# ANÁLISE DE ALGORITMOS
# PARTE 02

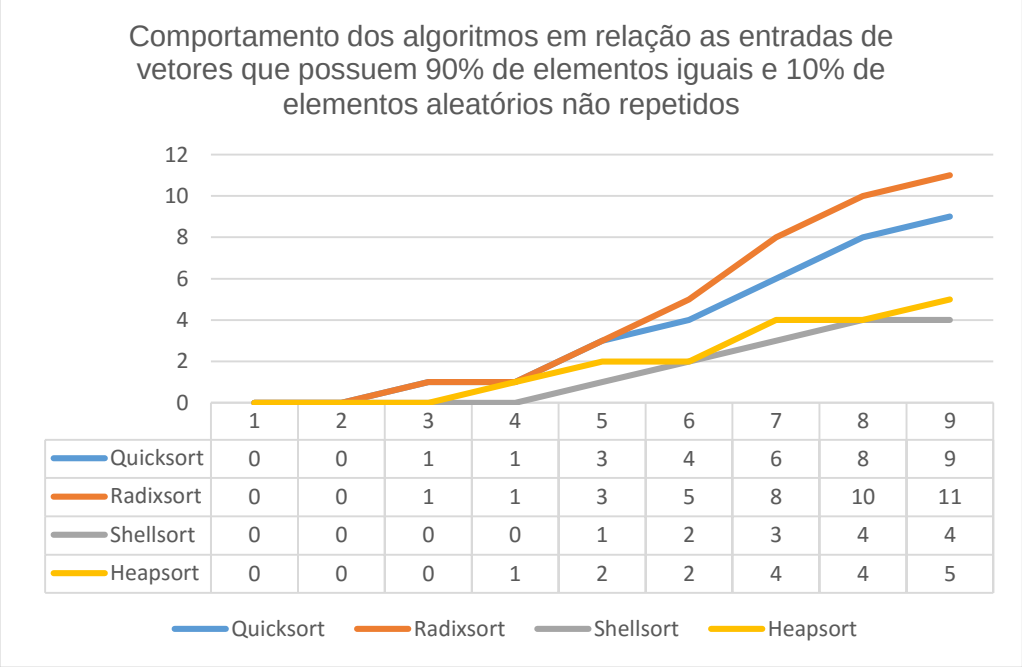
**Darcinel Júnior**
**Romilso Silva**

1. **Tabela com as entradas utilizadas e seus respectivos números identificadores nos gráficos de comportamento dos algoritmos.**

| Número Identificador | Tamanho da entrada |
|---|---|
| 1 | 10 |
| 2 | 100 |
| 3 | 1000 |
| 4 | 5.000 |
| 5 | 10.000 |
| 6 | 15.000 |
| 7 | 20.000 |
| 8 | 25.000 |
| 9 | 30.000 |

2. ***Letra a) da 5° questão relacionada a 2° parte do trabalho prático***

**2.1 Gráfico com as curvas de tempo de execução**




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Quicksort | 0 | 0 | 1 | 1 | 3 | 4 | 6 | 8 | 9 |
| Radixsort | 0 | 0 | 1 | 1 | 3 | 5 | 8 | 10 | 11 |
| Shellsort | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 4 |
| Heapsort | 0 | 0 | 0 | 1 | 2 | 2 | 4 | 4 | 5 |

### *2.2* Conclusão preliminar em relação a *Letra a) da 5° questão relacionada a 2° parte do trabalho prático*

Quando lidamos com um vetor que possui 90% de elementos iguais e 10% de números aleatórios distintos, todos os algorimos foram extremamente rápidos e apresentaram tempos iguais em relação a ordenar o vetor tanto com 10 elementos, como também com 100.

Com uma entrada de 1000 elementos, a mudança de apenas 1 segundo ocorre com os algoritmos quicksort e radixsort, enquanto os outros dois algoritmos, shellsort e heapsort permanecem com o mesmo valor da primeira análise.

Já quando a entrada aumenta para 5.000 elementos a unica mudança em relação a etapa anterior é que o heapsort iguala em 1 segundo os valores de quicksort e radixsort, enquanto o shellsort permance com o mesmo tempo de 0 segundos.

Já quando a entrada aumenta para 10.00 elementos o shellsorte se mostrou o mais eficiente,  o heapsort foi o que obteve um tempo intemediário em relação aos outro algorítmos, e os que levaram mais tempo para ordenar o vetor em relação aos outros algoritmos foram o quicksort e radixsort.

Quando a entrada aumenta para 15.00 é possível notar uma clara separação em relação ao tempo necessário para ordenar o algorítmo. Tanto o quicksort e radixsort ordenaram o vetor com o dobro do tempo e mais 1 segundo no caso do radixsort em relação ao shellsort e heapsort, que agora possuem tempo iguais e aqui são os mais eficientes.

Quando a entrada aumenta para 20.00, o shelssort passa a ser o mais eficiente devido ao aumento de 1 segundo em relação ao tempo de execução do heapsort e também pelo fato do shelssort ter mantido o tempo em comparação a análise anterior. O radixsort se mostra o algoritmo que demanda mais tempo, aumentado a sua diferenca de tempo para com o quicksort de 1 para 2 segundos.

