UNIPAC

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

4° PERÍODO 2021/02

DISCENTES:

BERNARDO RESENDE ANDRÉS

CLAUDIMAR JOSÉ DA CRUZ

RAFAEL DE SOUZA DAMASCENO

MÉTODOS DE ORDENAÇÃO

Atividade da 2°etapa para a
aprovação da disciplina de Estrutura
de dados, ministrada pelo
Prof. Nairon Neri Silva .

UNIVERSIDADE PRESIDENTE ANTÔNIO CARLOS

Barbacena – 2021

Descrição da Atividade

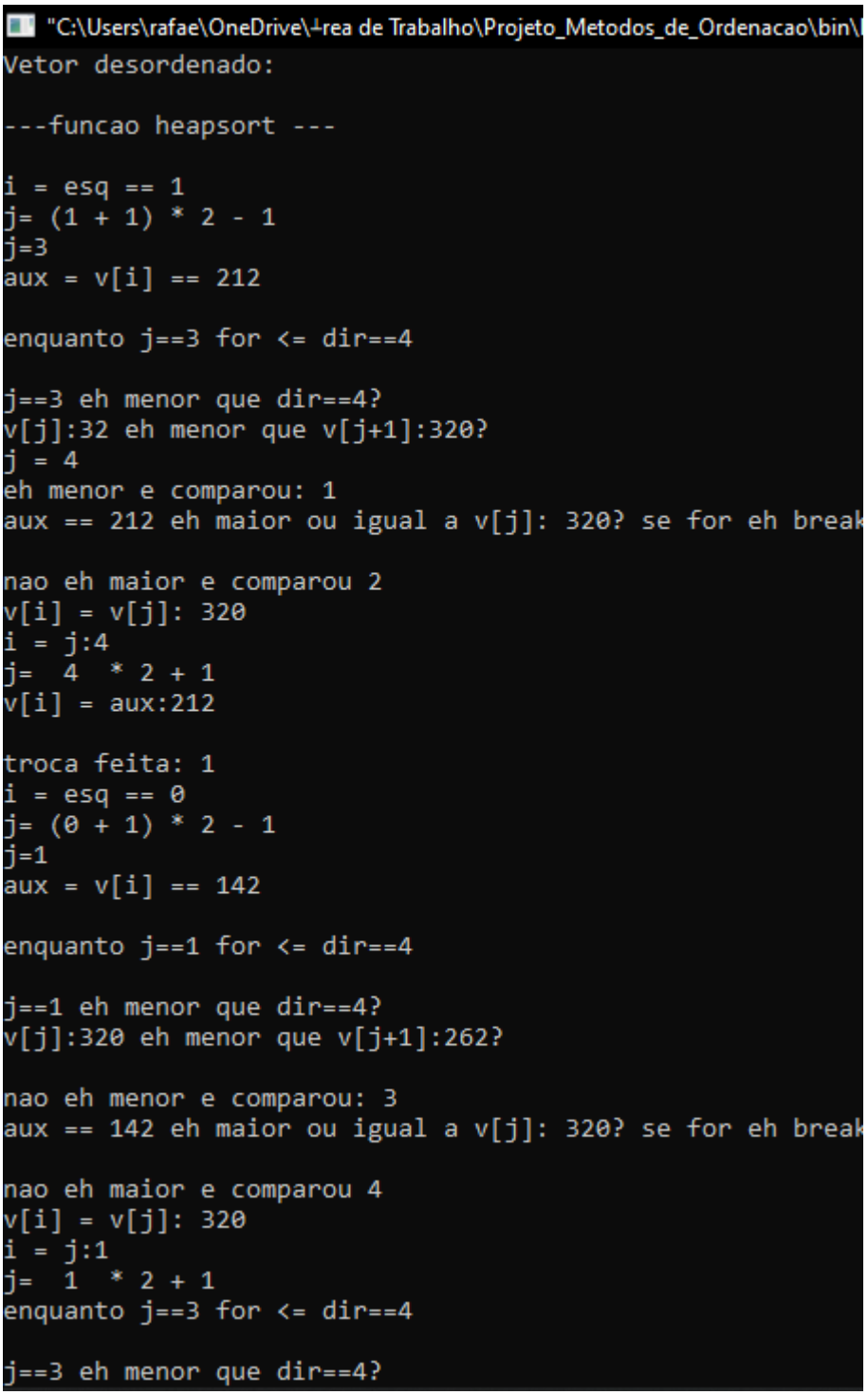
A atividade tem como propósito o uso dos Métodos de Ordenação, nela faremos testes comparativos entre eles, com relação ao números de trocas e de comparações envolvendo os elementos de um vetor gerado para análise. Os métodos utilizados serão os seguintes: Selection Sort, Insertion Sort, Bubble Sort, Shell Sort, Quick Sort, Heap Sort e Merge Sort. Para obter os dados para realizar as comparações será necessário alterações no código de cada método de ordenação, e todos devem constar em um único projeto do CodeBlocks.

Cada método receberá quatro vetores, que será gerado randomicamente, os vetores serão de tamanhos 100, 1000, 10000 e 100000. Para cada método receber o mesmo vetor, haverá cópias do vetor original.

Para a demonstração dos testes será usado tabelas e gráficos gerados com os dados obtidos, que serão enviados em um relatório, que conterá também as observações que o grupo fará em relação ao desenvolvimento e conclusão da atividade.

