UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
ALGORITMOS E ESTRUTURAS DE DADOS 1

TRABALHO PRÁTICO 1

WENDEL MARQUES DE JESUS SOUZA
201702793

GOIÂNIA
2018

WENDEL MARQUES DE JESUS SOUZA

TRABALHO PRÁTICO 1

Primeiro trabalho prático da disciplina Algoritmos e Estruturas de Dados 1, do curso de Ciência da Computação da Universidade Federal de Goiás ministrada pela professora Nádia Félix.

GOIÂNIA
2018

RESUMO

O presente trabalho foi desenvolvido para a verificação do funcionamento de alguns métodos de ordenação. Entre os métodos, estão: insertion sort, selection sort, bubble sort, quick sort, radix sort, counting sort, merge sort e bucket sort. Foram realizadas as análises assintótica (descrição geral dos métodos e verificação do desempenho de cada um deles e empírica (gráficos e análise das implementações).

SUMÁRIO

1) ANÁLISE ASSINTÓTICA

- **Insertion Sort**

Insertion Sort, ou ordenação por inserção, aplica seguinte ideia: percorre-se as posições do vetor até encontrar o menor elemento. Esse elemento obtido é adicionado na primeira posição do array. Ou seja, os elementos à direita da posição corrente estão desordenados, enquanto que os elementos à esquerda dele estão ordenados.

Desempenho: $O(n^2)$ no pior caso.

Exemplo:

Considere min = elemento mínimo e o seguinte vetor:

$$B = \{8, 6, 1, 2, 7\}$$

Percorremos o ele até encontrar o menor elemento. Devemos comparar um determinado elemento na posição i, com os elementos presentes nas posições que o antecede - a partir da segunda posição do vetor. Então:

B =	**8**	**6**	**1**	**2**

6 < 8? Sim. Então trocamos eles de posição.

6	**8**	1	2

1 < 8? Sim.

6	1	**8**	2

1 < 6? Sim.

1	**6**	**8**	2

2 < 8? Sim

1	**6**	2	**8**

2 < 6? Sim.

1	2	**6**	**8**

2 < 1? Não.

1	**2**	**6**	**8**

- **Selection Sort**

Selection Sort, ou ordenação por seleção, aplica a seguinte ideia: escolhe-se o menor/ maior elemento do vetor, depois o segundo menor/ maior e assim por diante. Em cada etapa descrita anteriormente, movimenta-se o elemento obtido (maior ou menor do array) para a primeira ou última posição da sequência.

Desempenho: $O(n^2)$ no pior caso.

Exemplo:

Considere mín = elemento mínimo e o seguinte vetor:

$$V = \{8, 10, 5\}.$$

Inicialmente, encontra-se o menor elemento. Nesse caso,

$$mín = 5$$

Movimentamos ele para a *primeira* posição:

$$V = \{5, 10, 8\}$$

Percorremos novamente o vetor e encontramos:

$$mín = 8$$

Movimentamos ele para a *segunda* posição. Então ficamos com :

$$\mathbf{V = \{5, 8, 10\}}$$

- **Bubble Sort**

Bubble sort, ou ordenação por flutuação, aplica a seguinte ideia: dois a dois os elementos são comparados até o fim do vetor; a cada comparação, há a troca ou não entre eles (depende se já está ou não ordenado). O objetivo é “flutuar” o maior elemento para a última posição.

Desempenho: $O(n^2)$ no pior caso.

Exemplo:

Considere o vetor:

$$V = \{15, 6, 8, 1\}$$

Devemos primeiramente comparar os dois primeiros elementos:

- 15 é maior do que 6? Sim, então trocamos eles de posição.

$$V = \{15, 6, 8, 1\} \rightarrow V = \{6, 15, 8, 1\}$$

Repetindo, temos:

- 15 é maior do que 8? Sim, logo:

$$V = \{6, 15, 8, 1\} \rightarrow V = \{6, 8, 15, 1\}$$

- 15 > 1? Sim.

$$V = \{6, 8, 15, 1\} \rightarrow V = \{6, 8, 1, 15\}$$

Portanto, o 15 já está na posição final, logo, devemos repetir os processos anteriores até a posição *n – 2.*

- 6 > 8, Não. Sendo assim, não há troca.
- 8 > 1. Sim. Então:

$$V = \{6, 1, 8, 15\}$$

Observe que 8 e 15 já estão nas suas posições corretas.