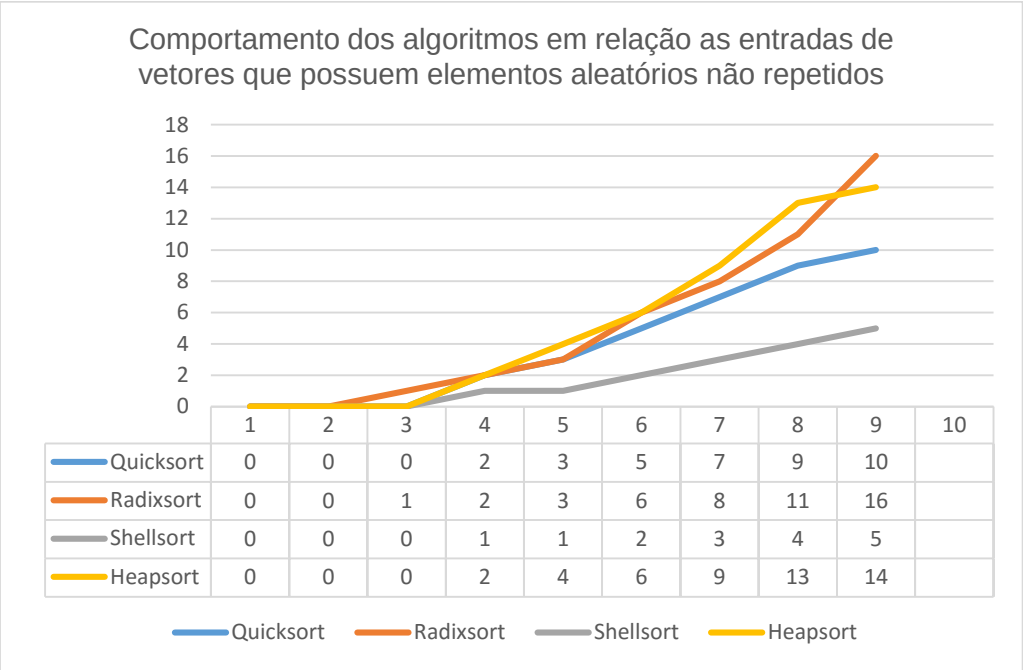
Quando a entrada aumenta para 25.000, tanto o quicksort e radixsort ordenaram o vetor com o dobro do tempo e mais 2 segundos no caso do radixsort em relação ao shellsort e heapsort, que diferente da etapa anterior, agora novamente possuem tempos iguais e ambos são os mais eficientes.

Quando a entrada aumenta para 30.00 tanto o quicksort e radixsort ordenaram o vetor com mais que o dobro do tempo em relação ao shellsort, que agora é o mais eficiente devido ao aumento de 1 segundo do heapsort.

É possível concluir nesta primeira fase que todos os algoritmos possuem um excelente comportamento para vetores de até 100 elementos. E a medida que os elementos crescem tanto o quicksort como o radixsort demandam muito mais tempo para realizar a ordenação dos vetores, onde o radixsort é o que demanda mais tempo, seguido de perto pelo quicksort. Já  o shellsort e heapsort se mostram muito mais eficientes que os outros dois algoritmos, onde o mais eficiente é o shellsort devido a ser o mais eficiente em números de entradas destintas, seguido de perto pelo quicksort que quando não possui o mesmo tempo de execução do shellsort, obtem diferença mínima de 1 segundo.

## 3 Letra b) da 5° questão relacionada a 2° parte do trabalho prático

### 3.1 Gráficos com as curvas de tempo de execução




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Quicksort | 0 | 0 | 0 | 2 | 3 | 5 | 7 | 9 | 10 | |
| Radixsort | 0 | 0 | 1 | 2 | 3 | 6 | 8 | 11 | 16 | |
| Shellsort | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 4 | 5 | |
| Heapsort | 0 | 0 | 0 | 2 | 4 | 6 | 9 | 13 | 14 | |

### 3.2 Conclusão preliminar em relação a *Letra b) da 5° questão relacionada a 2° parte do trabalho prático*

Quando lidamos com um vetor que possui com elementos aleatórios distintos, assim como os testes realizados na primeira questão, todos os algorimos foram extremamente rápidos e apresentaram tempos iguais em relação a ordenar o vetor tanto com 10 elementos, como também com 100 elementos.

Com uma entrada de 1000 elementos, a mudança de apenas 1 segundo ocorre apenas com o algoritmo radixsort, enquanto os outros 3 algoritmos permanecem com o mesmo valor da primeira análise.

Já quando a entrada aumenta para 5.00 elementos, os que tiveram crescimento de tempo menor foi o tanto o radixsort como o shellsort, crescimento de 1 segundo. Já os demais, quicksort e heapsort obtiveram crecimento de tempo de 2 segundos, onde o mais eficiente nesta etapa é o shellsort.

Já quando a entrada aumenta para 10.00 elementos o shellsort se mostra o mais eficiente, com uma diferença de 3 segundos para o menos eficiente, e com uma diferença de 2 segundos para os demais algoritmos.

Quando a entrada aumenta para 15.00 é possível notar uma clara diferença em relação ao tempo necessário para ordenar o vetor aleatório utilizando o shellsort para como os outros algoritmos, o shellsort se mostra o mais eficiente, com uma diferença de 4 segundos para os menos eficientes que são o radixsort e heapsort, e com uma diferença de 3 segundos para o quicksort.

Quando a entrada aumenta para 20.000 o shellsort se mostra o mais eficiente, com uma diferença de 6 segundos para o menos eficiente que é o heapsort.