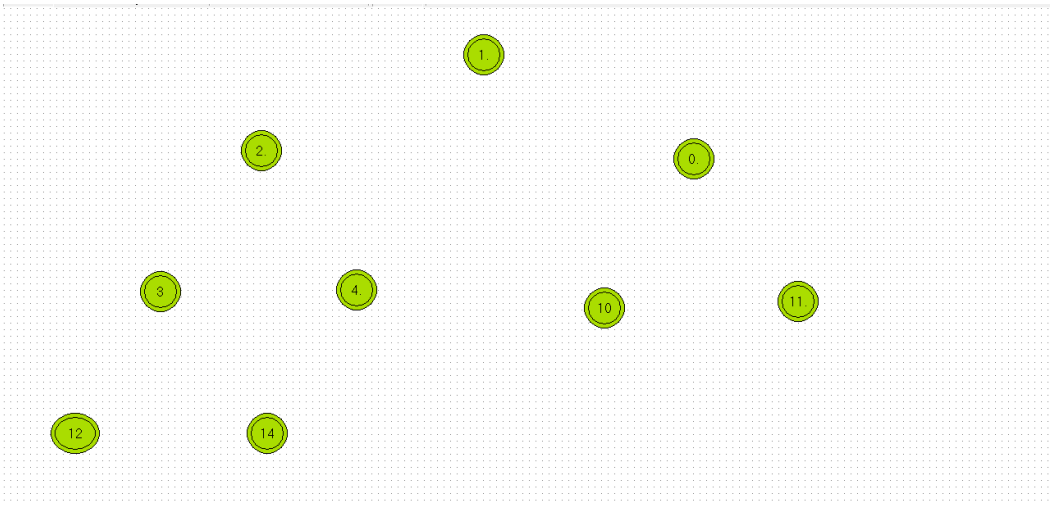
Dificuldades Encontradas e Soluções Aplicadas

- Para entendermos o funcionamento dos métodos de ordenação imprimimos o passo a passo de cada um no terminal.



- Como não sabíamos os resultados de cada teste, utilizamos o Bloco de Notas, Excel e simulador de autômatos (no teste do método Heap) para fazermos a contagem das trocas e comparações.

```
h = 4 = [3] [2] [1] [6] [14] [22] [5] [30] = 4 comparações / 2 trocas

h = 2 = [1] [2] [3] [6] [5] [22] [14] [30] = 6 comparações / 2 trocas

h = 1 = [3] [2] [1] [6] [14] [22] [5] [30]

[3] [2] [1] [6] [14] [22] [5] [30]

[2] [3] [1] [6] [14] [22] [5] [30] = 1

[1] [2] [3] [6] [14] [22] [5] [30] = 3

[1] [2] [3] [6] [14] [22] [5] [30] = 4

[1] [2] [3] [6] [14] [22] [5] [30] = 5

[1] [2] [3] [6] [14] [22] [5] [30] = 6

[1] [2] [3] [5] [6] [14] [22] [30] = 10

[1] [2] [3] [5] [6] [14] [22] [30] = 11

trocas = 5 trocas
```



Simulador de Autômatos

- Para gerar o mesmo vetor para todos os métodos utilizamos uma função, onde é possível escolher o tamanho do vetor e seu rand.

```
28  void original(int vetor[], int tamVetor, int r)
29  {
30      int i;
31      srand(time(NULL));
32
33      for(i=0 ; i < tamVetor ; i++)
34      {
35          vetor[i] = rand()%r;
36      }
37  }
```

- Na tela inicial do programa temos um menu onde temos a opção de escolher o tamanho do vetor que será testado.

- Em alguns métodos o resultado ultrapassou o limite alocado para uma variável do tipo *int,* então foi necessário utilizar o *unsigned long int* . No *unsigned* o número guardado não deverá ter sinal, aumentando o valor máximo devido a ausência de números negativos. Já o *long* aumenta o espaço tomado pela variável.

- No método Merge Sort ficamos em dúvida quanto ao número de trocas, como ele cria vetores auxiliares para trocar os elementos, não tínhamos certeza se essas trocas deveriam entrar na contagem. Mais uma vez utilizamos o Bloco de Notas para termos uma base dos resultados.

- No método Bubble Sort em algumas entradas de dados o método não fazia as devidas ordenações corretas, desse modo usamos o método em que ele não desconsidera os elementos à direita já ordenados.

Métodos de ordenação

Selection Sort

O Selection Sort é um algoritmo de ordenação geralmente usado em vetores de tamanhos pequenos, tem como princípio de funcionamento sempre passar o menor valor de um vetor para a primeira posição. Possui na sua estrutura dois laços *for,* o laço externo controla o índice inicial e o interno percorre todo o vetor. A cada iteração é achado o menor elemento que é colocado na posição inicial (primeira vez é [0]) , como esse elemento já está na posição correta ela não será mais conferida, na próxima iteração o menor elemento será colocado na posição [1] do vetor, indo sucessivamente assim até o vetor ficar ordenado.

Entre suas vantagens é que ele ocupa menos memória (não utiliza vetor auxiliar), é eficiente com vetores de tamanhos pequenos. Entre as desvantagens temos que ele não é estável, pouco eficiente com vetores muito grandes

```
40  void selectionSort(int a [], int n)
41  {
42      int i, j, min, aux;
43      for(i= 0 ; i < n - 1 ; i++)
44      {
45          min = i;
46
47          for(j=i+1 ; j < n ; j++)
48          {
49              if(a[j] < a[min])
50              {
51                  min = j;
52              }
53              // comparação
54              contadorComparacaoSelect++;
55          }
56          if(a[min] == a[i])
57          {
58              aux = a[min];
59              a[min] = a[i];
60              a[i] = aux;
61          }else
62          {
63              aux = a[min];
64              a[min] = a[i];
65              a[i] = aux;
66              contadorTrocaSelect++;
67          }
68      }
69  }
```

Insertion Sort

O Insertion Sort é conhecido por ter seu funcionamento parecido com a organização de cartas de um baralho. A comparação começa na posição [1] do vetor sempre com o elemento à sua esquerda, se o elemento à esquerda for maior a troca é realizada, passando para a posição [2] do vetor ela será conferida com a posição [1], caso a posição [1] maior a conferência passa para a posição [0], de acordo com o resultado dessas comparações os elementos vão encontrando o seu lugar correto, essa conferência é feita em todo o vetor até estar ordenado.

Uma de suas vantagens é que ele é estável, é eficiente quando o arquivo já está quase ordenado. Entre as desvantagens é que ele não é muito eficiente com vetores maiores.

```
void insertionSort(int v[], int n)
{
  //printf("\n contINS:%d",contadorComparacaoInsert);

   int i, j, valorAtual;
   int k = 0;
   unsigned long int p = 0;
   int t = 0;

   //int teste =0;

   for(i=1; i<n; i++)
   {
      valorAtual = v[i];
      j = i - 1;

      while((j >= 0) && (valorAtual < v[j]))
      {
      //printf("\nentrada do vetor: v[j] : %d\n", v[j]);
      //printf("\nentrada do vetor: v[j -h] : %d\n", v[j+1]);

      v[j+1] = v[j];
      j--;
      k++;
      //printf("\n i: %d / k: %d\n", i, k);
      t = k;
      }
      //printf("\nSiada de k: %d\n", k);
      if(t > 0)
      {
         if(i > k  )
         {
            k = k + 1;
         }
         //printf("if: [%d] [p]:%d + [k]:%d",i, p, k);
         p =  p + k;
         // printf("\n p = %d", p);
         // printf("== %d\n", p);
      }else
      {
         //k = k + 1;
         // printf("\nelse: [%d] [p]: %d + 1",i, p);
         p = p + 1;
         //printf("== %d\n", p);
         //printf(" \np = %d", p);
      }
      k =0;
      t =0;

      if(v[j+1] != valorAtual)
      {
         v[j+1] = valorAtual;
         contadorTrocaInsert++;
      }else
      {
         v[j+1] = valorAtual;
         // k++;
      }
         // k++;
   }
      contadorComparacaoInsert = p;
}
```

Quick Sort

O algoritmo Quicksort é um método de ordenação criado por Charles Antony Richard Hoare, com o intuito de tentar traduzir um dicionário de inglês para russo. O quicksort utiliza o método de divisão e conquista , ou seja, um algoritmo recursivo. O trabalho acontece todo na etapa da divisão do vetor. A estratégia acontece em rearranjar as chaves de modo que as chaves menores precedem as chaves maiores. Existe um elemento da lista, chamado pivô, onde que a partir do momento em que houve as partições, trocas e comparações, todos os elementos antes do pivô devem ser menores que o mesmo.

```
int particao(int v[], int esq, int dir)
{
    int pivo, i, j, aux;
    //printf("pivo: %d\ni= %d\nj = %d", v[esq], esq, dir );
    pivo = v[esq];
    i = esq;
    j = dir;
    //printf("\nenquanto [i] = %d for menor que [j] = %d faca:", i,j);

    while(i<j)
    {
        //printf("\nenquanto v[i] <= %d < pivo = %d e [i]:%d <= dir: %d, faca:", v[i],pivo, i, dir);
        while((v[i] <= pivo) && (i<=dir))
        {
            //printf("\nv[i] <= %d < pivo = %d e [i]:%d <= dir: %d?\n",v[i], pivo, i, dir);
            contadorComparacaoQuick++;
            //printf("\ncomparação: %d\n", contadorComparacaoQuick);
            i++;
            //printf("\ni = %d\n", i);
            //printf("\nv[i] <= %d < pivo = %d e [i]:%d <= dir: %d?\n",v[i], pivo, i, dir);
        }
        if((v[i] > pivo)&& (i<=dir))
        {
            //printf("\nFINISH WHILE V[i] = %d eh maior que pivo = %d\n", v[i], pivo);
            contadorComparacaoQuick++;
            //printf("\ncomparação: %d\n", contadorComparacaoQuick);
        }
            //printf("enquanto v[j]: %d > pivo: %d faca:", v[j], pivo);
        while(v[j] > pivo)
        {
            //printf("\nv[j]:%d > pivo = %d?\n",v[j], pivo);
            j--;
            contadorComparacaoQuick++;
            //printf("\ncomparação: %d\n", contadorComparacaoQuick);
            //printf("\nj--: %d\n", j);
            //printf("\nv[j]:%d > pivo = %d?\n",v[j], pivo);
        }
        if((v[j] <= pivo) && (i<=j))
        {
            //printf("\nFINISH WHILE V[J] = %d eh Maior que pivo = %d\n", v[j], pivo);
          contadorComparacaoQuick++;
            //printf("\ncomparação: %d\n", contadorComparacaoQuick);
        }
            //printf("\nse i:[%d] < j:[%d] faca:", i,j);
        if(i<j)
        {
            //printf("\naux = %d\nv[i] = %d\nv[j] = %d", v[i], v[j], aux);
            //printf("\naux = v[i]: %d", v[i]);
            aux = v[i];
            v[i] = v[j];
            //printf("\nv[i] = v[j]:%d", v[j]);
            v[j] = aux;
            //printf("\nv[j] = aux: %d", aux);
            contadorTrocaQuick++;
            //printf("\ntrocas: %d\n", contadorTrocaQuick);
        }
            //contadorTrocaQuick++;
    }
            //contadorComparacaoQuick++;
        if(v[esq] != v[j])
        {
            //printf("\nv[esq] = v[j]: %d", v[j]);
            v[esq] = v[j];
            v[j] = pivo;
            //printf("\nv[j] = pivo:%d", pivo);
            contadorTrocaQuick++;
            //printf("\ntrocas: %d\n", contadorTrocaQuick);
            return j;
```

```
            return j;
        }else
        {
            v[esq] = v[j];
            v[j] = pivo;
            //contadorTrocaQuick++;
            return j;
        }
    }
```

```
void QuickSort(int v[], int esq, int dir)
{
    int pivo;
    //printf("\no i(esq): %d eh menor que j(dir): %d?\n", esq, dir);

    if (dir > esq)
    {
        //printf("\nentao faca:\n");
        pivo = particao(v, esq, dir);
        QuickSort(v,esq,pivo-1); //metade esquerda
        QuickSort(v,pivo+1,dir); //metade direita
    }else
    {
        //printf("\nnao eh maior");
    }
}
```

Heap Sort

O método HeapSort é um algoritmo de ordenação bem generalista, desenvolvido em 1964, por Robert W. Floyd e J.W. Williams. O Heap Sort ultiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os mesmos são inseridos. Desse modo, ao final das inserções, os elementos do vetor podem ser sucessivamente removidos da raiz do heap. O heap pode ser apresentado como uma árvore binária .

```c
void  refazHeap(int esq, int dir, int v[])
{
    int i = esq;
    //printf("\ni = esq == %d", i);
    int j, aux;
    j = (i+1) * 2 - 1;
    //printf("\nj= (%d + 1) * 2 - 1", i);
    //printf("\nj=%d", j);
    aux = v[i];
    //printf("\naux = v[i] == %d\n", v[i]);

    while(j <= dir)
    {
        //printf("\nenquanto j==%d for <= dir==%d\n",j, dir );

    if(j < dir)
    {
        //printf("\nj==%d eh menor que dir==%d?", j,dir);
        //printf("\nv[j]:%d eh menor que v[j+1]:%d?\n", v[j], v[j+1]);

    if(v[j] < v[j+1])
    {
        j++;
        //printf("j = %d", j);
        contadorComparacaoHeap++;
        //printf("\neh menor e comparou: %d", contadorComparacaoHeap);
    }else
    {
        contadorComparacaoHeap++;
        //printf("\nnao eh menor e comparou: %d", contadorComparacaoHeap);
    }
        //printf("\nsimm");
    }

        //printf("\naux == %d eh maior ou igual a v[j]: %d? se for eh break\n", aux, v[j]);
    if(aux >= v[j])
    {
        contadorComparacaoHeap++;
        //printf("\neh maior e comparou: %d", contadorComparacaoHeap);
        break;

    }else
    {
        contadorComparacaoHeap++;
        //printf("\nnao eh maior e comparou %d", contadorComparacaoHeap);
        v[i] = v[j];
        //contadorTrocaHeap++;
        //printf("\ntroca feita: %d", contadorTrocaHeap);
        //printf("\nv[i] = v[j]: %d", v[j]);
        i = j;
        //printf("\ni = j:%d",j);
        j = i * 2 +1;
        //printf("\nj=  %d  * 2 + 1", i);
        //contadorTrocaHeap++;
    }
        //printf("\nparouu");
    }
    if(v[i] != aux)
    {
        v[i] = aux;
        contadorTrocaHeap++;
        //printf("\nv[i] = aux:%d\n", aux);
        //printf("\ntroca feita: %d", contadorTrocaHeap);
    }else
    {
        v[i] = aux;
        //printf("\nv[i] = aux:%d\n", aux);
    }
}
```

```c
//Programa para construir o heap:
void constroiHeap(int v[], int n)
{
    int esq;
    esq = n / 2;

    while (esq > 0)
    {
        esq--;
        refazHeap(esq, n, v);
    }
}


//função que implementa o Heapsort
void heapSort(int v[], int n)
{
    //printf("\n---funcao heapsort ---\n");
    int esq, dir;
    int x;

    constroiHeap(v, n-1); //constroi o heap
    esq = 0;
    dir = n - 1;

    while (dir > 0)  // ordena o vetor
    {
        contadorTrocaHeap++;
        // printf("\n----------\n");
        x = v[0];
        //printf("\nx = v[0]:%d", v[0]);
        v[0] = v[dir];
        //printf("\nv[0] = v[dir]:%d", v[dir]);
        v[dir] = x;
        //printf("\nv[dir] = x %d\n", x);
        //printf("\ntroca feita: %d", contadorTrocaHeap);
        dir--;
        refazHeap(esq, dir, v);
    }
}
```

Merge Sort

Este algoritmo se baseia na reordenação de uma estrutura linear por meio de quebras e união de todos os elementos existentes. De forma recursiva, a estrutura será reordenada, subdividida em estruturas menores até que seja feita nenhuma

