Trabalho de Estruturas de Dados e Algoritmos

Professor: Eduardo falcão

Aluno: Luis Felipe Vanin Martins

1. Introdução

Nos estudos de estruturas de dados foi visto diversos tipos de maneiras de armazenar valores na memória usando C, desde o modelo de *Array List* até *Linked Stack*, porém agora iremos trabalhar algoritmos de extrema importâncias no mundo dos dados que se relaciona com a organização dessas estruturas, no caso deste relatório, iremos trabalhar com a ordenação de nossas estruturas.

Atualmente existem diversos tipos de algoritmos de ordenação em que variam de eficiência, neste relatório iremos trabalhar com cinco algoritmos de ordenação, sendo eles: Insertion Sort, Selection Sort, Bubble Sort, Quick Sort e Merge Sort. Para parâmetro de comparação iremos aplicar testes de performances com o mesmo array para todos os algoritmos e analisar suas performance.

2. Desenvolvimento

Para demonstrar e implementar os algoritmos citados acima iremos criar uma série de ilustrações com breves explicações para todos os algoritmos, com isso podemos apresentar o funcionamento e o desempenho de cada método de ordenação. Com intuito de realizar essas ilustrações será utilizado dois vetores a serem ordenados de forma crescente, esses vetores são:

a = [3, 6, 2, 5, 4, 3, 7, 1] e b = [1, 2, 3, 3, 4, 5, 6, 7]

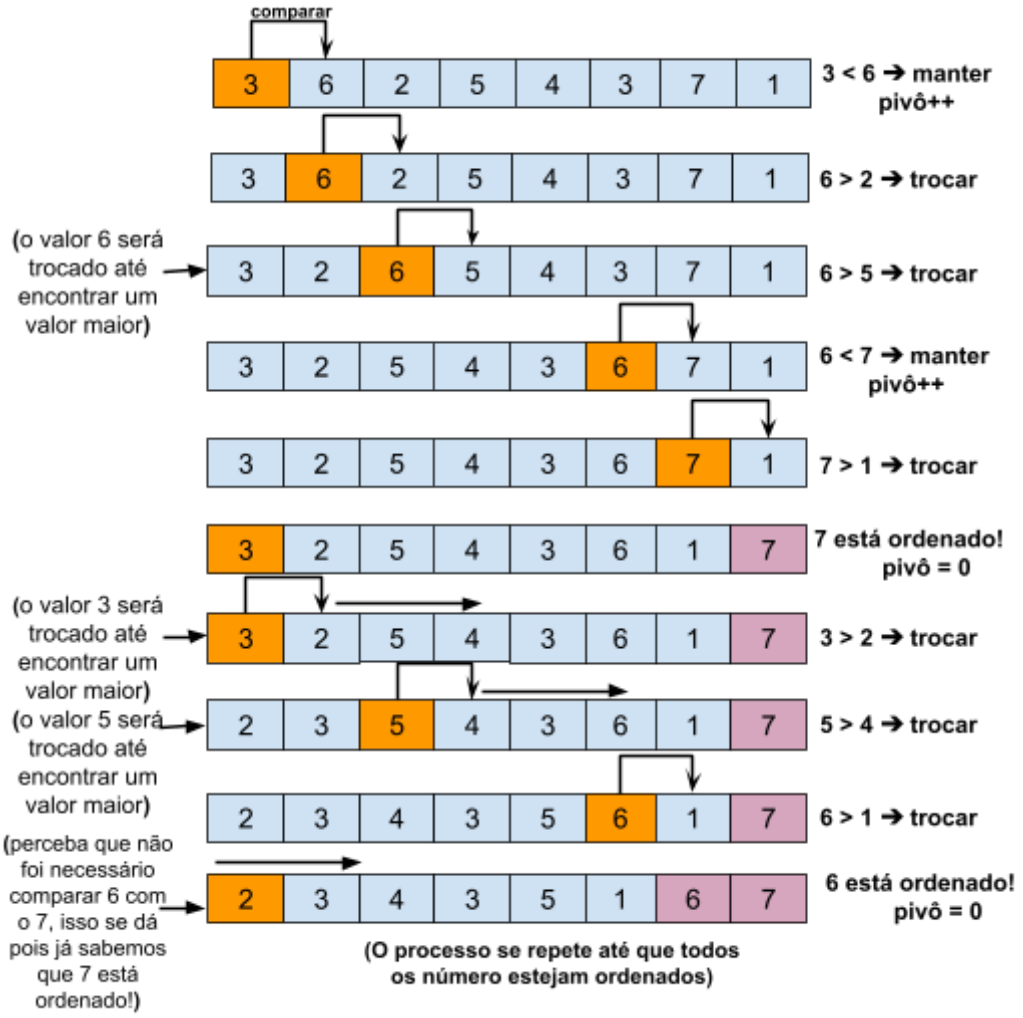
As demonstrações ocorreram das seguintes maneiras: ordenação usando BubbleSort (melhor versão) com o vetor **a**, InsertionSort (in-place, melhor versão) com o vetor **b**, MergeSort com o vetor **a** e QuickSort (s/ randomização de pivô) com o vetor **b.** Vale salientar que para tornar as ilustrações mais intuitivas haverá uma breve explicação do funcionamento de cada algoritmo em seu respectivo tópico. Após as ilustrações também realizaremos uma otimização no algoritmo de quicksort fazendo com que ao invés de seu pivô sempre ser metade do array ele será selecionado aleatoriamente.