Quando a entrada aumenta para 25.000, o shellsort se mostra o mais eficiente, com uma diferença de 9 segundos para o menos eficiente que é o radixsortsort. No entanto

é importante resaltar o aumento de diferença de tempo entre o quicksort para com o menos eficiente, que na etapa de análise anterior possuia uma diferenca de apenas 2 segundos, e agora cresce para 4 segundos para com o heapsort.

Quando a entrada aumenta para 30.000, o shellsort que ainda se mostra o mais eficiente, possui uma diferença maior que 3x para com o menos eficiente que é o radixsort, com uma diferença de 11 segundos. O algoritmo que se distancia do menos eficiente e que tem a tendência de aumentar a sua eficiencia a medida que a entrada cresce é o algoritmo quicksort. O aumento de diferença de tempo entre o quicksort para com o menos eficiente, que na etapa de análise anterior possuia uma diferença de 4 segundos para o heapsort que era o menos eficiente, agora cresce para 6 segundos para com o radixsort que agora é o menos eficiente.

É possível concluir nesta segunda fase que todos os algoritmos possuem um excelente comportamento para vetores de até 100 elementos. E a medida que os elementos crescem tanto o radixsort como o heapsort demandam muito mais tempo para realizar a ordenação dos vetores, onde o radixsort é o que demanda mais tempo, seguido de perto pelo heapsort. É importante evidenciar o aumento da diferença de tempo entre o quicksort para com os algoritmos menos eficientes nas 3 ultimas etapas, onde na etapa de análise relacionada a um vetor de 20.000 elementos possuia uma diferença de 2 segundos para com o heapsort que era o menos eficiente, posteriormente na etapa de análise relacionada a um vetor de 25.000 elementos aumentou a sua diferença de 2 para 4 segundos para o com o heapsort que era o menos eficiente, e na ultima entrada com o vetor de 30.000 elementos cresce para 6 segundos para com o radixsort que se tornou o menos eficiente. Isso implica dizer que essa diferença tende aumentar a media que o vetor cresce, e dessa forma o o quicksort tende a melhorar o seu comportamento para vetores superiores a 30.00 elemento, podendo até a disputar o posto de mais eficiente com o shellsort. Já o shellsort se mostra muito mais eficiente que os outros algoritmos com uma grande diferença que cresce a medida que o numero de elementos do vetor também cresce. Outro fator relevante é o aumento drastico de tempo em relação ao comportamento do radixsort, no qual o seu tempo dobra após o intervalo de uma entrada, possuindo um tempo de 8 segundos para um vetor de 20.000 elementos, e posteriormente um tempo de 16 segundos para um vetor de 30.000 elementos. Isso implica dizer que essa diferença tende aumentar a media que o vetor cresce, ou seja, o radixsort tende a piorar o seu comportamento para vetores superiores a 30.000 elementos.

## 4 Indicação do melhor algoritmo para vetores com diferentes tamanhos de entrada, o melhor algoritmo para vetores com 90 % de elementos iguais e 10% de elementos aleatórios distintos, e o melhor algoritmo para vetores com elementos aleatórios.

Com base na análise dos algoritmos e dos resultados dos testes, todos os algoritmos analisados foram eficientes para tamanhos de entrada pequenas de 10 até 100 elementos, entretanto, quando o número de elementos do vetor cresce e vai até 30.00 elementos, o shellsort se mostrou o mais eficiente nas duas situações apresentadas, ou seja, é o melhor algoritmo para vetores com 90% de elementos iguais e 10% de elementos aleatórios distintos, e é o melhor algoritmo para vetores com elementos aleatórios. E também é o mais eficiente para vetores com diferentes tamanhos de entrada nos 2 problemas analisados, pois em todos os casos possui uma diferença de tempo de execução extremamente significativa para com os demais algoritmos de ordenação analisados.

## 5 Explicação dos algoritmos

### 5.1 Quicksort

A ideia central do algoritmo quicksort é dividir o problema de realizar a ordenação de um conjunto com um número de itens em dois problemas menores, onde estes possam ser ordenados independentemente, dessa forma os resultados são combinados para solução final seja produzida. A parte mais delicada do método é o processo de partição, o vetor A [Esq..Dir] é rearranjado por meio da escolha arbitrária de um pivô x, que é escolhido como sendo o elemento central: A[(i + j) / 2]. O vetor A é particionado em duas partes, parte esquerda (chaves ≤ x), e parte direita (chaves ≥ x). A ideia aqui é organizar o vetor de uma forma que todos os elementos anteriores ao pivô sejam menores que ele, e da mesma forma todos os elementos após o pivô sejam maiores que ele, e dessa forma ao final desse processo o pivô estará em sua posição final e haverá duas parte não ordenadas. Um pivô x é escolhido arbitrariamente, então o vetor é percorrido a partir da esquerda até que A[i] ≥ x. 3, e de forma similar, o vetor é percorrido a partir da a partir da direita até que A[j] ≤ x. 4. É feita a troca de A[i] com A[j] e este processo ocorre até que i e j se cruzem. Ao final do algoritmo de partição, o vetor A[Esq..Dir] está particionado de tal forma que os itens em A[Esq], A[Esq + 1], ..., A[j] são menores ou iguais a x, os itens em A[i], A[i + 1], ..., A[Dir] são maiores ou iguais a x.

### 5.2 Radixsort

Existem dois formatos básicos diferentes do algoritmo de ordenação radix, e a variação do algoritmo escolhida foi a forma de ordenação chamada classificação radix com menor dígito significativo (LSD), que trabalha processando o dígito menos significativo, movendo-se em direção ao dígito maior e mais significativo à medida que continua a classificar. Esse método é semelhante ao MSD, pois também usa a contagem ou o ordenamento do bucket internamente, no entanto, geralmente é aplicado de forma iterativa e não de forma recursiva. O algoritmo de ordenação radix adotado é um algoritmo de ordenação de inteiros, que ordena agrupando números por seus dígitos individuais, ou seja, ele manipula cada dígito como se fosse uma chave e implementa a contagem de ordenação para fazer o trabalho de classificação. No radixsort iniciamos primeiro a classificação dos elementos com base no último dígito, que é o dígito menos significativo, e logo em seguida o resultado é novamente classificado pelo segundo dígito, e esse processo continua para todos os dígitos até alcançarmos o dígito mais significativo.

### 5.3 Shellsort

É um refinamento do método de inserção direta, e difere deste método pelo fato de no lugar de considerar o vetor a ser ordenado não apenas por um único segmento. Quando um elemento precisa ser movido para muito longe, são necessários movimentar várias elementos no meio termo de dois elementos, ou seja, elementos adjacentes, e para que isso seja feito de forma facilitada, o shellsort permiti a troca desses itens que estão em posições distantes. O Shell faz trocas a uma certa distância que diminui a cada passada onde ele primeiro compara elementos separados por h posições e os reorganiza, e com isso h vai diminuindo a distância de comparação, e dessa forma o valor de h vai sendo reduzido até que se torne 1, e quando h = 1, o algoritmo shellsort corresponde ao algoritmo de inserção que irá ser utilizar para ordenar pequenos seguimentos. O shellsort considera vários segmentos, e dessa forma é aplicado o método de inserção direta em cada um desses segmentos, assim o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores, e nos grupos menores é aplicado o método da ordenação por inserção.

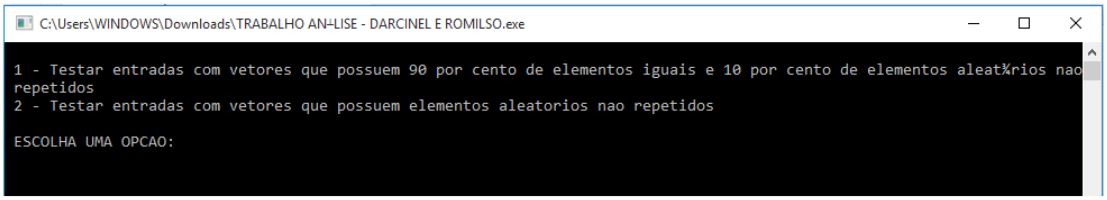
### 5.4 Heapsort

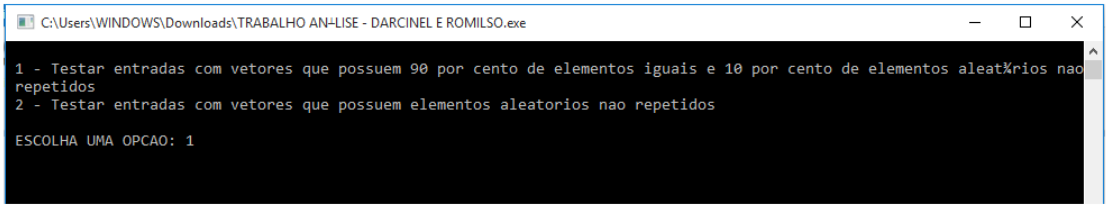
O HeapSort baseado no método de seleção direta, ordena através de sucessivas seleções do elemento correto a ser posicionado em um segmento ordenado assim sendo, o algoritmo utiliza uma estrutura de dados chamada de heap, que consiste em uma arvore que a raiz é o elemento de menor valor, caso seja um heap de mínimo, ou o de maior valor, caso seja um heap de máximo. O HeapSort utiliza um heap binário que é uma árvore binária mantida na forma de vetor para manter o próximo elemento a ser selecionado, o heap é gerado e mantido no próprio vetor a ser ordenado. Com o heap construído o HeapSort funciona da seguinte forma, ele apenas cria o heap a partir do vetor de entrada, e o algoritmo apenas vai removendo do heap e colocando no vetor de saída. O heapsort cria a heap a partir dos dados, pegamos o maior elemento da heap e colocamos na sua posição correspondente ao array, e posteriormente recontruimos a heap. Na função criaHeap, verificamos se o pai tem dois filhos, quem é o maior filho, e se o filho é maior que o pai, nesse caso o filho se torna o pai, e repetidos o processo, o antigo pai ocupa o lugar do último filho analisado.

