cabral		
15/10/2021	Reunião para discussão dos algoritmos – Reunião com todos os	2	Guilherme Cabral		
16/10/2021	implementação dos algoritmos Guilherme Cabral, Matheus, Felip	5	Guilherme Cabral		
18/10/2021	Desenvolvimento do programa para execução do algoritmo – Gu	5	Guilherme Cabral		
20/10/2021	Teste algoritmos parte I – Isaac, Ismael e Felipe	3	Guilherme Cabral		
25/10/2021	Teste algoritmos parte II – Matheus, Guilherme de Souza	3	Guilherme Cabral		
29/10/2021	Elaboração da documentação parte I – Luis e Guilherme Cabral	2	Guilherme Cabral		
02/11/2021	Elaboração da documentação parte II – Guilherme Cabral e Thai	3	Guilherme Cabral		
03/11/2021	Elaboração da documentação parte III – Guilherme Cabral	2	Guilherme Cabral		
05/11/2021	Reunião para discussão dos testes – Reunião com todos os inte	4	Guilherme Cabral		
06/11/2021	Decisão da escolha dos algoritmos – Todos os participantes	2	Guilherme Cabral		
09/11/2021	Formatação da documentação no padrão abnt – Thais Cabral	2	Guilherme Cabral		

(1) Horas atribuídas de acordo com o regulamento das Atividades Práticas Supervisionadas do curso.

Declaro que as informações acima são verdadeiras e que possuo os comprovantes a serem entregues ao coordenador do meu curso, no final do período de suspensão emergencial/Covid-19.

TOTAL DE HORAS ATRIBUÍDAS:_____

AVALIAÇÃO:_____

Aprovado ou Reprovado

NOTA:_____

DATA:____/_____/_____

_____

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO