Curso de Engenharia de Computação

Estruturas e Recuperação de Dados B

Projeto 1 - Implementação dos Algoritmos de Ordenação
Prof. Me. Edmar Roberto Santana de Rezende

Equipe:

Alcides Gomes Beato Neto	19060987
Henrique Sartori Siqueira	19240472
Rafael Silva Barbon	19243633

Introdução

O presente projeto tem o objetivo de implementar os algoritmos de ordenação e verificar o desempenho dos mesmos, comparando o tempo de execução de cada um para comparar e realizar o contraste entre cada um.

O desenvolvimento do projeto foi feito no sistema operacional Linux na distribuição Ubuntu 20.04 com o compilador GCC e armazenamento em nuvem via Github. Os valores utilizados nos valores são inteiros não sinalizados de 0 a 2^{32} - 1 (4 294 967 295), gerados através da função *rand*.

Para cada algoritmo de ordenação, foi realizada a tentativa de confecção do algoritmo pela equipe, realizando os testes e apontamentos de erros. Após o algoritmo estar funcionando, foi efetuada uma comparação com o algoritmo original para apontar possíveis otimizações, porém tais otimizações não foram implementadas para manter a originalidade de criação da equipe.

Descrição

Primeiramente, foram criados todos os cinco algoritmos de ordenação, em que para cada um houve o teste de verificação, se o mesmo encontra-se funcionando e realizando sua função de ordenação. Para isso, um vetor com os mesmos valores foi utilizado, com um modelo de *main* registrado a seguir:

```c
int main(){
    unsigned int v[] = {3,67,31,90,1,44,654,89,21,20};

    for(int i = 0; i < 10; i++)
        printf("%d ",v[i]);

    printf("\n\n");

    selection(v,10);

    for(int i = 0; i < 10; i++)
        printf("%d ",v[i]);

    return 0;
}
```

Figura 1. Modelo de main para conferência dos algoritmos.

Os testes contendo os erros de programação e execuções foram registrados a seguir:

Resultados de implementação para o selection sort:

Para conferência do funcionamento do algoritmo:

```
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./selection_sort
3 67 31 90 1 44 654 89 21 20

1 3 21 31 44 67 89 90 654 20 rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./
gcc selection_sort.c -o selection_sort
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./selection_sort
3 67 31 90 1 44 654 89 21 20

1 3 20 21 31 44 67 89 90 654 rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./
```

Figura 2. Execução do selection sort.

Para a primeira tentativa de execução, o último elemento do vetor não foi modificado.

```c
11  void selection(unsigned int vet[], int tam){
12      int i, j, menor;
13      for(i=0; i < tam ; i++){
14          menor = i;
15          for(j = i; j < tam-1; j++ )
16              if(vet[j] < vet[menor])
17                  menor =  j;
18          swap(vet, menor, i);
19      }
20  }
```

Figura 3. Selection sort com o erro destacado.

O erro ocorreu por conta da condição do for mais interno, em que o tamanho foi diminuído em um, não atingindo, assim, o vetor inteiramente.

O algoritmo confeccionado poderia ter sido melhorado, a partir da condição do loop mais externo ser *tam-1* ao invés de *tam* o que economizaria um loop, porém o modelo de algoritmo feito pela equipe foi mantido.

Resultados de implementação para o insertion sort:

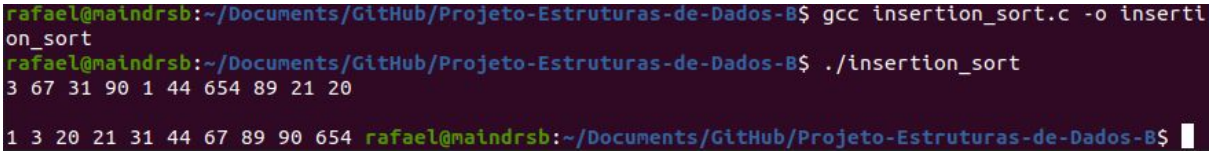


Figura 4. Execução do insertion sort.

O teste da primeira execução do insertion sort foi perfeito e ordenado.

Resultados de implementação do merge sort (Terminal do Linux dentro do VS Code):

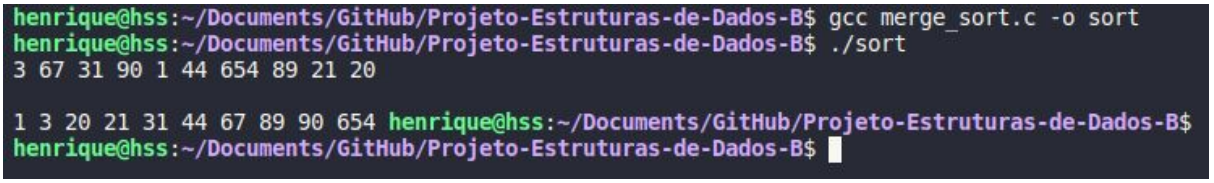


Figura 5. Execução do merge sort.

O teste da primeira execução foi perfeita e ordenada. Entretanto, pode ser feita uma otimização na função mergesort, em que antes de chamá-la recursivamente. Haveria a conferência se o *meio* é menor que o *meio+1*, assim caso duas partições de elementos únicos fossem satisfeitas, não haveria necessidade de chamar a função merge por já estarem ordenados, economizando, assim, chamadas desnecessárias da função merge.

Resultados de implementação do quick sort:

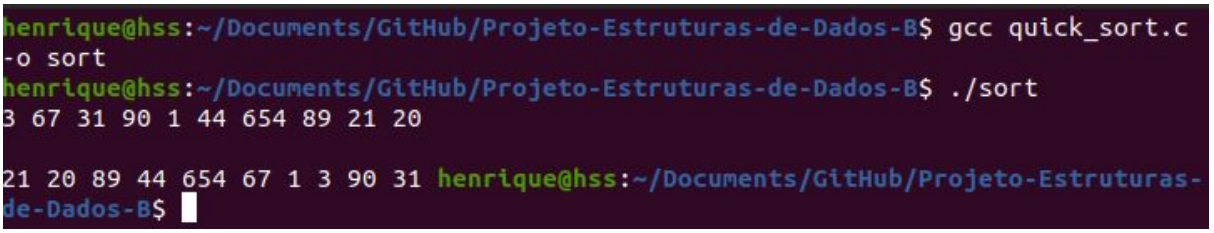


Figura 6. Execução falha do quick sort.

A execução acima apenas moveu alguns dos valores sem que haja o correto procedimento do quick sort, em que valores maiores que o pivô deveriam estar a direita e os valores menores deveriam estar a esquerda.

```c
void quick(unsigned int V[], int comeco, int fim){
    int pivo = (comeco + fim) / 2, i = comeco, j = fim;
    while(i < j) { //while para percorrer todo o vetor
        while(V[i] < V[pivo] && i < j)
            i++;
        while(V[j] < V[pivo] && j < i)
            j--;
        if(i < j){
            swap_quick(V,i,j);
            i++;
            j--;
        }
    }
    if(i < fim)
        quick(V, i, fim);
    if(j > comeco)
        quick(V, comeco, j);
}
```

Figura 7. Erro de comparação no quick sort.

Na figura acima a condição foi realizada de maneira contrária no segundo while, enquanto *v[j]* for maior que *v[pivo]* não é necessário realizar a troca de lado do número, ou seja a condição do while deveria ser *(v[j] > [pivo]) && j >i)*.

```
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ gcc quick_sort.c
-o sort
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort
3 67 31 90 1 44 654 89 21 20

Segmentation fault (core dumped)
```

Figura 8. Erro de execução para o quick sort.

```c
void quick(unsigned int V[], int comeco, int fim){
    int pivo = (comeco + fim) / 2, i = comeco, j = fim;
    while(i < j) { //while para percorrer todo o vetor
        while(V[i] < V[pivo] && i < j)
            i++;
        while(V[j] > V[pivo] && j > i)
            j--;
        if(i < j){
            swap_quick(V,i,j);
            i++;
            j--;
        }
    }
    if(i < fim)
        quick(V, i, fim);
    if(j > comeco)
        quick(V, comeco, j);
}
```

Figura 9. Destaque do erro na comparação do loop.

No código acima as condições do while deveriam adentrar também a igualdade entre o elemento e o pivô, já que este pode mudar de lugar, além foi identificado erro de segmentação na execução do código.

```
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ gcc quick_sort.c
-o sort
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort
3 67 31 90 1 44 654 89 21 20

1 67 3 20 21 31 654 44 89 90 henrique@hss:~/Documents/GitHub/Projeto-Estruturas-
```

Figura 10. Erro de ordenação do algoritmo quick sort.

```c
void quick(unsigned int V[], int comeco, int fim){
    int pivo = (comeco + fim) / 2, i = comeco, j = fim;
    while(i <= j) { //while para percorrer todo o vetor
        while(V[i] < V[pivo] && i < j)
            i++;
        while(V[j] > V[pivo] && j > i)
            j--;
        if(i <= j){
            swap_quick(V,i,j);
            i++;
            j--;
        }
    }
    if(i < fim)
        quick(V, i, fim);
    if(j > comeco)
        quick(V, comeco, j);
}
```

Figura 11. Possível erro do algoritmo quick sort.

Em seguida foi identificado o erro do posicionamento da chave do while, não realizando assim a recursão (no entanto, posteriormente, tal ação realizada alterou o algoritmo que estava certo).

```
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ gcc quick_sort.c
-o sort
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort
3 67 31 90 1 44 654 89 21 20

1 3 20 21 31 44 67 89 90 654 henrique@hss:~/Documents/GitHub/Projeto-Estruturas-
de-Dados-B$ gcc quick_sort.c -o sort
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort
3 67 31 90 1 44 654 89 21 20

1 3 20 21 31 44 67 89 90 654 henrique@hss:~/Documents/GitHub/Projeto-Estruturas-
de-Dados-B$
```

Figura 12. Execução do algoritmo quick sort sem erros.

Resultados de implementação do heap sort:

Para a primeira tentativa de execução, houve a tentativa de implementar a função de ordenação diretamente, o que não surtiu efeito:

```
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ gcc heap_sort.c
-o sort
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort
0 3 67 31 90 1 44 654 89 21 20

0 3 67 31 90 1 44 654 89 21 20 henrique@hss:~/Documents/GitHub/Projeto-Estrutur
```

Figura 13. Tentativa de execução do heap sort.

```
void heap_sort(unsigned int v[], int i, int tam){
    int menor = i;

    //while(i < tam){
        if(v[i*2] < v[menor] && (i*2) < tam){
            swap_heap(v, menor, i*2);
            heap_sort(v,tam,i*2);
        }
        if(v[i*2+1] < v[menor] && (i*2+1) < tam){
            swap_heap(v, menor, i*2+1);
            heap_sort(v,tam,i*2+1);
        }
        //i *= 2;
    //}
}
```

Figura 14. Tentativa de função do algoritmo heap sort.

O código acima apresenta uma tentativa falha de implementar o heap sort.

Após entender o funcionamento do algoritmo, foi criada a implementação de flutuar a metade dos maiores números, o que resultou em falha:

```
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./heap
3 67 31 90 1 44 654 89 21 20

*** stack smashing detected ***: terminated
Aborted (core dumped)
```

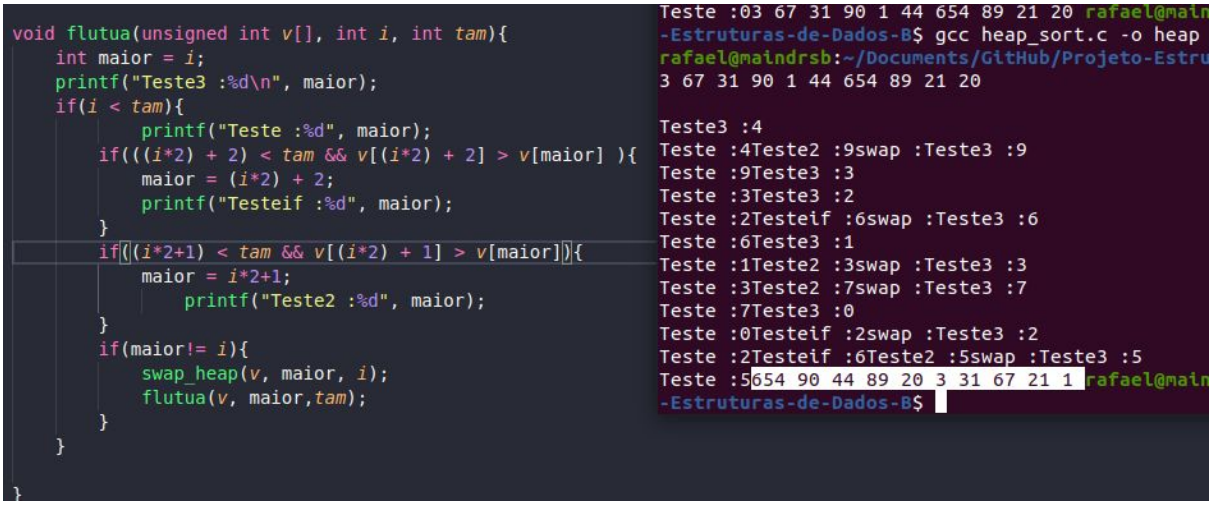
Figura 15. Erro de acesso a memória inexistente.

```
void heap_sort(unsigned int v[], int tam){
    int meio = tam/2;
    for(int i=meio; i >= 0; i--){
        flutua(v, i, tam);
    }
}

void flutua(unsigned int v[], int i, int tam){
    int menor = i;

    if(i < tam){
        if(v[(i*2) + 2] > v[menor] && ((i*2) + 2) > tam){
            menor = (i*2) + 2;

        }
        if(v[i*2+1] > v[menor] && (i*2+1) > tam){
            menor = i*2+1;
        }
        if(menor != i)
            swap_heap(v, menor, i);
        flutua(v,(i*2) + 1,tam);
        flutua(v,(i*2) + 2,tam);
    }
}
```

Figura 16. Código com erro na ordem na comparação das condicionais internas.

O código acima possui o erro nas duas condições mais internas que estão verificando primeiramente o elemento do vetor na posição *(i*2+1)* e *(i*2+2)*, o que não há a devida conferência se a posição de fato está presente no vetor.

Depois de alguns testes, foi percebido que as condições mais internas apresentaram discrepância com relação ao tamanho, em que o resultado da operação de i deveria ser menor que o tamanho do vetor, para isso, utilizamos prints para conferência de entrada na condicional, pois anteriormente o resultado estava com os mesmos valores iniciais do vetor, bem como o erro de execução estava relacionado ao valor *meio* da função *heap_sort*, em que o mesmo não decrementava 1 após a divisão para encontrar a posição mediana no vetor.

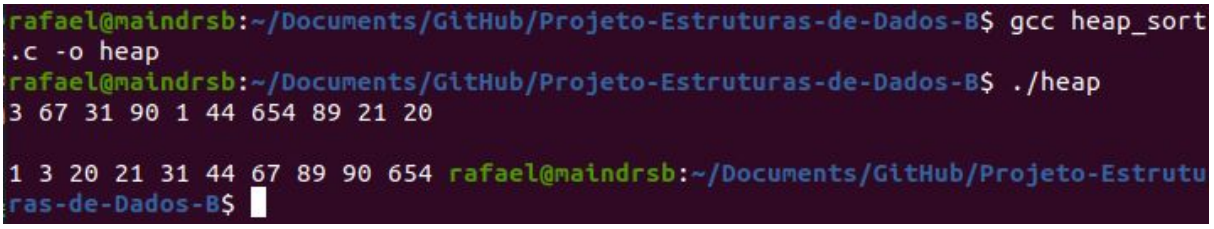


```
void flutua(unsigned int v[], int i, int tam){
    int maior = i;
    printf("Teste3 :%d\n", maior);
    if(i < tam){
            printf("Teste :%d", maior);
        if(((i*2) + 2) < tam && v[(i*2) + 2] > v[maior] ){
            maior = (i*2) + 2;
            printf("Testeif :%d", maior);
        }
        if((i*2+1) < tam && v[(i*2) + 1] > v[maior]){
            maior = i*2+1;
                printf("Teste2 :%d", maior);
        }
        if(maior!= i){
            swap_heap(v, maior, i);
            flutua(v, maior,tam);
        }
    }

}
```

```
Teste :03 67 31 90 1 44 654 89 21 20 rafael@main
-Estruturas-de-Dados-B$ gcc heap_sort.c -o heap
rafael@maindrsb:~/Documents/GitHub/Projeto-Estru
3 67 31 90 1 44 654 89 21 20

Teste3 :4
Teste :4Teste2 :9swap :Teste3 :9
Teste :9Teste3 :3
Teste :3Teste3 :2
Teste :2Testeif :6swap :Teste3 :6
Teste :6Teste3 :1
Teste :1Teste2 :3swap :Teste3 :3
Teste :3Teste2 :7swap :Teste3 :7
Teste :7Teste3 :0
Teste :0Testeif :2swap :Teste3 :2
Teste :2Testeif :6Teste2 :5swap :Teste3 :5
Teste :5654 90 44 89 20 3 31 67 21 1 rafael@main
-Estruturas-de-Dados-B$
```

Figura 17. Teste de execução da função *flutua*.

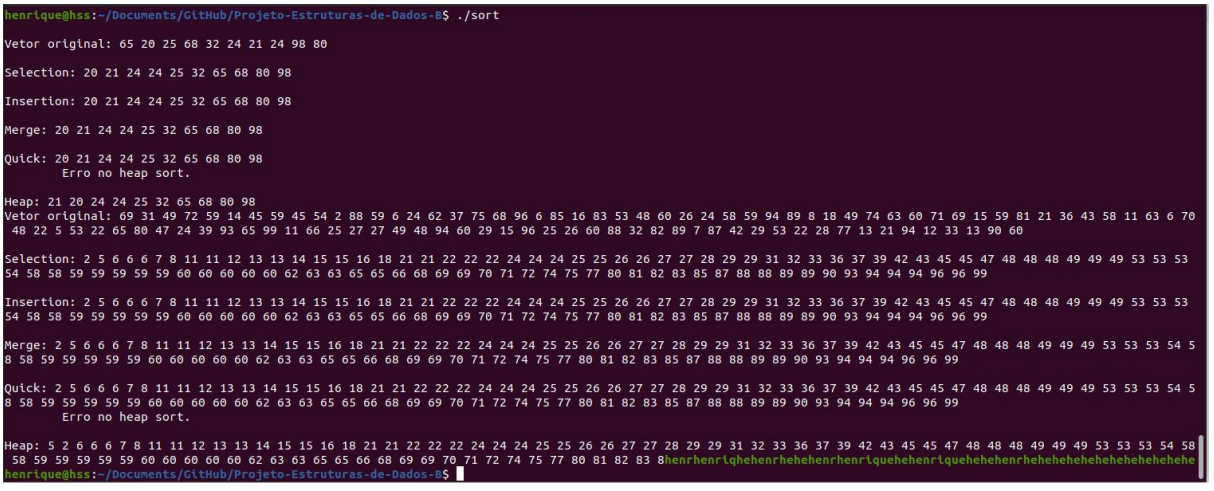
Após descobrir os erros da função de flutuar, pode-se concluir o código fazendo o segundo loop que troca a raiz com o último elemento do vetor e depois realizar novamente a função *flutua* para atualizar o maior valor para a raiz, resultando no seguinte teste abaixo:



```
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ gcc heap_sort
.c -o heap
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./heap
3 67 31 90 1 44 654 89 21 20

1 3 20 21 31 44 67 89 90 654 rafael@maindrsb:~/Documents/GitHub/Projeto-Estrutu
ras-de-Dados-B$
```

Figura 18. Teste bem sucedido do heap sort.

Resultados de implementação do programa principal:



```
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort

Vetor original: 65 20 25 68 32 24 21 24 98 80

Selection: 20 21 24 24 25 32 65 68 80 98

Insertion: 20 21 24 24 25 32 65 68 80 98

Merge: 20 21 24 24 25 32 65 68 80 98

Quick: 20 21 24 24 25 32 65 68 80 98
       Erro no heap sort.

Heap: 21 20 24 24 25 32 65 68 80 98
Vetor original: 69 31 49 72 59 14 45 59 45 54 2 88 59 6 24 62 37 75 68 96 6 85 16 83 53 48 60 26 24 58 59 94 89 8 18 49 74 63 60 71 69 15 59 81 21 36 43 58 11 63 6 70
 48 22 5 53 22 65 80 47 24 39 93 65 99 11 66 25 27 27 49 48 94 60 29 15 96 25 26 60 88 32 82 89 7 87 42 29 53 22 28 77 13 21 94 12 33 13 90 60

Selection: 2 5 6 6 6 7 8 11 11 12 13 13 14 15 15 16 18 21 21 22 22 22 24 24 24 25 25 26 26 27 27 28 29 29 31 32 33 36 37 39 42 43 45 45 47 48 48 48 49 49 49 53 53 53
54 58 58 59 59 59 59 59 60 60 60 60 60 62 63 63 65 65 66 68 69 69 70 71 72 74 75 77 80 81 82 83 85 87 88 88 89 89 90 93 94 94 94 96 96 99

Insertion: 2 5 6 6 6 7 8 11 11 12 13 13 14 15 15 16 18 21 21 22 22 22 24 24 24 25 25 26 26 27 27 28 29 29 31 32 33 36 37 39 42 43 45 45 47 48 48 48 49 49 49 53 53 53
54 58 58 59 59 59 59 59 60 60 60 60 60 62 63 63 65 65 66 68 69 69 70 71 72 74 75 77 80 81 82 83 85 87 88 88 89 89 90 93 94 94 94 96 96 99

Merge: 2 5 6 6 6 7 8 11 11 12 13 13 14 15 15 16 18 21 21 22 22 22 24 24 24 25 25 26 26 27 27 28 29 29 31 32 33 36 37 39 42 43 45 45 47 48 48 48 49 49 49 53 53 53 54 5
8 58 59 59 59 59 59 60 60 60 60 60 62 63 63 65 65 66 68 69 69 70 71 72 74 75 77 80 81 82 83 85 87 88 88 89 89 90 93 94 94 94 96 96 99

Quick: 2 5 6 6 6 7 8 11 11 12 13 13 14 15 15 16 18 21 21 22 22 22 24 24 24 25 25 26 26 27 27 28 29 29 31 32 33 36 37 39 42 43 45 45 47 48 48 48 49 49 49 53 53 53 54 5
8 58 59 59 59 59 59 60 60 60 60 60 62 63 63 65 65 66 68 69 69 70 71 72 74 75 77 80 81 82 83 85 87 88 88 89 89 90 93 94 94 94 96 96 99
       Erro no heap sort.

Heap: 5 2 6 6 6 7 8 11 11 12 13 13 14 15 15 16 18 21 21 22 22 22 24 24 24 25 25 26 26 27 27 28 29 29 31 32 33 36 37 39 42 43 45 45 47 48 48 48 49 49 49 53 53 53 54 58
 58 59 59 59 59 59 60 60 60 60 60 62 63 63 65 65 66 68 69 69 70 71 72 74 75 77 80 81 82 83 8henrhenriqhehenrhehehenrhenriquehehenriquehehehenrhehehehehehehehehehehehehe
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$
```