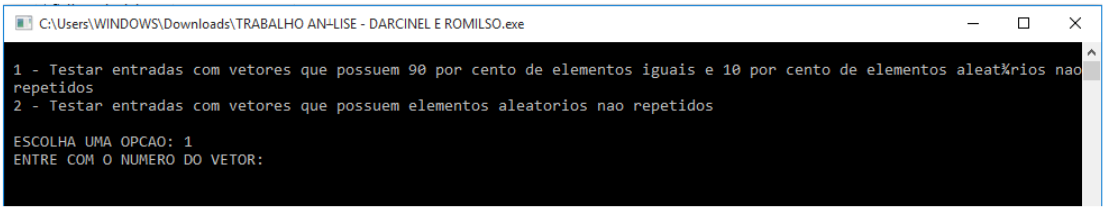
## 6 Manual mostrando como reproduzir os experimentos feitos

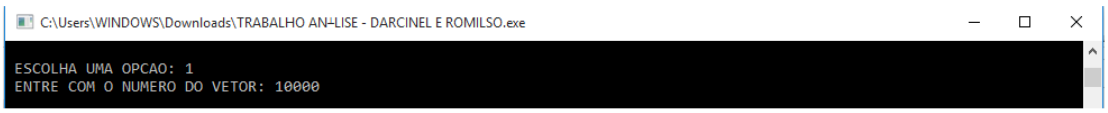
***Letra a) da 5° questão relacionada a 2° parte do trabalho prático***

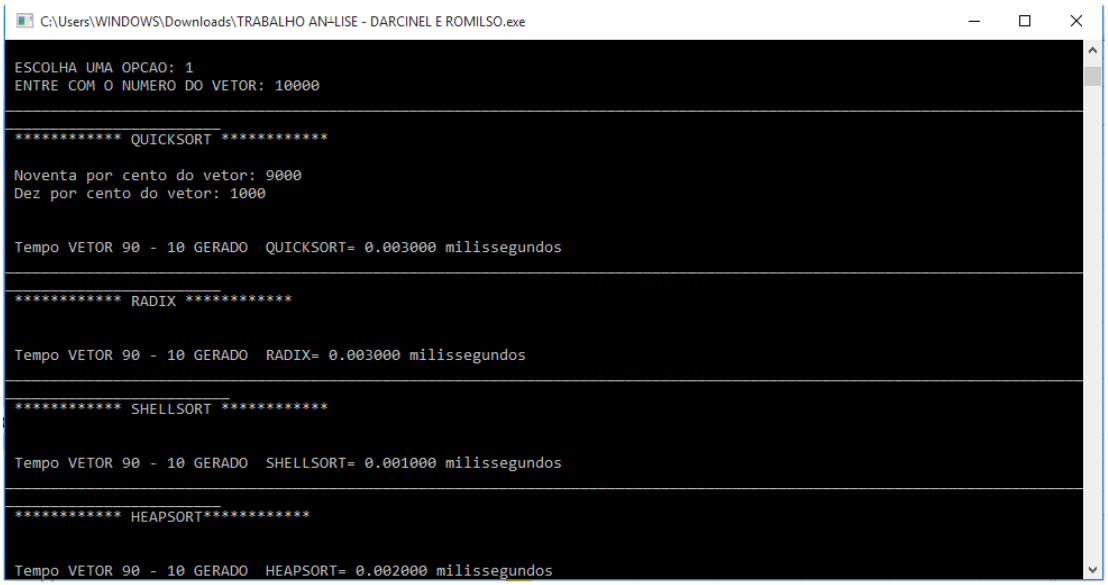
**Comportamento dos algoritmos em relação as entradas de vetores que possuem 90% de elementos iguais e 10% de elementos aleatórios não repetidos**



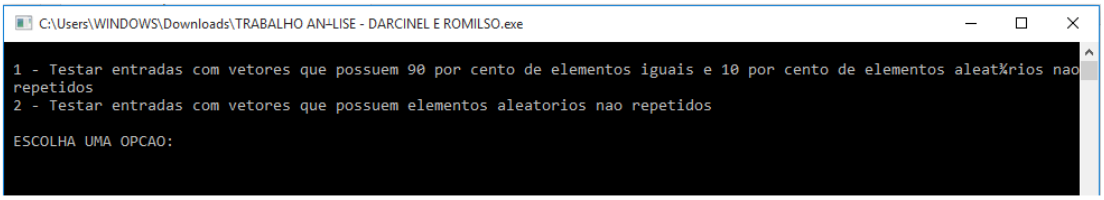



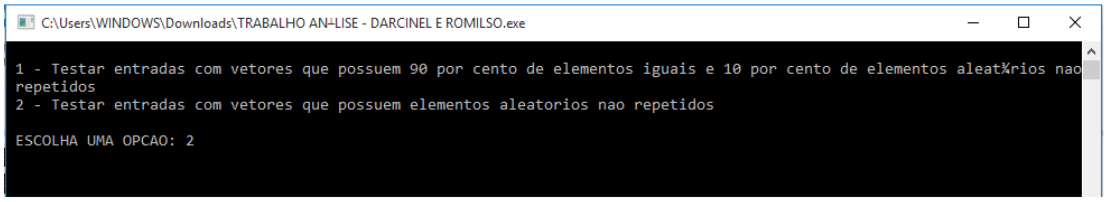



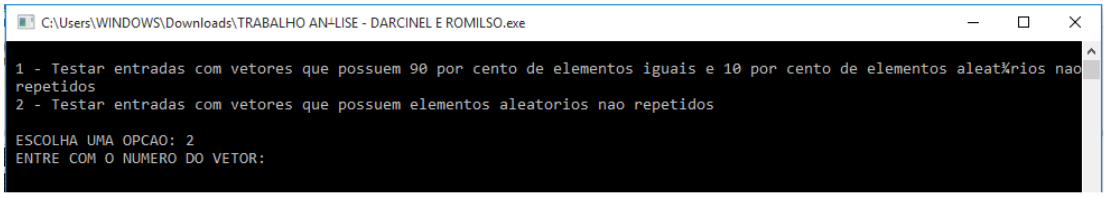



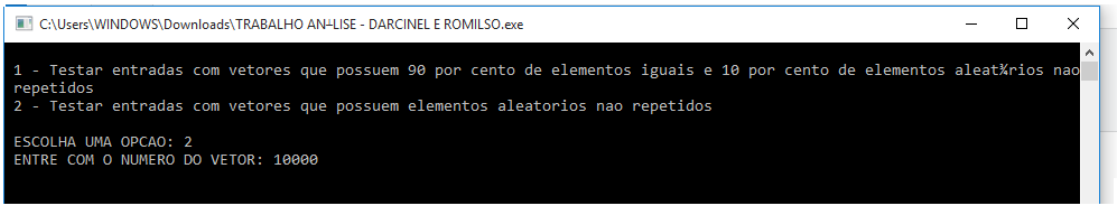

***Letra b) da 5° questão relacionada a 2° parte do trabalho prático***

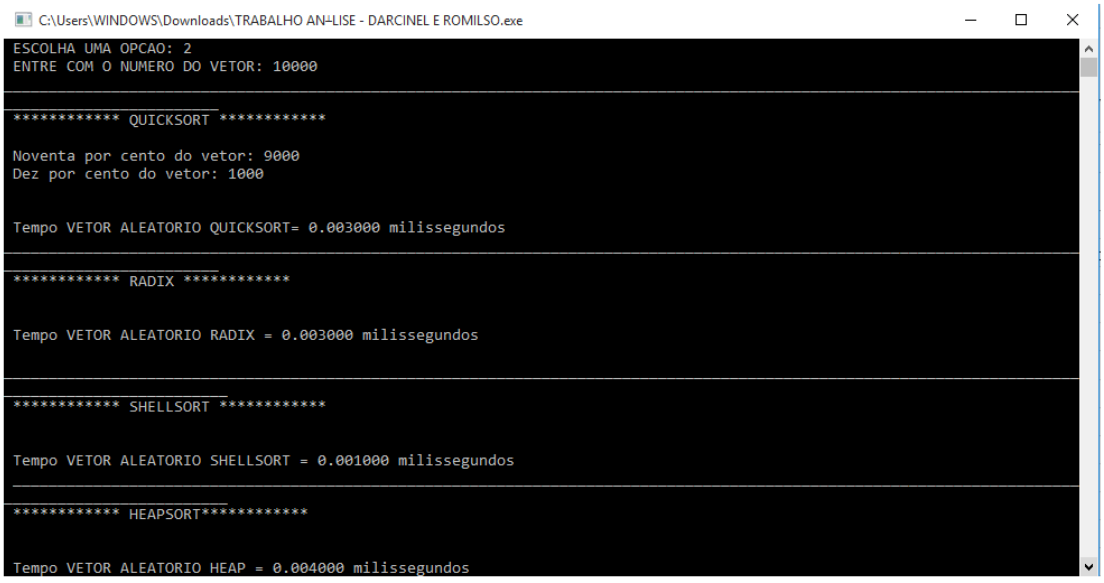
**Comportamento dos algoritmos em relação as entradas de vetores que possuem elementos aleatórios não repetidos**


C:\Users\WINDOWS\Downloads\TRABALHO AN╬LISE - DARCINEL E ROMILSO.exe
1 - Testar entradas com vetores que possuem 90 por cento de elementos iguais e 10 por cento de elementos aleat%rios nao repetidos
2 - Testar entradas com vetores que possuem elementos aleatorios nao repetidos
ESCOLHA UMA OPCAO:


C:\Users\WINDOWS\Downloads\TRABALHO AN╬LISE - DARCINEL E ROMILSO.exe
1 - Testar entradas com vetores que possuem 90 por cento de elementos iguais e 10 por cento de elementos aleat%rios nao repetidos
2 - Testar entradas com vetores que possuem elementos aleatorios nao repetidos
ESCOLHA UMA OPCAO: 2


C:\Users\WINDOWS\Downloads\TRABALHO AN╬LISE - DARCINEL E ROMILSO.exe
1 - Testar entradas com vetores que possuem 90 por cento de elementos iguais e 10 por cento de elementos aleat%rios nao repetidos
2 - Testar entradas com vetores que possuem elementos aleatorios nao repetidos
ESCOLHA UMA OPCAO: 2
ENTRE COM O NUMERO DO VETOR:


C:\Users\WINDOWS\Downloads\TRABALHO AN╬LISE - DARCINEL E ROMILSO.exe
1 - Testar entradas com vetores que possuem 90 por cento de elementos iguais e 10 por cento de elementos aleat%rios nao repetidos
2 - Testar entradas com vetores que possuem elementos aleatorios nao repetidos
ESCOLHA UMA OPCAO: 2
ENTRE COM O NUMERO DO VETOR: 10000


C:\Users\WINDOWS\Downloads\TRABALHO AN╬LISE - DARCINEL E ROMILSO.exe
ESCOLHA UMA OPCAO: 2
ENTRE COM O NUMERO DO VETOR: 10000
********** QUICKSORT **********
Noventa por cento do vetor: 9000
Dez por cento do vetor: 1000
Tempo VETOR ALEATORIO QUICKSORT= 0.003000 milissegundos
********** RADIX **********
Tempo VETOR ALEATORIO RADIX = 0.003000 milissegundos
********** SHELLSORT **********
Tempo VETOR ALEATORIO SHELLSORT = 0.001000 milissegundos
********** HEAPSORT**********
Tempo VETOR ALEATORIO HEAP = 0.004000 milissegundos