troca. Assim, os elementos serão organizados de modo que cada estrutura estabelecida pelo algoritmo estará ordenada.

```
473  void merge(int vetor[], int comeco, int meio, int fim)
474  {
475      int com1 = comeco, com2 = meio+1, comAux = 0, tam = fim-comeco+1;
476      int *vetAux;
477
478      vetAux = (int*)malloc(tam * sizeof(int));
479      //printf("\ncom1 = comeco: %d", comeco);
480      //printf("\ncom2 = meio+1: %d", meio);
481      //printf("\ncomAux = 0");
482      //printf("\ntam = fim:%d-comeco:%d+1 == %d\n",fim, comeco, fim-comeco+1);
483      while(com1 <= meio && com2 <= fim)
484      {
485          //printf("\nenquanto com1: %d <= meio: %d e com2: %d <= fim:%d, faca:",com1, meio, com2, fim);
486          //printf("\nvetor[com1:%d]: %d, eh menor que vetor[com2:%d]:%d?",com1, vetor[com1], com2, vetor[com2]);
487          if(vetor[com1] < vetor[com2])
488          {
489              //printf("\neh menor:");
490              vetAux[comAux] = vetor[com1];
491              //printf("\nvetAux[comAux:%d]:%d = vetor[com1:%d]", comAux, vetAux[comAux], com1, vetor[com1]);
492              com1++;
493              //printf("\ncom1 = %d", com1);
494          }else
495          {
496              //printf("\nnao eh menor:");
497              vetAux[comAux] = vetor[com2];
498              //printf("\nvetAux[comAux:%d]:%d = vetor[com2:%d]:%d",comAux, vetAux[comAux], com2, vetor[com2]);
499              com2++;
500              //printf("\ncom2 = %d ", com2);
501              contadorTrocaMerge++;
502              //printf("\ncontador de trocas: %d",contadorTrocaMerge);
503              }
504              contadorComparacaoMerge++;
505              //printf("\ncomparou: %d", contadorComparacaoMerge);
506              comAux++;
507              //printf("\ncontador comAux = %d\n", comAux);
508      }
509
510      //printf("\ntem elementos na primeira metade? enquanto tiver faca enquanto com1:%d <= meio:%d", com1, meio);
511      while(com1 <= meio)  //Caso ainda haja elementos na primeira metade
512      {
513          //printf("\nsim");
514          vetAux[comAux] = vetor[com1];
515          //printf("\nvetAux[comAux:%d]:%d = vetor[com1:%d]:%d", comAux, vetor[comAux], com1, vetor[com1]);
516          comAux++;
517          //printf("\n comAux = %d", comAux);
518          com1++;
519          //printf("\ncom1 = %d", com1);
520      }
521      //printf("\ntem elementos na segunda metade? enquanto tiver faca enquanto com2:%d <= fim:%d", com2, fim);
522
523      while(com2 <= fim)  //Caso ainda haja elementos na segunda metade
524      {
525          //printf("\nsimm");
526          vetAux[comAux] = vetor[com2];
527          //printf("\nvetAux[comAux:%d]:%d = vetor[com2:%d]:d", comAux, vetor[comAux], com2, vetor[com2]);
528          comAux++;
529          //printf("\n comAux = %d", comAux);
530          com2++;
531          //printf("\ncom2 = %d", com2);
532      }
533      //printf("\nmovendo os elementos de volta para o vetor original:");
534      for(comAux = comeco; comAux <= fim; comAux++)  //Move os elementos de volta para o vetor original
535      {
536          vetor[comAux] = vetAux[comAux-comeco];
537          //printf("\nvetor[comAux%d]:%d = vetAux[comAux:%d - comec0:%d]:%d", comAux, vetor[comAux], comAux, comeco, vetAux[comAux
538      }
539      free(vetAux);
540      //printf("\nliberou a memoria do vetAux");
```

```c
void MergeSort(int vetor[], int comeco, int fim)
{
    if (comeco < fim)
    {
        int meio = (fim+comeco)/2;
        MergeSort(vetor, comeco, meio);
        MergeSort(vetor, meio+1, fim);
        merge(vetor, comeco, meio, fim);
    }
}
```