Figura 19. Teste de execução da main sem o cálculo do tempo.

Foi percebido que o heap sort, por coincidência, apresentou uma ordenação correta para o vetor designado para testes pelo grupo. Resultando em erros de ordenação para outros elementos de um vetor. Assim, o erro do código se encontra destacado a seguir:

```
void heap_sort(unsigned int v[], int tam){
    int meio = (tam/2)-1, i;
    for(i=meio; i >= 0; i--)//Reorganização para a raiz possuir o maior elemento
        flutua(v, i, tam);

    for(i = tam-1; i > 0; i--){
        swap_heap(v, 0, i);//Troca a raiz com  ultima folha do vetor(árvore) (i)
        flutua(v, 0, (i - 1));//Vai da raiz(posição 0) ate i-1
    }
}
```

Figura 20. Erro de implementação do heap sort.

Realizamos mais um teste sem o registro do tempo para conferir se todas as funções estão executando de maneira correta. Para isso, foram inseridas impressões após cada algoritmo, bem como o tamanho do vetor. Porém, o algoritmo merge sort estava demorando muito mais do que o selection e o insertion sort juntos, então foi abortada a execução e checado o algoritmo.

```
gcc -o sort main_p.o selection_sort.o insertion_sort.o merge_sort.o quick_sort.o heap_sort.o -lm
henrique@hss:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ ./sort


        i = 10

        Selection sort OK.
        Insertion sort OK.
        Merge sort OK.
        Quick sort OK.
        Heap sort OK.

        i = 100

        Selection sort OK.
        Insertion sort OK.
        Merge sort OK.
        Quick sort OK.
        Heap sort OK.

        i = 1000

        Selection sort OK.
        Insertion sort OK.
        Merge sort OK.
        Quick sort OK.
        Heap sort OK.

        i = 10000

        Selection sort OK.
        Insertion sort OK.
        Merge sort OK.
        Quick sort OK.
        Heap sort OK.

        i = 100000

        Selection sort OK.
        Insertion sort OK.
^C
```

Figura 21. Tentativa de execução dos algoritmos no programa principal.

```c
void Merge(unsigned int vet[], int comeco, int meio, int fim){
    int i = comeco, j = meio, k = 0;
    unsigned int array[fim+1-comeco]; // vetor auxiliar que tem tamanho de dois subarrays

    while(i < meio && j <= fim){ // Verifica o inicio dos subvetores (já ordenados) e realiza o merge ordenadamente
        if(vet[i] < vet[j])
            array[k++] = vet[i++]; // Só avança com oq foi manipulado
        else
            array[k++] = vet[j++]; // Só avança com oq foi manipulado
    }
    while(i < meio)
        array[k++] = vet[i++]; // Só avança com oq foi manipulado
    while(j <= fim)
        array[k++] = vet[j++]; // Só avança com oq foi manipulado

    for(i = comeco, k = 0; i <= fim; i++,k++) // Transfere os valores do vetor auxiliar para o principal ordenadamente
        vet[i] = array[k];
}

void Merge_sort(unsigned int vet[], int comeco, int fim){
    if(comeco < fim){
        int meio = (comeco + fim) / 2;

        Merge_sort(vet, comeco, meio); // Primeira metade de subvetores
        Merge_sort(vet, meio+1, fim); // Segunda metade de subvetores
        Merge(vet, comeco, meio+1, fim); // Junção dos subvetores
    }
}
```

Figura 22. Destaque de erro no merge sort.

A função *Merge* possuía um erro em que o parâmetro inteiro *meio* estava sendo compartilhado por dois subvetores com tamanhos ligeiramente diferentes, isto é, o subvetor da direita ficaria com mais elementos que o subvetor da esquerda. Como também isso se deu pela chamada recursiva na função *Merge_sort* estava com *meio+1* ao invés de somente meio.