Por fim, neste trabalho haverá a realização de testes de performance usando nossos algoritmos de ordenação no qual usaremos o tempo médio que cada algoritmo leva para ordenar um mesmo array.

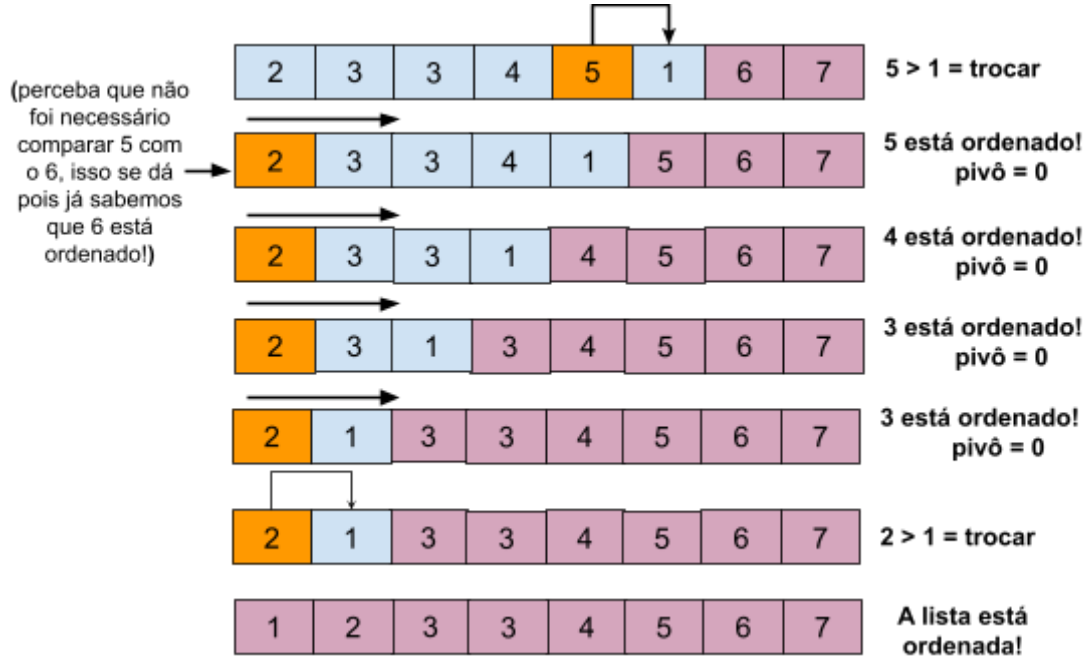
2.1. Ilustrações

2.1.1. BubbleSort

O bubble sort é um algoritmo que varre um vetor **n** vezes, sendo "n" o tamanho do vetor que está sendo percorrido. As varreduras de array funcionam da seguinte maneira: o pivô sempre começará no item de índice 0, ou seja, no início da lista, e o valor do pivô é comparado com o valor do item seguinte. Caso o valor no pivô seja maior que o do ítem comparado os valores são trocados, caso contrário os valores serão preservados em seus lugares. Ao final desse processo, independente se o valor do pivô foi maior ou menor em relação do próximo ítem, o pivô passará a ter o índice de 1, na verdade toda vez que esse processo ocorre o pivô tem o seu valor de índice adicionado aumentado em 1(pivô++). A varredura acaba quando o pivô chega no valor de índice igual a **n - 1 - i**, sendo "i" a iteração que o bubble sort se encontra; ao finalizar o processo de varredura o pivô volta a ter o valor 0 começando uma nova iteração. Abaixo temos uma ilustração de como o bubble sort funciona sobre o **vetor a**, em que os valores com a cor azul estão desordenados, o valor em laranja é o pivô, os valores em roxo são os valores já ordenados. Vale salientar que as setas indicam tanto valores a serem comparados quanto aos serem trocados.

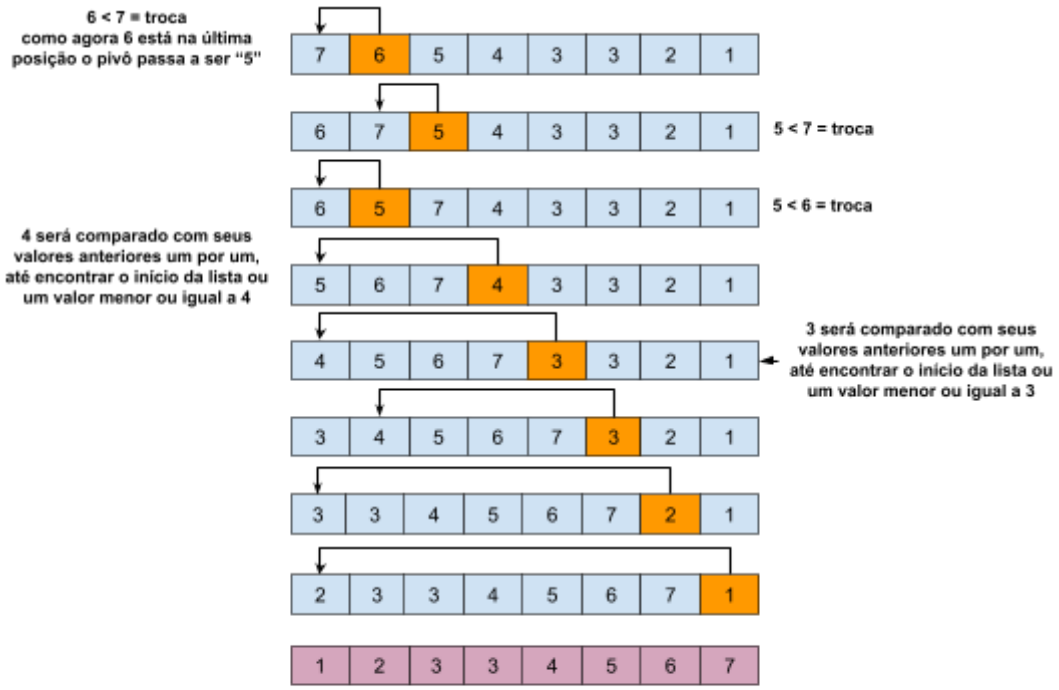


A partir de agora, para tornar mais curta as ilustração iremos apenas mostrar o resultado final de cada iteração:



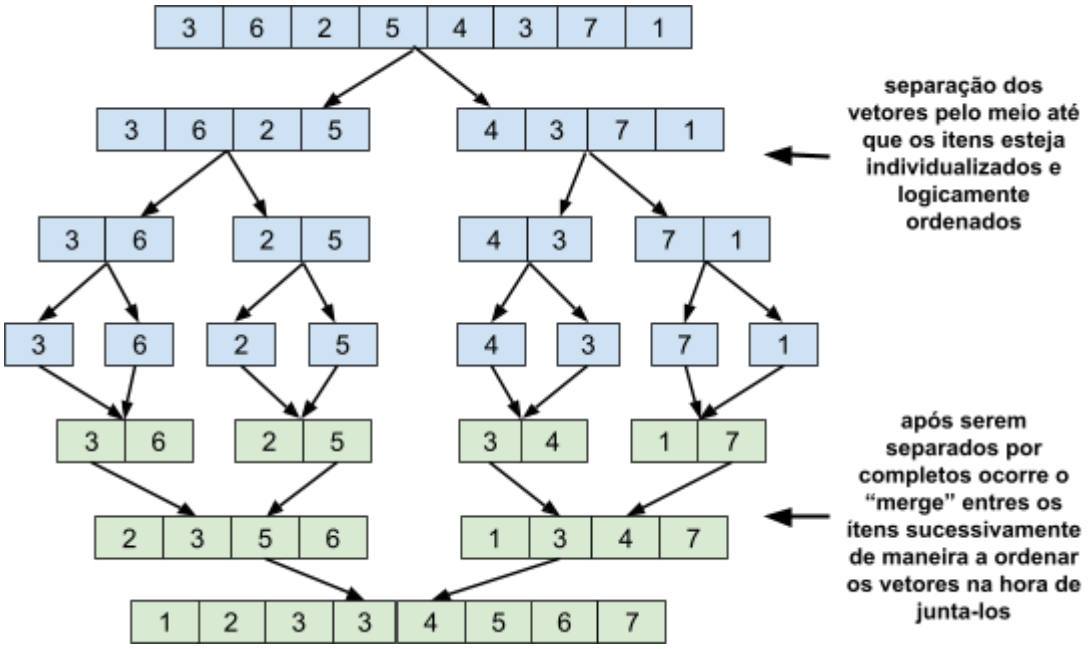
2.1.2. InsertionSort in-place

O algoritmo de InsertionSort realiza **n** iterações de varreduras, sendo “n” o tamanho do vetor, em que em cada uma dessa varreduras o valor do pivô é comparado com o valor de seu ítem antecessor, caso esse valor seja maior que o do pivô os valor trocam e o pivô passa a ser o índice de valor igual a **j - 1**, sendo “j” o valor atual do pivô . Vale salientar que o valor inicial do pivô em cada iteração é igual a **i**(“i” é o valor da iteração, “i” começa com o **1).** O processo de varredura é análogo a **inserir** o pivô na posição ordenada à esquerda. O algoritmo é dado como **in-place** quando não se usa de outro vetor auxiliar para se realizar o algoritmo. Abaixo temos uma ilustração de como o insertion sort funciona sobre o **vetor b**, em que os valores com a cor azul estão desordenados, o valor em laranja é o pivô, os valores em roxo são os valores já ordenados. Vale salientar que as setas indicam tanto valores a serem comparados quanto aos serem trocados.



2.1.3. MergeSort

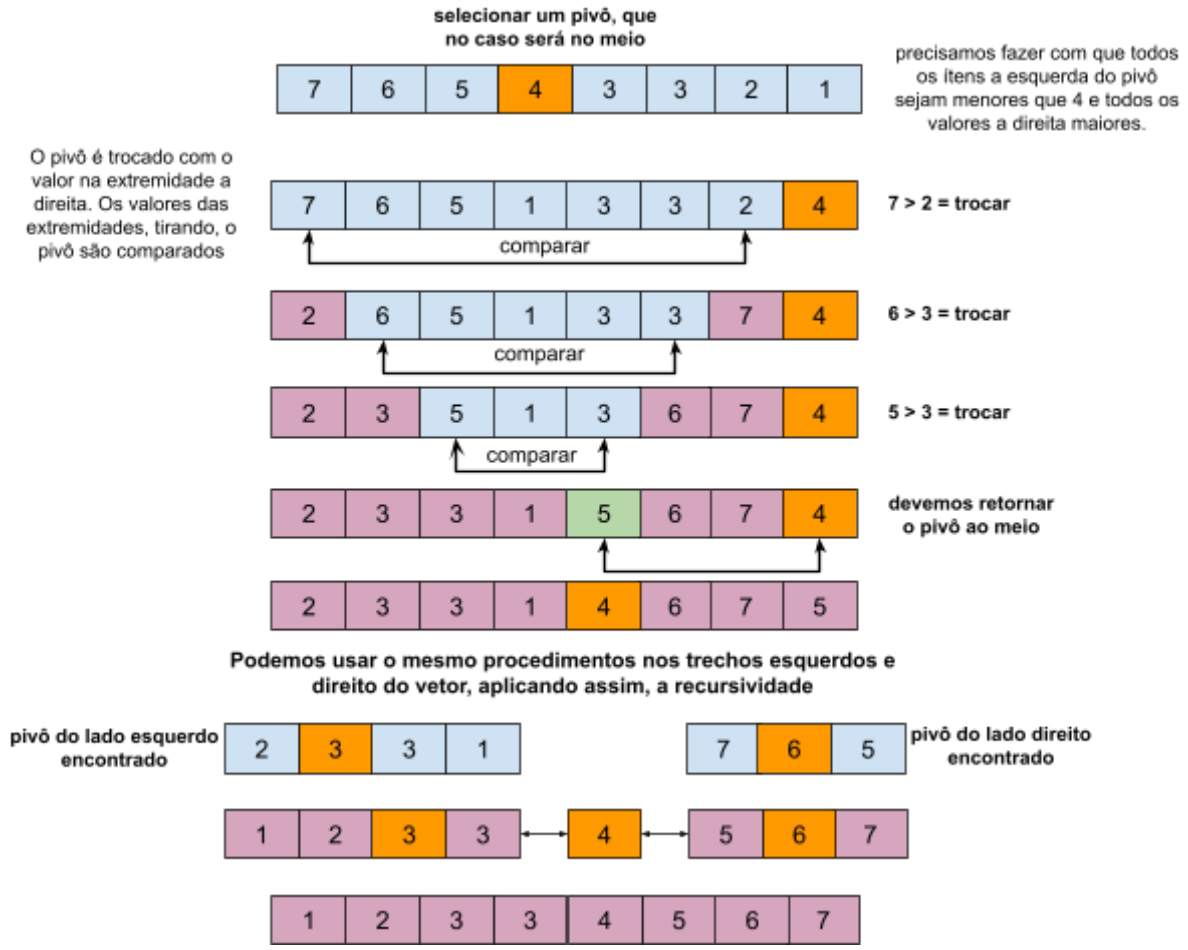
No MergeSort o vetor original é dividido ao meio gerando outros dois vetores, denominados subvetores, esses subvetores por sua vez são divididos em mais outros dois vetores menores. Esses processo de divisão de vetores ocorre até que os vetores resultantes sejam contidos por apenas um valor, dessa maneira cada indivíduo vetor está ordenado. O próximo passo após separar todos os valores do vetor original iremos agrupá-los de maneira a ordena-los.