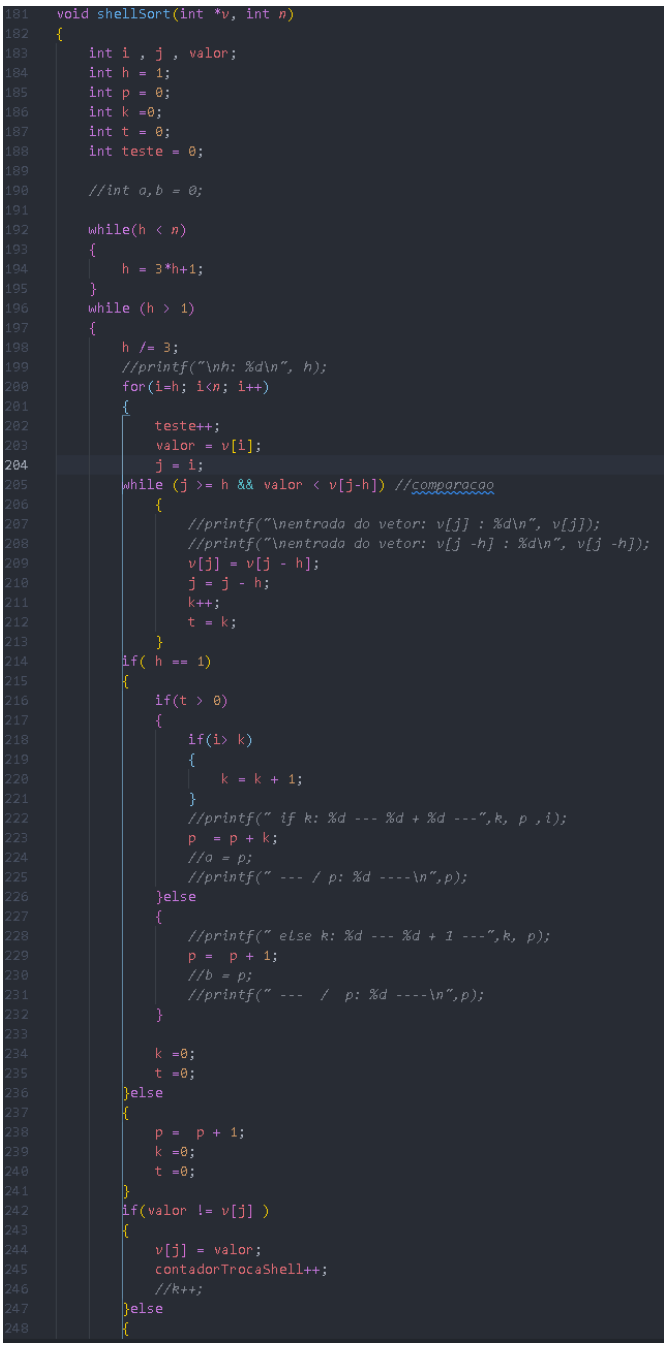
Bubble Sort

O Bubble Sort, ordenação por flutuação, ou por bolha, é um dos mais simples algoritmos de ordenação. A ideia é percorrer o vetor várias vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Esta movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, daí vem o nome do algoritmo.

```c
void bubbleSort(int v[], int n)
{
    int i, j,contTrocas;
    int x =0;
    for (i=0; i<n-1; i++)
    {
        contTrocas = 0;
        for (j=0; j<n-1; j++)
        {
            //printf("\no numero v[j] = %d eh maior que v[j + 1] = %d?\n", v[j], v[j+ 1]);
            if (v[j] > v[j+1])  // comparacao
            {
                //printf("\nentao faca:\n");
                //printf("\n x = v[j] / %d = %d\n", x, v[j]);
                x = v[j];
                //printf("\n v[j] = v[j + 1]/ %d = %d\n", v[j], v[j+1]);
                v[j] = v[j+1];
                //printf("\n v[j + 1] = x / %d = %d\n", v[j+1], x);
                v[j+1] = x;

                contadorComparacaoBubble++;
                contadorTrocaBubble++;
                contTrocas++;
                //printf("\nconttrocas com i = %d : %d", i, contadorTrocaBubble);
                //printf("\ncomparacao com i = %d : %d", i, contadorComparacaoBubble);
                //trocas
            }else
            {
                //printf("\no numero %d nao eh maior que %d\n", v[j], v[j+1]);
                contadorComparacaoBubble++;
                //printf("\ncomparacoes: %d\n", contadorComparacaoBubble);
            }
        }
                //contadorComparacaoBubble++;
                if(contTrocas == 0)
                {
                //contadorComparacaoBubble;
                    break;
                }

                //contadorComparacaoBubble++;
    }
}
```

Shell Sort

O Shell Sort é o mais eficiente algoritmo dentre os de complexidade quadrática, criado por Donald Shell, basicamente o algoritmo passa diversas vezes pela lista dividindo o grupo maior em grupos menores, e nesses menores é aplicado o método da inserção. Dessa forma, dividido, a ordenação fica mais ágil que alguns outros métodos.



```c
void shellSort(int *v, int n)
{
    int i , j , valor;
    int h = 1;
    int p = 0;
    int k =0;
    int t = 0;
    int teste = 0;

    //int a,b = 0;

    while(h < n)
    {
        h = 3*h+1;
    }
    while (h > 1)
    {
        h /= 3;
        //printf("\nh: %d\n", h);
        for(i=h; i<n; i++)
        {
            teste++;
            valor = v[i];
            j = i;
        while (j >= h && valor < v[j-h]) //comparacao
            {
                //printf("\nentrada do vetor: v[j] : %d\n", v[j]);
                //printf("\nentrada do vetor: v[j -h] : %d\n", v[j -h]);
                v[j] = v[j - h];
                j = j - h;
                k++;
                t = k;
            }
        if( h == 1)
        {
            if(t > 0)
            {
                if(i> k)
                {
                    k = k + 1;
                }
                //printf(" if k: %d --- %d + %d ---",k, p ,i);
                p  = p + k;
                //a = p;
                //printf(" --- / p: %d ----\n",p);
            }else
            {
                //printf(" else k: %d --- %d + 1 ---",k, p);
                p =  p + 1;
                //b = p;
                //printf(" ---  /  p: %d ----\n",p);
            }

            k =0;
            t =0;
        }else
        {
            p =  p + 1;
            k =0;
            t =0;
        }
        if(valor != v[j] )
        {
            v[j] = valor;
            contadorTrocaShell++;
            //k++;
        }else
        {
```

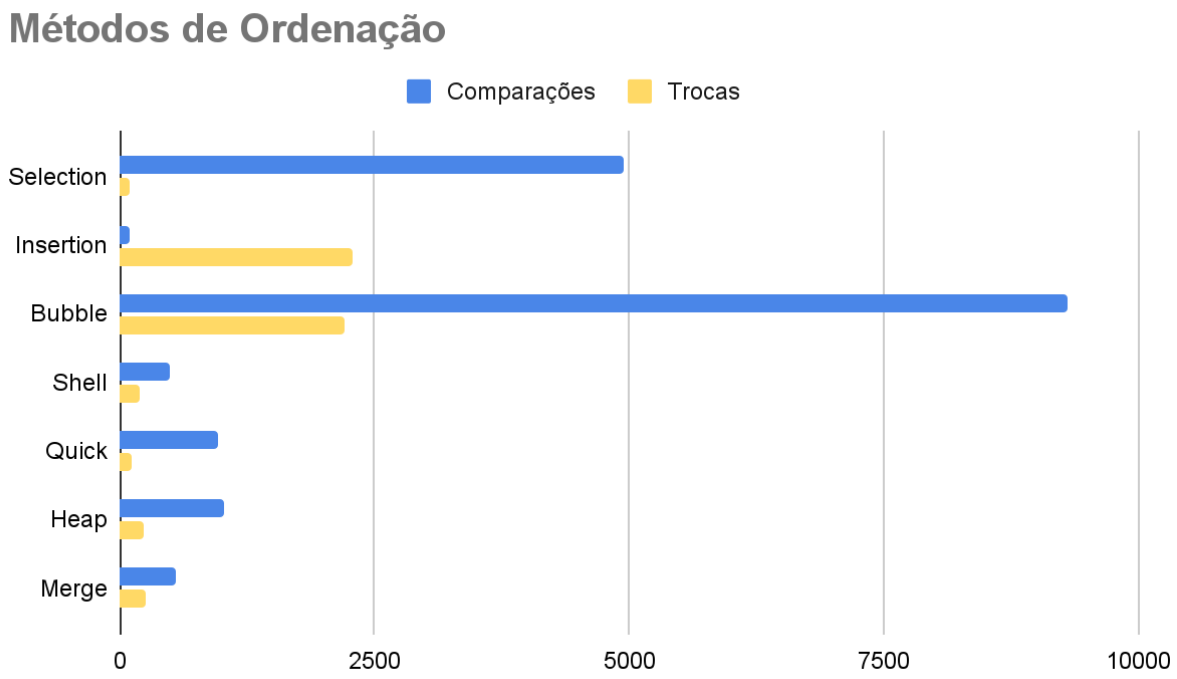
Tabelas Comparativas

NÚMERO DE COMPARAÇÕES				
MÉTODO UTILIZADO	**TAMANHO DO VETOR**			
	100	**1000**	**10000**	**100000**
SelectionSort	4950	499500	49995000	704982704
InsertionSort	90	990	9986	99986
BubbleSort	9306	907092	99160083	1342366084
ShellSort	494	6875	89165	1094381
QuickSort	969	15465	207952	2729532
HeapSort	1033	16898	235410	3019544
MergeSort	541	8719	120479	1536457