N	Selectionsort	Insertionsort	Mergesort	Quicksort	Heapsort
10	0 s 1 ms	0 s 1 ms	0 s 2 ms	0 s 4 ms	0 s 1 ms
100	0 s 32 ms	0 s 30 ms	0 s 18 ms	0 s 497 ms	0 s 26 ms
1000	0 s 107 ms	0 s 855 ms	0 s 138 ms	0 s 927 ms	0 s 179 ms
10000	0 s 922 ms	0 s 856 ms	0 s 609 ms	13 s 804 ms	0 s 713 ms
100000	15 s -410 ms	19 s 608 ms	0 s 326 ms	2157 s -176 ms	0 s 725 ms

Figura 23. Testes com tempo de execução.

Foi notado que o tempo de execução do quick sort estava com um tempo de execução inesperado.

```c
void quick_sort(unsigned int V[], int comeco, int fim){
    int pivo = V[(comeco + fim) / 2], i = comeco, j = fim;
    while(i <= j){ //while para percorrer todo o vetor
        while(V[i] < pivo && i < fim) // Caso o elemento da esquerda for maior que o pivo, ele não atualiza o valor
            i++;
        while(V[j] > pivo && j > comeco) // Caso o elemento da direita for menor que o pivo, ele não atualiza o valor
            j--;
        if(i <= j){ // Verifica se permanece no loop para poder fazer um swap
            swap_quick(V,i,j); // Realiza o swap e atualiza os valores das posições
            i++;
            j--;
        }
        if(i < fim) // Ordena o subvetor da esquerda
            quick_sort(V, i, fim);
        if(j > comeco) // Ordena o subvetor da direita
            quick_sort(V, comeco, j);
    }
}
```

Figura 24. Algoritmo com a chave de fechamento de loop em local errado.

Entretanto, no momento de checagem do algoritmo foi perceptível que a chave de fechamento do loop estava no local errado, e a mesma estava causando mais recursões que o esperado, fazendo com que o tempo se torne muito maior que os dos algoritmos selection e insertion sort, em que ambos são da complexidade de $O(N^2)$.

Bem como outro erro foi encontrado nas condições se o elemento é maior ou menor que o pivô, em que deveria ser comparado o elemento que percorre com o fim ou começo do subvetor.

Porém, mesmo com tais alterações, foi constatado falha de execução no algoritmo. Coincidentemente, após uma modificação realizada, em que o valor do pivô antes considerado como a posição no vetor foi substituído pelo valor do elemento e isso fez com que o algoritmo realizasse uma boa execução.

```
void quick_sort(unsigned int V[], int comeco, int fim){
    int pivo = (comeco + fim) / 2, i = comeco, j = fim;
    while(i <= j){ //while para percorrer todo o vetor
        while(V[i] < V[pivo] && i < fim) // Caso o elemento da esquerda for maior que o pivo, ele não atualiza o valor
            i++;
        while(V[j] > V[pivo] && j > comeco) // Caso o elemento da direita for menor que o pivo, ele não atualiza o valor
            j--;
        if(i <= j){ // Verifica se permanece no loop para poder fazer um swap
            swap_quick(V,i,j); // Realiza o swap e atualiza os valores das posições
            i++;
            j--;
        }
    }
    if(i < fim) // Ordena o subvetor da esquerda
        quick_sort(V, i, fim);
    if(j > comeco) // Ordena o subvetor da direita
        quick_sort(V, comeco, j);
}
```

Figura 25. Manipulação da variável pivô.

Tal efeito deve-se, possivelmente, ao fato de que os elementos poderiam estar muito distantes na memória fazendo com que ocasionasse erros de acesso a mesma.

Outro erro ocorrido foi a consideração de micro como mili, no momento de calcular o tempo de execução de cada algoritmo, em que era considerado apenas a parte de microssegundos ao invés de dividir tal valor por 1000 para que resulte em milissegundos. Além disso, o fato do tempo de execução estar negativo (figura 23) se dá pelo valor inteiro, em que a variável atingiu o último bit, sendo este representado pelo bit de sinal, para corrigir tal erro, foi necessário substituir do tipo inteiro para o tipo inteiro longo.

```
void calcular(struct timeval comeco, struct timeval fim, int *mili,int *seg){
    *seg = fim.tv_sec - comeco.tv_sec;
    *mili = fim.tv_usec - comeco.tv_usec;
    *mili %= 1000;
}
```

Figura 26. Destaque ao operador usado de forma errada para o cálculo do tempo.

Conclusão

Com desenvolvimento dos algoritmos e a análise dos mesmos, foi possível observar na prática os custos computacionais requeridos por esses algoritmos. Com o intuito de verificar se a diferença entre especificações dos processadores utilizados implicaria em uma diferença significativa entre os tempos de execução dos algoritmos de ordenação, a execução do programa foi realizada por todos os integrantes da equipe, comprovando a influência das especificações do processador no tempo de execução dos algoritmos de ordenação.

Intel i7 - 7500U

Caso médio:

```
rafael@maindrsb:~/Documents/GitHub$ cd Projeto-Estruturas-de-Dados-B/
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ make
gcc -c main_p.c
gcc -c selection_sort.c
gcc -c insertion_sort.c
gcc -c merge_sort.c
gcc -c quick_sort.c
gcc -c heap_sort.c
gcc -o sort main_p.o selection_sort.o insertion_sort.o merge_sort.o quick_sort.o heap_sort.o -lm
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ make run
./sort

       N              Selectionsort          Insertionsort          Mergesort              Quicksort              Heapsort
      10                     0 ms                   0 ms                   0 ms                   0 ms                   0 ms
     100                     0 ms                   0 ms                   0 ms                   0 ms                   0 ms
    1000                     6 ms                   6 ms                   0 ms                   0 ms                   0 ms
   10000                   134 ms                 144 ms                   1 ms                   1 ms                   2 ms
  100000                 11357 ms               14289 ms                  17 ms                  14 ms                  27 ms
 1000000               1457877 ms             1754002 ms                 246 ms                 189 ms                 666 msrafael@m
Projeto-Estruturas-de-Dados-B$
```

Figura 27. Execução do caso médio.