2.1.4. QuickSort

Assim como nos algoritmos anteriores, o quicksort também possui uma espécie de pivô, porém ele atua de uma maneira diferente. O pivô nesse algoritmo se coloca no meio do vetor a ser ordenado dividindo nosso array em dois, o lado a direita do pivô e o lado a esquerda. Para ordenar nosso array usando quicksort, todos os valores maiores que o nosso pivô selecionado devem ficar a sua direita e todos os valores menores devem ficar à esquerda, porém isso não é o suficiente para ordenar todos os valores do vetor, é aí que entra a recursividade. Usando a recursividade utilizaremos o método explicado para ambos os lados do nosso array em que cada um deles terá um novo pivô que dividirá novamente os valores de cada lado. Esse processo se repetirá até que todos os valores estejam ordenados.

Abaixo a ilustração de como esse processo de QuickSort ocorre:



OBS.: Para tornar a demonstração mais fácil foi utilizado como pivô o valor no meio da lista, porém, no quicksort aprendido em sala de aula, o pivô utilizado é sempre o último valor do vetor.

2.2 Otimizando o QuickSort com pivôs aleatórios

Utilizando o algoritmo padrão do quicksort das aulas observamos que o pivô do nosso vetor a ser ordenado será sempre o elemento que está no final desse vetor, mas a implementação poderia ocorrer em qualquer posição entre o início e o fim de nosso vetor para se tornar o pivô. Veja abaixo a implementação padrão do algoritmo de QuickSort em que usamos o último elemento como pivô:

```
void quickSort(int* v, int ini, int fim){
    if(fim>ini){
        int indexPivo = partition(v,ini,fim);
        quickSort(v,ini,indexPivo-1);
        quickSort(v,indexPivo+1,fim);
    }
}


void swap(int* v, int i, int j){
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}


int partition(int* v, int ini, int fim){
    int pIndex = ini;
    int pivo = v[fim];
    for(int i = ini; i < fim; i++){
        if(v[i] <= pivo){
            swap(v,i,pIndex);
            pIndex++;
        }
    }
    swap(v,pIndex,fim);
    return pIndex;
}
```

(code 1 - código de ordenação com quicksort utilizando C)

No código acima temo a implementação de três funções, uma delas é **swap** que é responsável por trocar dois ítem de determinada posição, temos a função de **partition**, função que é responsável por escolher um pivô, que no caso é sempre o último valor e ordenar o array com os valores menores que esse pivô a esquerda e os maiores a direita retornando a posição final desse pivô após a ordenação, e a função de **quicksort** que é a responsável por realizar o processo de partição para se achar o índex do pivô após a ordenação da lista e aplicar o quicksort recursivamente duas vezes, uma para o lado esquerdo e outra para o lado direito.