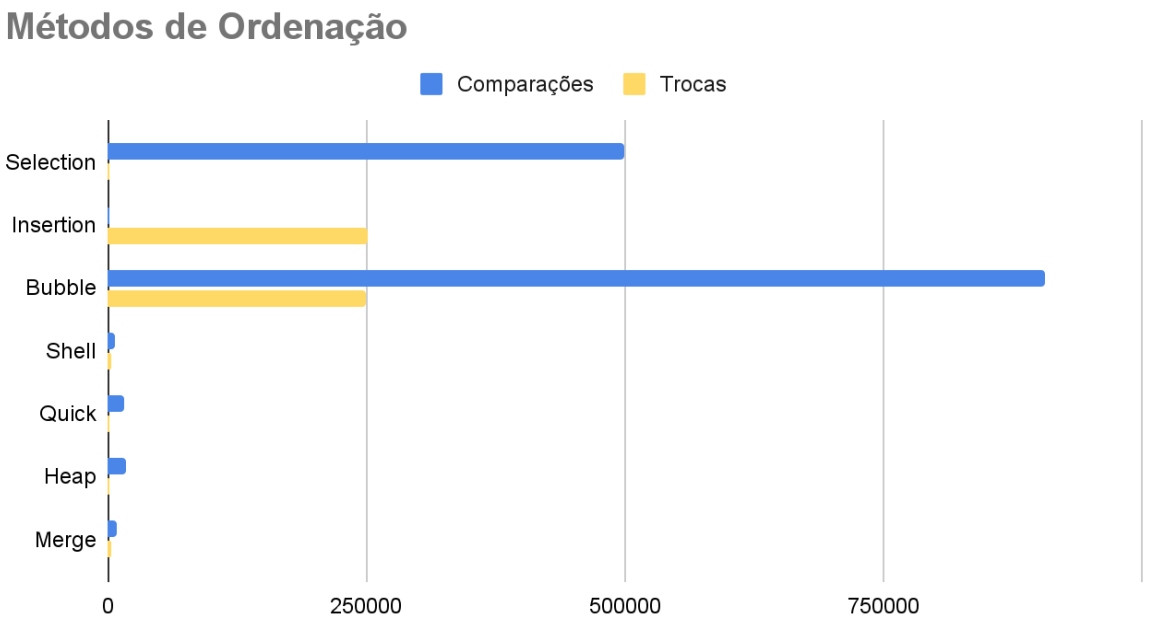
NÚMERO DE TROCAS				
MÉTODO UTILIZADO	**TAMANHO DO VETOR**			
	100	**1000**	**10000**	**100000**
SelectionSort	93	991	9986	99985
InsertionSort	2294	251517	24717787	2499283575
BubbleSort	2197	250523	24707797	2506616052
ShellSort	192	3232	48581	646458
QuickSort	116	1966	27648	342583
HeapSort	235	2383	23958	239263
MergeSort	257	4337	59070	760020