Melhor caso:

```
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ make run
./sort

       N              Selectionsort          Insertionsort          Mergesort              Quicksort              Heapsort
      10                     0 ms                   0 ms                   0 ms                   0 ms                   0 ms
     100                     0 ms                   0 ms                   0 ms                   0 ms                   0 ms
    1000                     2 ms                   0 ms                   0 ms                   0 ms                   0 ms
   10000                   129 ms                   0 ms                   0 ms                   0 ms                   1 ms
  100000                 11812 ms                   0 ms                   9 ms                   4 ms                  22 ms
 1000000               1190650 ms                   2 ms                 111 ms                  46 ms                 249 msrafael
Projeto-Estruturas-de-Dados-B$
```

Figura 28. Execução do melhor caso.

Pior caso:

```
rafael@maindrsb:~/Documents/GitHub/Projeto-Estruturas-de-Dados-B$ make run
./sort

       N              Selectionsort          Insertionsort          Mergesort              Quicksort              Heapsort
      10                     0 ms                   0 ms                   0 ms                   0 ms                   0 ms
     100                     0 ms                   0 ms                   0 ms                   0 ms                   0 ms
    1000                     5 ms                  19 ms                   0 ms                   0 ms                   0 ms
   10000                   200 ms                 321 ms                   0 ms                   0 ms                   1 ms
  100000                 13168 ms               35103 ms                  11 ms                   4 ms                  23 ms
 1000000               1336740 ms             3158112 ms                 131 ms                  47 ms                 282 msrafael@m
Projeto-Estruturas-de-Dados-B$
```

Figura 29. Execução do pior caso.

Intel i5 - 3330

Caso médio:

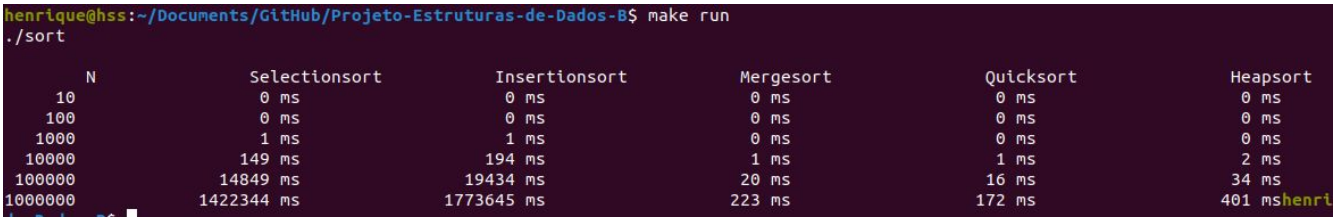


Figura 30. Execução do caso médio.

Melhor caso:

Figura 31. Execução do melhor caso.

Pior caso:

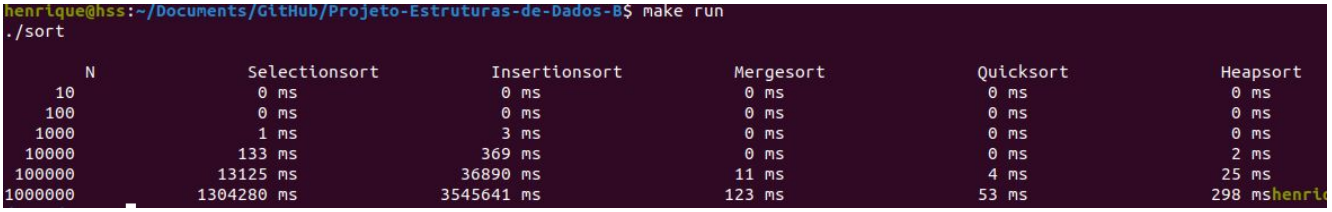


Figura 32. Execução do pior caso.

Intel I7 - 8550U

Caso médio:

Figura 33. Execução do caso médio.

Melhor caso:

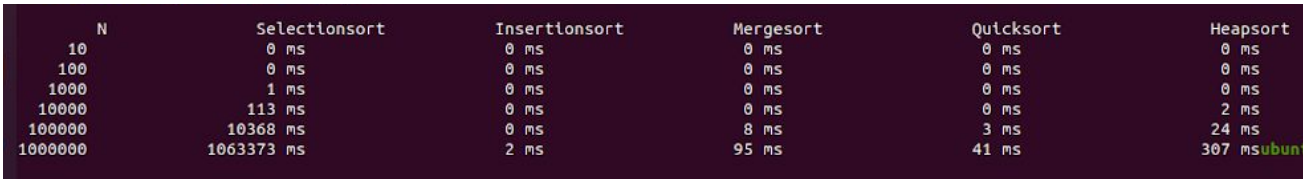
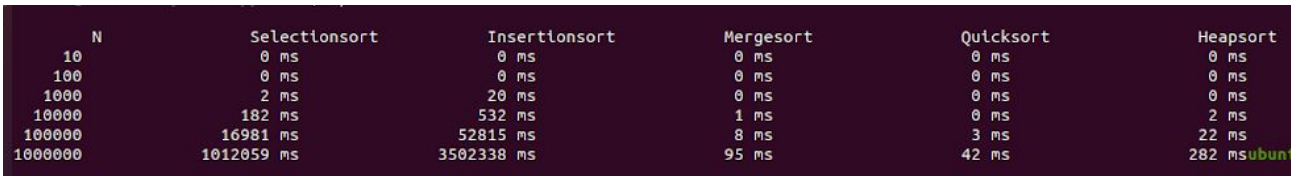


Figura 34. Execução do melhor caso.