Para utilizar um pivô aleatório toda vez que o processo de partição é executado podemos criar mais uma função denominada **partition_random** que utiliza a função partition porém antes de aplicá-lo trocamos de lugar o último valor da nossa lista com um valor de índice aleatório, fazendo o processo equivalente de randomizar usando um índice aleatório. Abaixo a função "partition_random":

```c
int partition_random(int *v, int ini, int fim)
{
    srand(time(NULL));
    int random = ini + rand() % (fim - ini);
    swap(v, random,fim);

    return partition(v,ini, fim);
}
```

(code 2 - código da partição randômica C)

Agora encontramos uma maneira análoga de usarmos uma pivô randômico devemos reestruturar a função de "quicksort" utilizada no code 1, trocando a função de partição normal pela randômica.

```c
void quickSort(int* v, int ini, int fim){
    if(fim>ini){
        int indexPivo = partition_random(v,ini,fim);
        quickSort(v,ini,indexPivo-1);
        quickSort(v,indexPivo+1,fim);
    }
}
```

(code 3 - reestruturação da função de quicksort)

Abaixo como fica o código completo:

```c
void swap(int* v, int i, int j){
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

int partition(int* v, int ini, int fim){
    int pIndex = ini;
    int pivo = v[fim];
    for(int i = ini; i < fim; i++){
        if(v[i] <= pivo){
            swap(v,i,pIndex);
            pIndex++;
```

```
            }
        }
        swap(v,pIndex,fim);
        return pIndex;
}


int partition_random(int *v, int ini, int fim)
{
    srand(time(NULL));
    int random = ini + rand() % (fim - ini);
    swap(v, random,fim);

    return partition(v,ini, fim);
}

void quickSort(int* v, int ini, int fim){
    if(fim>ini){
        int indexPivo = partition_random(v,ini,fim);
        quickSort(v,ini,indexPivo-1);
        quickSort(v,indexPivo+1,fim);
    }
}
```

(code 4 - quicksort com pivô randômico)

3. Testes e Resultados

3.1. Testes

Como dito na introdução, para entender melhor a performance dos algoritmos ilustrados e explicados iremos implementar testes, no caso, iremos testar BubbleSort, InsertionSort, MergeSort e QuickSort(pivô randômico). Os testes performados serão elaborados de maneira a serem iguais e padronizados, assim, poderemos ver e comparar a performance.

Os testes que serão realizados serão envolvendo a ordenação de vetores criados dinamicamente, porém padronizado usando a função **srand** que permite a utilização de seeds para geração de números, dessa maneira todos os arrays que foram criados com a mesma seed irão possuir o mesmos valores, logo, usaremos esse métodos de geração de arrays para testar todos os nossos algoritmos. Para teor comparativo realizaremos os testes de ordenação com três arrays de diferentes tamanhos para cada algoritmo, os tamanhos de arrays serão 10(dez), 1000(mil) e 100000(cem mil). Por fim, para garantir que outliers não sejam escolhidos como tempo de execução, iremos realizar todos os testes 30 vezes com o intuito de pegar o valor total de realização desses 30 testes, a média de tempo de ordenação por teste e a mediana dos testes. Os algoritmos que serão testados serão os de **SelectionSort (in-place), BubbleSort (melhor versão), InsertionSort (in-place, melhor versão), MergeSort, QuickSort, QuickSort (com seleção randomizada de pivô)** e **CountingSort.** Abaixo temos o código que tira o tempo que se leva para a ordenação de um algoritmo, em que a função recebe como parâmetro o tamanho do vetor e a função que iremos utilizar e retorna o tempo tomado para realizar uma única ordenação, essa função é nomeada **bench_mark**.

```c
double bench_mark(int size, void (*fun)(int*, int) ){ // recebe o tamanho do vetor(size)
                                                      // e o algoritmo(fun)
   clock_t timer;
   int* v;

   srand(4135);  // setando a seed que gerará nossa função rand
   v = (int*)calloc(size, sizeof(int));

   for (int i = 0; i < size; i++){
      v[i] = rand() % (100);
   }

   timer = clock();

   fun(v, size);

   timer = clock() - timer;

   free(v);
   return ((double)timer) / CLOCKS_PER_SEC;
}
```