Continuando:

- 6 > 1. Sim.

$$\boldsymbol{V = \{1, 6, 8, 15\}}$$

- **Merge Sort**

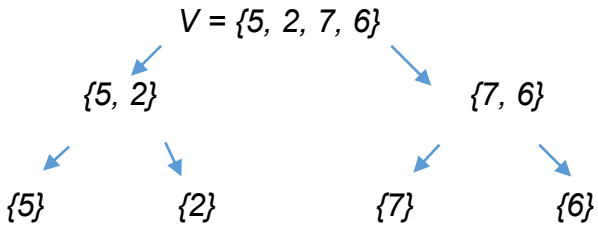
É do tipo dividir e conquistar. Consiste em dividir o problema em problemas menores (subproblemas) através da recursão, resolvendo-os. Em seguida, após resolver os subproblemas, é feita a união das soluções dos mesmos.

Desempenho: consome n* $\log_2$ (n) unidades de tempo no pior caso.

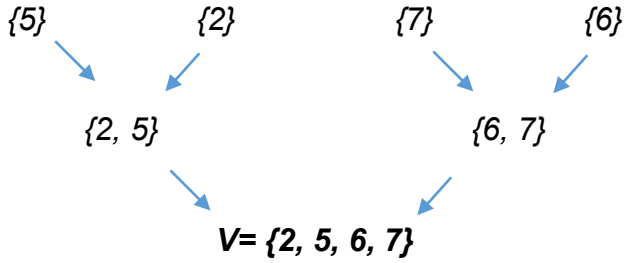
Exemplo:

V = {5, 2, 5, 6}

Dividimos o vetor V em subvetores até o caso básico:



E então retrocedemos ordenando os subvetores do caso básico:



- **Quick Sort**

Também do tipo dividir e conquistar, o Quick sort é um método de ordenação instável. A ideia básica consiste em: a partir de um elemento do vetor denominado pivô (estabelecido de acordo com critérios do programador), rearranja a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os posteriores ao pivô sejam maiores que o mesmo. Por fim, o pivô encontrara-se na sua posição final e então haverá sublistas não ordenadas. Tal processo é repetido recursivamente quantas vezes forem necessárias de modo que as sublistas sejam ordenadas.

Desempenho: consome, em média, n* $\log_2$ (n) unidades de tempo - e n^2 no pior caso.

Exemplo:

Considere o vetor V = {2, 23, 89, 7, 14, 35}

2	9	8	1	3	5	6	4
2	9	8	1	3	5	6	4
2	9	8	1	3	5	6	4
2	9	8	1	3	5	6	4
2	9	8	1	3	5	6	4
2	1	8	9	3	5	6	4
2	1	3	9	8	5	6	4
2	1	3	9	8	5	6	4
2	1	3	9	8	5	6	4
2	1	3	4	8	5	6	9

Pivô
Partição 2
Partição 1

As mesma etapas são realizadas recursivamente nas partições 1 e 2.

- **Radix Sort**

É um algoritmo de ordenação do tipo estável. A ordenação é realizada em função dos dígitos. Ideia (MSD): começa do dígito menos significativo até o mais significativo. É necessário utilizar um segundo algoritmo estável para fazer a ordenação de cada dígito.

Desempenho: O(n) (utilizando-se o counting sort).

Exemplo:

V = {170, 045, 075, 090, 002, 024, 802, 066}

Passo 1: Digito menos significativo: 17**0**, 09**0**, 00**2**, 80**2**, 02**4**,, 04**5,** 07**5,** 00**6.**

Dígito	Qtd	Valores
0	2	170
2	2	002, 802
4	1	024
5	2	045, 075
6	1	066

Passo 2: Andar uma casa para a esquerda e fazer o mesmo processo do passo 1.

Dígito	Qtd	Valores
0	2	002, 802
2	1	024
4	1	045
6	1	066
7	2	170, 075
9	1	090

Passo 3: Andar uma cada para a esquerda e repete-se o passo 1.

Dígito	Qtd	Valores
0	6	002, 045, 075 066, 090
1	1	170
8	1	802

- **Counting Sort**

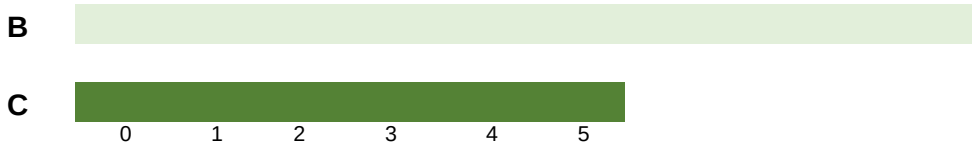
A ideia básica do counting sort consiste em: determina-se, para cada entrada n, o número de elementos menores que n, para que assim, após essa etapa, n esteja na sua posição final.

Complexidade: O(n).

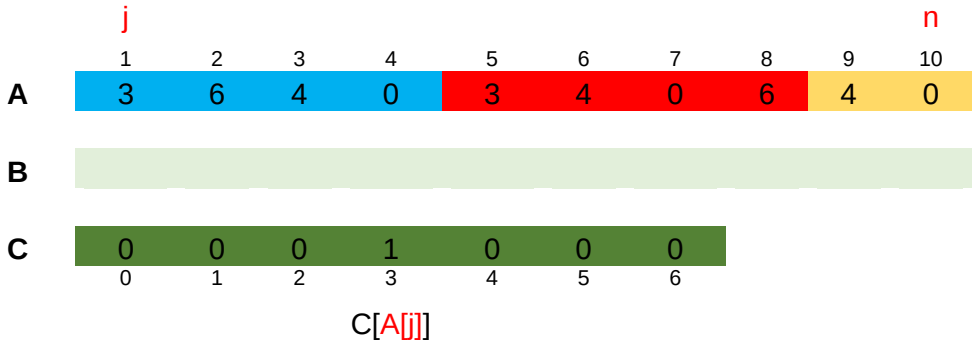
Exemplo:

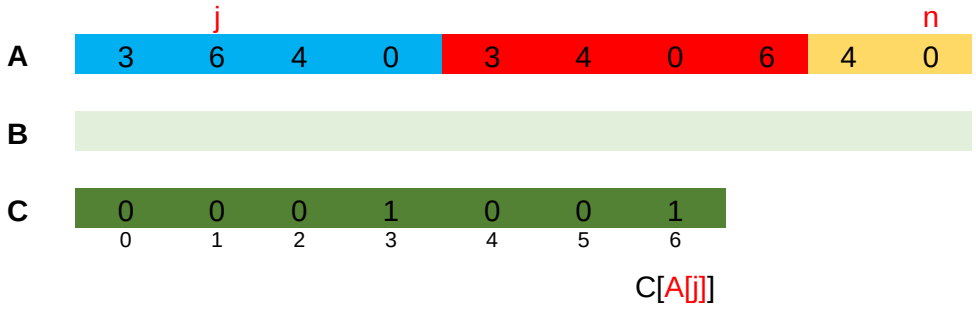
Deve resultar em B (ordenado):