Pior caso:



N	Selectionsort	Insertionsort	Mergesort	Quicksort	Heapsort
10	0 ms	0 ms	0 ms	0 ms	0 ms
100	0 ms	0 ms	0 ms	0 ms	0 ms
1000	2 ms	20 ms	0 ms	0 ms	0 ms
10000	182 ms	532 ms	1 ms	0 ms	2 ms
100000	16981 ms	52815 ms	8 ms	3 ms	22 ms
1000000	1012059 ms	3502338 ms	95 ms	42 ms	282 ms

Figura 35. Execução do pior caso.

Além disso foi possível concluir a importância da necessidade de realizar o teste dos algoritmos para diferentes vetores, esta conclusão foi determinada devido ao erro obtido na execução do algoritmo heap sort para um vetor diferente do utilizado na função main em que foi utilizado apenas um vetor para o teste dos algoritmos no seu desenvolvimento.

Após os testes realizados, foi confeccionada uma tabela para melhor visualização e análise dos dados, que foram organizados de acordo com processador [Intel i7 - 7500U | Intel i5 - 3330 | Intel i7 - 8550U] e destacando os piores tempos com a cor vermelha e os melhores com a cor verde.

Caso médio	Selection	Insertion	Merge	Quick	Heap
10	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
100	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
1000	6 | 1 | 2	6 | 1 | 2	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
10000	134 | 149 | 123	144 | 194 | 179	1 | 1 | 1	1 | 1 | 1	2 | 2 | 2
100000	11357 | 14849 | 10745	14289 | 19434 | 17598	17 | 20 | 15	14 | 16 | 13	27 | 34 | 29
1000000	1457877 | 1422344 | 1245559	1754002 | 1773645 | 1706771	246 | 223 | 197	189 | 172 | 171	666 | 401 | 410

Malhor caso	Selection	Insertion	Merge	Quick	Heap
10	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
100	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
1000	2 | 1 | 1	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
10000	129 | 150 | 113	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	1 | 2 | 2
100000	11812 | 14622 | 10368	0 | 0 | 0	9 | 11 | 8	4 | 4 | 3	22 | 28 | 24
1000000	1190650 | 1416586 | 1063373	2 | 2 | 2	111 | 128 | 95	46 | 53 | 41	249 | 317 | 307

Pior Caso	Selection	Insertion	Merge	Quick	Heap
10	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
100	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
1000	5 | 1 | 2	19 | 3 | 20	0 | 0 | 0	0 | 0 | 0	0 | 0 | 0
10000	200 | 133 | 182	321 | 369 | 532	0 | 0 | 1	0 | 0 | 0	1 | 2 | 2
100000	13168 | 13125 | 16981	35103 | 36890 | 52815	11 | 11 | 8	4 | 4 | 3	23 | 25 | 22
1000000	1336740 | 1304280 | 1012059	3158112 | 3545641 | 3502338	131 | 123 | 95	47 | 53 | 42	282 | 298 | 282

Figura 36. Tabela com os tempos de execução.

Na realização dos teste, estes foram realizados para o melhor, pior e médio caso, sendo possível observar a variação do tempo de execução devido a maneira em que os dados estão armazenados no vetor de números aleatórios (caso médio),

e também o contraste de desempenho entre os diferentes processadores de uma mesma marca (denotado pelos piores e melhores casos). Entretanto, para o melhor e o pior caso o intervalo de números do vetor é denotado pelo tamanho do vetor, ou seja, um vetor de mil posições teria os elementos para tais casos descritos de 0 a 999 para o melhor caso e 999 a 0 para o pior caso.

O melhor tempo registrando, quando utilizado o vetor de 1 000 000 de posições, para o melhor caso foi no algoritmo do insertion sort, compatível com a teoria, já que sua complexidade no melhor caso é O(N). Foi possível concluir também que o quick sort foi o algoritmo que executou em menor tempo para o pior e médio caso, que também é condizente, em partes, com a teoria, possuindo em seu pior caso $O(N^2)$ e caso médio O(N log N). Os algoritmos selection e insertion sort foram os que demandaram o maior tempo de execução, já que possuem suas complexidades de $O(N^2)$ para o pior e caso médio.

Foi notado que certos processadores obtiveram tempos melhores para certos algoritmos, como observado na tabela apresentada acima, porém o tamanho influencia diretamente no tempo de processamento, independente do desempenho do processador.

A diferença da ordenação de vetores entre um vetor com poucos elementos e vetores com muitos elementos foi facilmente observada na execução dos algoritmos.

Dessa forma o desenvolvimento do projeto permitiu uma melhor compreensão da importância e da implementação dos algoritmos de ordenação e também a percepção prática do custo computacional na ordenação de vetores com muitas posições, sendo ela bem maior, dependendo do custo do algoritmo, quando comparada com o custo computacional exigido para ordenação de vetores com tamanho menor, bem como o hardware do computador influencia diretamente no tempo de execução do algoritmo.