Gráficos

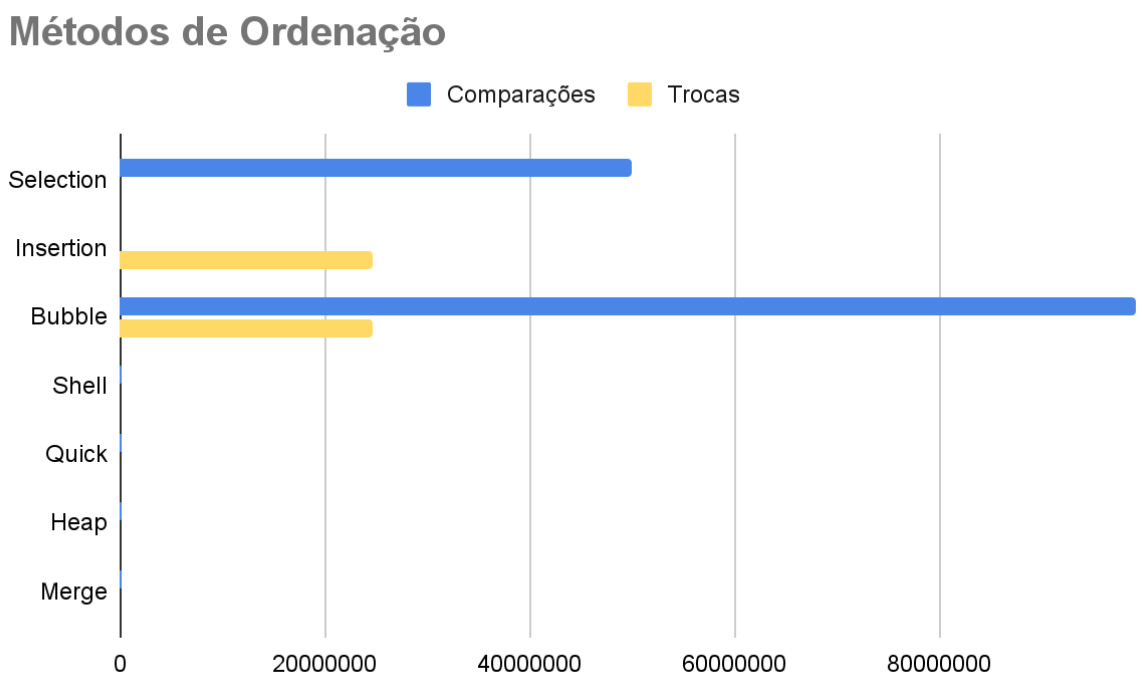
Vetor com 100 elementos



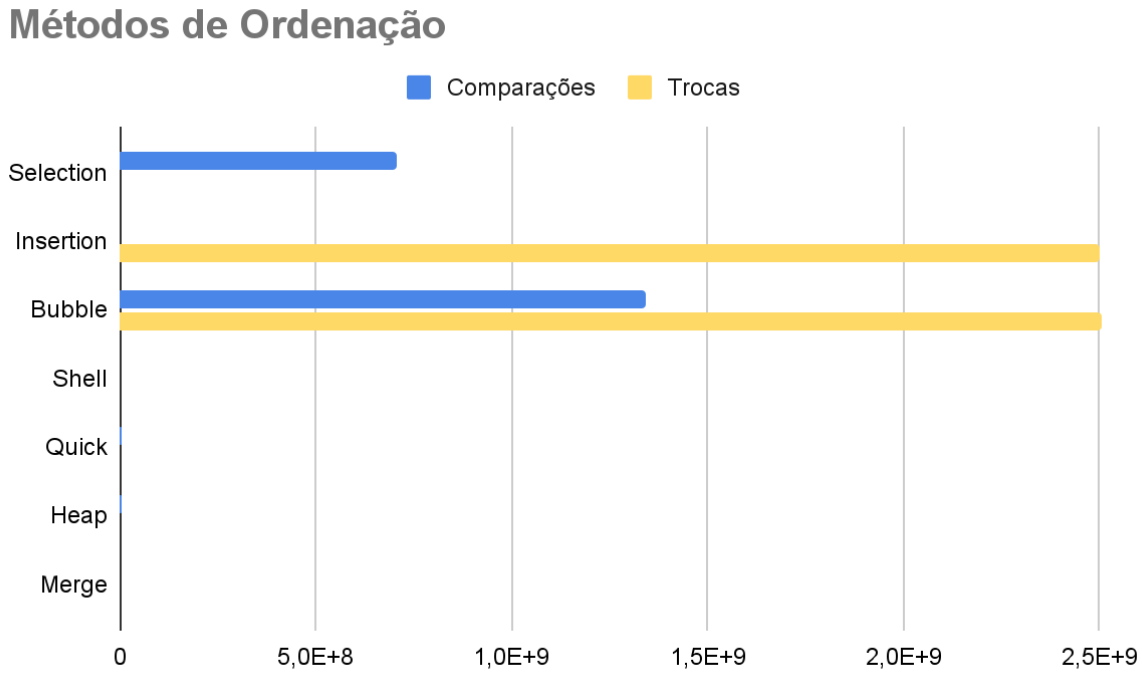
Vetor com 1000 elementos



Vetor com 10000 elementos



Vetor com 100000 elementos



Conclusão

A partir dos dados obtidos durante os testes feitos nos métodos de ordenação, foi possível analisar e avaliar a eficiência de cada algoritmo. Para os testes, foram executados 4 vetores de tamanho 100, 1000, 10000 e 100000 com números randômicos. Apesar de possíveis mudanças em números de comparações e trocas, é fato observar que existe uma média entre cada método estabelecido, sendo assim, conseguimos obter dados que irão facilitar todo o entendimento da coleta de dados e por fim uma breve análise e conclusão dos algoritmos.

Diante dos resultados apresentados nos testes de ordenação, concluímos que:

1) Nos testes do Selection Sort, podemos observar que a quantidade de comparações se mantém iguais, independente da sequência gerada para cada vetor, só há alterações quando muda o tamanho do vetor. Além disso, em vetores maiores sua eficiência é reduzida, sendo assim, consome mais tempo de leitura.

2) O Bubble Sort, mesmo em vetores de menor tamanho, é um método bem mais lento comparado aos outros, nos resultados dos testes, podemos observar que ele não apresenta bons números, referente a seu contador de comparações e trocas.

3) O Shell Sort, é o que apresenta mais eficiência, comparações e trocas se mantém de forma equilibrada para qualquer tamanho de vetor, com isso, seus resultados são satisfatórios.

4) Embora o Selection Sort seja um método não tão eficiente, notamos que em vetores maiores, ele apresentou um baixo número de trocas em relação aos métodos mais rápidos.

5) O Insertion Sort também não é um método tão eficiente, porém em vetores maiores, ele apresentou um baixo número de comparações em relação aos métodos mais rápidos.

6) O Merge Sort é um algoritmo de ordenação mediano, e como a sua estrutura se forma a partir de sua recursividade, em um vetor pequeno ele gera um número elevado de trocas e comparações.

7) O Quick Sort e o Heap Sort tem números semelhantes, quanto a contagem de comparações.

Referências

Selection sort. Wikipédia. Disponível em: https://pt.wikipedia.org/wiki/Selection_sort. Acesso em: 14 de novembro de 2021.

Insertion sort. Wikipédia. Disponível em:https://pt.wikipedia.org/wiki/Insertion_sort. Acesso em: 14 de novembro de 2021.

Bubble sort. Wikipédia. Disponível em:https://pt.wikipedia.org/wiki/Bubble_sort. Acesso em: 13 de novembro de 2021.

Heap sort. Wikipédia. Disponível em:https://pt.wikipedia.org/wiki/Heap_sort. Acesso em: 13 de novembro de 2021.

Merge sort. Wikipédia. Disponível em:https://pt.wikipedia.org/wiki/Merge_sort. Acesso em: 12 de novembro de 2021.

Quick sort. Wikipédia. Disponível em:https://pt.wikipedia.org/wiki/Quick_sort. Acesso em: 14 de novembro de 2021.

Shell sort. Wikipédia. Disponível em:https://pt.wikipedia.org/wiki/Shell_sort. Acesso em: 13 de novembro de 2021.