(code 5 - função benchmark que cronometra o tempo de realização de uma função "fun")

A seguir a função para realizar múltiplas vezes a função "bench_mark" somando os resultados de tempo de execução, no final temos o tempo total que o algoritmo "fun" levou para ordenar um array de tamanho "size", **n** vezes. Vale salientar que o valor de repetição do algoritmo ser sempre de 30.

```c
double Test(int size, int n, void (*fun)(int*, int) ){
   double sum_time = 0;

   for (int i = 0; i < n; i++){
      sum_time = sum_time + bench_mark(size, fun);
   }

   return sum_time;
}
```

(code 6 - teste completo)

OBS.: vale salientar que na realização dos testes do quickSort a função bench_mark foi modificada para que o parâmetro respectivo ao algoritmo em que a função ficaria da seguinte maneira: `double bench_mark(int size, void (*fun)(int*, int, int) )`.

Por fim, a chamada das funções de teste ficam da seguinte forma dentro da função "main":

```
int main() {
   int times = 1;
   double time = Test(100000, times, bubbleSort);

   printf("%f segundos\n", time);
   printf("%f segundos por execução\n", time/times);
   return 0;
}
```

(code 7 - chamada da função teste dentro da main)

3.2. Resultados

Como resultado dos testes temos o tempo que cada algoritmo tomou para realizar 30 interações para a ordenação de um vetor com a)10, b)1000 e c)100000 ítens. Para descobrir o tempo médio de um processo de ordenação basta dividir os resultados por 30, como visto no código 7.

3.2.1. BubbleSort

O tempo total de realização de 30 operações de ordenação utilizando o BubbleSort com:

a) 10 itens = 0.000023 **segundos**
b) 1000 itens = 0.078249 **segundos**
c) 100,000 itens = 16.706211 **minutos**

Já o tempo médio de uma iteração utilizando o BubbleSort é:

a) 10 itens ≃ 0.000001 **segundos**
b) 1000 itens = 0.0026083 **segundos**
c) 100,000 ítems = 33.412422 **segundos**

3.2.2. SelectionSort

O tempo total de realização de 30 operações de ordenação utilizando o SelectionSort com:

a) 10 itens = 0.000036 **segundos**
b) 1000 itens = 0.040746 **segundos**
c) 100,000 itens = 6.5607015 **minutos**

Já o tempo médio de uma iteração utilizando o SelectionSort é:

a) 10 itens ≃ 0.000001 **segundos**
b) 1000 itens = 0.0026083 **segundos**
c) 100,000 ítems = 13.121403 **segundos**

3.2.3. InsertionSort

O tempo total de realização de 30 operações de ordenação utilizando o InsertionSort com:

a) 10 itens = 0.000023 **segundos**
b) 1000 itens = 0.022877 **segundos**
c) 100,000 itens = 3.719171 **minutos**

Já o tempo médio de uma iteração utilizando o InsertionSort é:

a) 10 itens ≃ 0.000001 **segundos**
b) 1000 itens = 0.000762 **segundos**
c) 100,000 itens = 7.438342 **segundos**

3.2.4. CoutingSort

O tempo total de realização de 30 operações de ordenação utilizando o CoutingSort com:

a) 10 itens = 0.000039 **segundos**
b) 1000 itens = 0.096710 **segundos**
c) 100,000 itens = 1.537210 **minutos**

Já o tempo médio de uma iteração utilizando o CoutingSort é:

a) 10 itens ≃ 0.000001 **segundos**
b) 1000 itens = 0.0032236 **segundos**
c) 100,000 itens = 2.512490 **segundos**

3.2.5. MergeSort

Agora iremos entrar nos tempos de algoritmos mais eficientes, no caso iremos ver os resultados do MergeSort que tem um desempenho muito superior aos algoritmos de anteriores, tendo um nível de desempenho igual a O.log(n). Abaixo os resultados:

a) 10 itens = 0.000043 **segundos**
b) 1000 itens = 0.000311 **segundos**
c) 100,000 itens = 0.589670 **segundos**

Já o tempo médio de uma iteração utilizando o CoutingSort é:

a) 10 itens ≃ 0.000001 **segundos**
b) 1000 itens = 0.0032236 **segundos**
c) 100,000 items = 0.0196556 **segundos**