## 7 Código fonte

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include <windows.h>
#define VALIDO   1
#define INVALIDO 0

double PCFreq = 0.0;
__int64 CounterStart = 0;

void particao(int esq, int dir, int *i, int *j, int *A){
   int pivo,tmp;
   *i=esq;
   *j=dir;
   pivo=A[(*i+*j)/2];
   do{
      while(pivo>A[*i])
         (*i)++;
      while(pivo<A[*j])
         (*j)--;
      if(*i<=*j){
         tmp=A[*i];
         A[*i]=A[*j];
         A[*j]=tmp;
         (*i)++;
         (*j)--;
      }
```

```
   }while(*i<=*j);
}
void ordena(int esq, int dir, int *A){
   int i,j;
   particao(esq,dir,&i,&j,A);
   if(esq<j)
      ordena(esq,j,A);
   if(i<dir)
      ordena(i,dir,A);
}

void quicksort(int *A , int n){

   ordena(0,n-1,A);

}

void radix(int *vetor, int n) {

   int i;
   int *b;
   int maior = vetor[0];
   int exp = 1;

   b = (int *)calloc(n, sizeof(int));

   for (i = 0; i < n; i++) {
      if (vetor[i] > maior)
                                          maior = vetor[i];
   }
```

```
   while (maior/exp > 0) {
      int bucket[10] = { 0 };
                                        for (i = 0; i < n; i++)
                                           bucket[(vetor[i] / exp) % 10]++;
                                        for (i = 1; i < 10; i++)
                                           bucket[i] += bucket[i - 1];
                                        for (i = n - 1; i >= 0; i--)
                                           b[--bucket[(vetor[i] / exp) % 10]] = vetor[i];
                                        for (i = 0; i < n; i++)
                                           vetor[i] = b[i];
                                        exp *= 10;
   }

   free(b);

}

void shell(int *vetor, int n){

   int i;

   int j , aux;
   int gap = 1;

   while(gap < n) {
      gap = 3*gap+1;
   }

   while ( gap > 1) {
      gap /= 3;
```

```
      for(i = gap; i < n ; i++) {
         aux = vetor[i];
         j = i;
         while (j >= gap && aux < vetor[j - gap]) {
            vetor[j] = vetor[j - gap];
            j = j - gap;
         }
         vetor [j] = aux;
      }
   }

}

void criaHeap(int *vet, int i, int f){
   int aux = vet[i];
   int j = i * 2 + 1;
   while (j <= f){
      if(j < f){
         if(vet[j] < vet[j + 1]){
            j = j + 1;
         }
      }
      if(aux < vet[j]){
         vet[i] = vet[j];
         i = j;
         j = 2 * i + 1;
      }else{
         j = f + 1;
      }
   }
   vet[i] = aux;
```

```
}

void heap(int *vet, int N){


   int i, aux;
   for(i=(N - 1)/2; i >= 0; i--){
      criaHeap(vet, i, N-1);
   }
   for (i = N-1; i >= 1; i--){
      aux = vet[0];
      vet [0] = vet [i];
      vet [i] = aux;
      criaHeap(vet, 0, i - 1);
   }


}

void exibe(int *vet, int N){

int l;

    printf("\n");

                                          for (l = 0; l < N; l++) {
      printf(" %d", vet[l]);
   }


}
```

```
int* geravetor(int N){

                                        int i, j, status;
   int *A;

   A = (int*) malloc(sizeof(int) * N);

 srand( (unsigned)time(NULL) );

   for (i = 0; i < N; ++i) {
     do {
       A[i] = rand() % N;
       status = VALIDO;
       for (j = 0; j < i; ++j)
         if (A[i] == A[j])
           status = INVALIDO;
     } while (status == INVALIDO);

   }

   return A;
}

int* geravetorNoventa(int N){

                                        int noventa, dez;
                                        noventa = (N*90)/100;
                                        dez  = N - noventa;

                                        int nove = 0;
```

```
                                        int i, j, status;
  int *A;

  A = (int*) malloc(sizeof(int) * N);

srand( (unsigned)time(NULL) );

   nove = rand() % N;
  for (i = 0; i < noventa; i++) {
      A[i] = nove;
      status = VALIDO;
  }

  srand(time(NULL));

  for (i = noventa; i < N; ++i) {
    do {
      A[i] = rand() % dez;
      status = VALIDO;
      for (j = noventa; j < i; ++j)
        if (A[i] == A[j])
          status = INVALIDO;
    } while (status == INVALIDO);

  }

  return A;
}

int main(){
```

```
 int i, n, op;
                                clock_t begin;
                                clock_t end;
                                double time_spent;
                                int noventa, dez;

 for(;;){

                                printf("\n 1 - Testar entradas com vetores que possuem 90 por cento de elementos iguais e 10 por cento de elementos aleatórios nao repetidos ");
                                printf("\n 2 - Testar entradas com vetores que possuem elementos aleatorios nao repetidos");

                                printf(" \n\n ESCOLHA UMA OPCAO: ");
                                scanf("%d", &op);

                                switch (op){

  case 1 :
                                {

                                printf(" ENTRE COM O NUMERO DO VETOR: ");
                                scanf("%d", &n);

                                int *vetNoventa[n];

printf("_____
_____");
```

```
                                printf(" \n ********** QUICKSORT ********** \n ");



                                noventa = (n*90)/100;
                                dez  = n - noventa;

printf(" \n Noventa por cento do vetor: %d", noventa);
                                printf(" \n Dez por cento do vetor: %d \n", dez);



                                //printf(" \n VETOR 90 - 10 GERADO QUICK: ");

                                vetNoventa[n] = NULL;

                                vetNoventa[n] = geravetorNoventa(n);



// exibe(vetNoventa[n], n);

 begin = clock();

 quicksort( vetNoventa[n],  n);

                                end = clock();



// exibe(vetNoventa[n], n);

                                time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

```
   printf("\n\n Tempo VETOR 90 - 10 GERADO  QUICKSORT= %f milissegundos \n", time_spent);




printf("_____
_____");

                                    printf(" \n ********** RADIX ********** \n");

                                    //printf(" \n VETOR 90 - 10 GERADO RADIX: ");


                                    vetNoventa[n] = NULL;

                                    vetNoventa[n] = geravetorNoventa(n);



  //exibe(vetNoventa[n], n);

                                    begin = clock();

                                    radix( vetNoventa[n],  n);

                                    end = clock();
```

```
   //exibe(vetNoventa[n], n);

   time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

   printf("\n\n Tempo VETOR 90 - 10 GERADO  RADIX= %f milissegundos \n", time_spent);




printf("_____
_____");

   printf(" \n ************ SHELLSORT ************ \n ");

                                    //printf(" \n VETOR 90 - 10 GERADO SHELL ");

                                    vetNoventa[n] = 0;

   vetNoventa[n] = geravetorNoventa(n);




   //exibe(vetNoventa[n], n);

   begin = clock();

   shell( vetNoventa[n],  n);

                                    end = clock();



   //exibe(vetNoventa[n], n);
```

```
   time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

   printf("\n\n Tempo VETOR 90 - 10 GERADO  SHELLSORT= %f milissegundos \n", time_spent);



printf("_____
_____");
                                    printf(" \n ************ HEAPSORT************ \n ");

                                    //printf(" \n VETOR 90 - 10 GERADO HEAP: ");

                                    vetNoventa[n] = 0;

   vetNoventa[n] = geravetorNoventa(n);



   //exibe(vetNoventa[n], n);

   begin = clock();

   heap( vetNoventa[n],  n);

                                    end = clock();



   //exibe(vetNoventa[n], n);

   time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

```
   printf("\n\n Tempo VETOR 90 - 10 GERADO  HEAPSORT= %f milissegundos \n",time_spent
);


  break;


}


  case 2 :
  {


    for(;;){
                                        printf(" ENTRE COM O NUMERO DO VETOR: ");
                                        scanf("%d", &n);


   int *vetAleatorio[n];


printf("_____
_____");
                                        printf(" \n ************ QUICKSORT ************ \n ");


                                        noventa = (n*90)/100;
                                        dez  = n - noventa;


   printf(" \n Noventa por cento do vetor: %d", noventa);
                                        printf(" \n Dez por cento do vetor: %d \n", dez);




                                        //printf(" \n VETOR ALEATÓRIO GERADO QUICK: \n");
```

```
                                        vetAleatorio[n] = NULL;

                                        vetAleatorio[n] = geravetor(n);

   //exibe(vetAleatorio[n], n);

   begin = clock();

                                        quicksort( vetAleatorio[n],  n);

                                        end = clock();


  // exibe(vetAleatorio[n], n);

   double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;


   printf("\n\n Tempo VETOR ALEATORIO QUICKSORT= %f milissegundos \n", time_spent);


printf("_____
_____");

                                        printf(" \n ********** RADIX ********** \n");


   //printf(" \n VETOR ALEATORIO GERADO RADIX: \n ");

   vetAleatorio[n] = 0;
```

```
                                        vetAleatorio[n] = geravetor(n);



   //exibe(vetAleatorio[n], n);

                                        begin = clock();

                                        radix( vetAleatorio[n], n) ;

                                        end = clock();



   //exibe(vetAleatorio[n], n);

   time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

   printf("\n\n Tempo VETOR ALEATORIO RADIX = %f milissegundos \n\n",  time_spent);


printf("_____
_____");

   printf(" \n ************ SHELLSORT ************ \n ");



   //exibe(vetAleatorio[n], n);
```

```
                                        begin = clock();

    shell( vetAleatorio[n], n) ;

                                        end = clock();



    //exibe(vetAleatorio[n], n);

    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

    printf("\n\n Tempo VETOR ALEATORIO SHELLSORT = %f milissegundos \n ",  time_spent);


printf("_____
_____");
                                        printf(" \n ************ HEAPSORT************ \n ");


                                        //printf(" \n VETOR ALEATORIO GERADO HEAP: \n ");

                                        vetAleatorio[n] = 0;

                                        vetAleatorio[n] = geravetor(n);



    //exibe(vetAleatorio[n], n);

                                        begin = clock();
```

```
                                heap( vetAleatorio[n], n) ;

                                end = clock();



   //exibe(vetAleatorio[n], n);

   time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

   printf("\n\n Tempo VETOR ALEATORIO HEAP = %f milissegundos\n",time_spent);
  break;

}
}
}
}
}
```