3.2.6. QuickSort

Agora iremos entrar nos tempos de algoritmos mais eficientes, no caso iremos ver os resultados do QuickSort que tem um desempenho muito superior aos algoritmos de anteriores, tendo um nível de desempenho igual a O.log(n). Abaixo os resultados:

a) 10 itens = 0.000039 **segundos**

b) 1000 itens = 0.003671 **segundos**
c) 100,000 itens = 10.61844 **segundos**

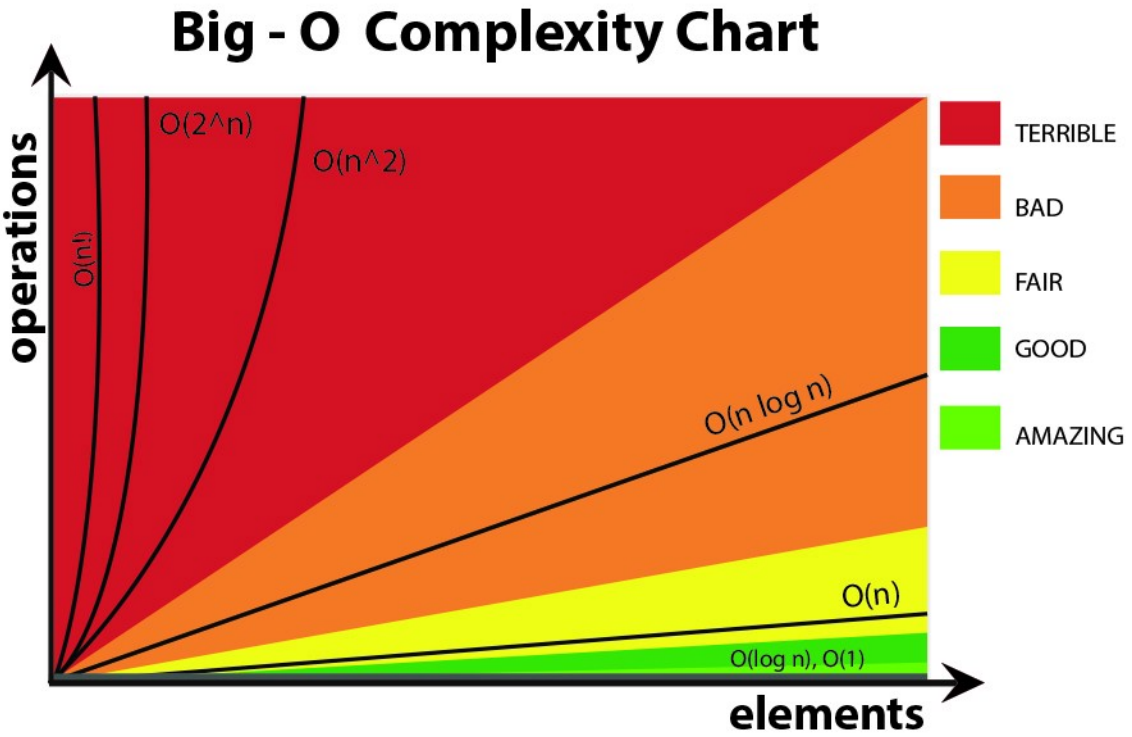
Já o tempo médio de uma iteração utilizando o QuickSort é:

a) 10 itens ≃ 0.000001 **segundos**
b) 1000 itens = 0.000122 **segundos**
c) 100,000 items = 0.353948 **segundos**

Conclusão

Com os resultados obtidos na seção 3.2, podemos comparar a eficiência de nossos algoritmos e ver se o que foi obtido corrobora com o que era esperado.

Podemos concluir a partir dos nossos resultados que os algoritmos de MergeSort e QuickSort levam uma vantagem enorme quando analisamos seus tempos em relação aos demais, principalmente quando aumentamos muito a magnitude do nosso vetor. Esse comportamento é o esperado, pois sabemos que a eficiência tanto do MergeSort quanto do QuickSort é de O(log(n)), ou seja, se torna mais eficiente em relação aos algoritmos com perfomance O(n) conforme o vetor a ser ordenado se torna maior. Esse comportamento em relação os algoritmos O(n) quando termo que ordenar o vetor com tamanho de cem mil em que os algoritmo de BubbleSort e SelectionSort em que os testes tomam mais de 10 minutos para serem performados enquanto que o QuickSort e MergeSort tomam de 0.5 a 10 segundos.



(fig - gráfico comparativo dos algoritmo em relação à notação "big O")