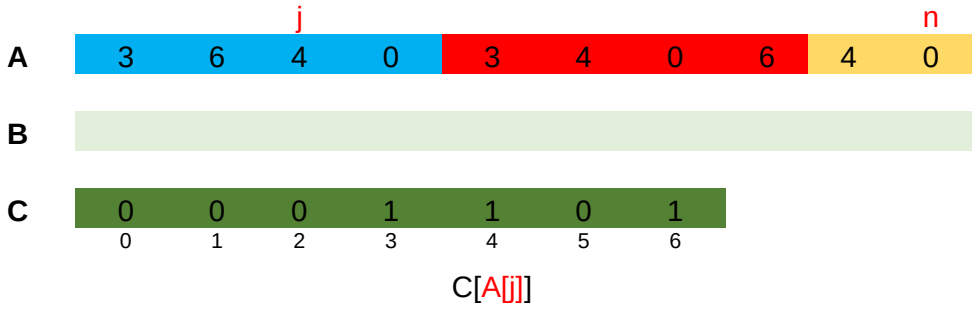
Inicializa-se os vetores auxiliares B e C:

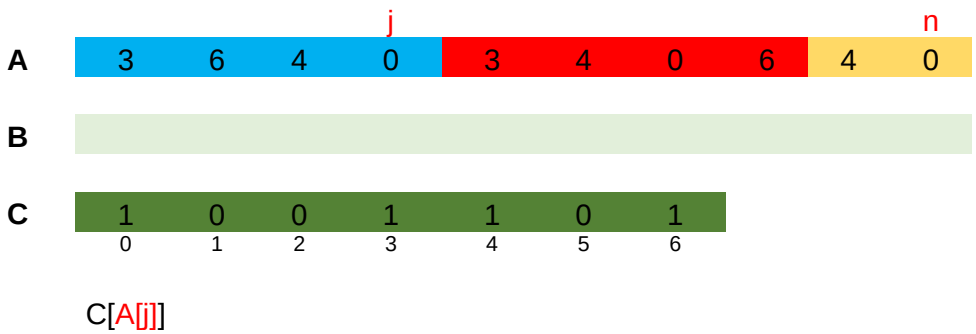


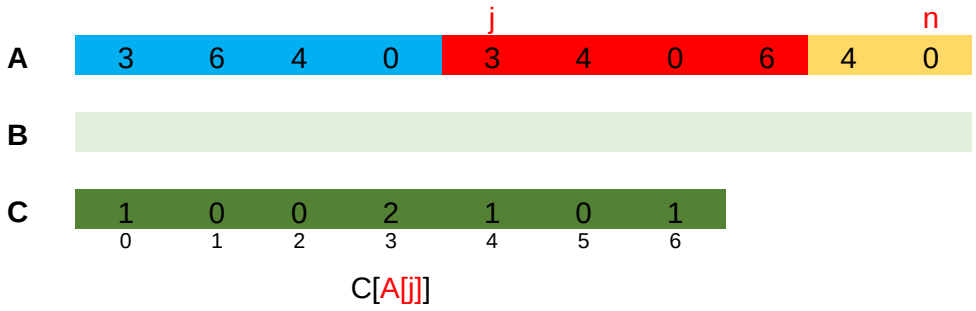
A variável j varre *A*.

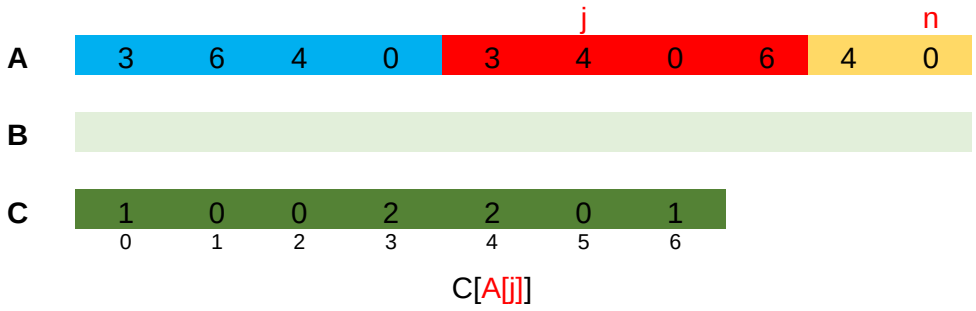


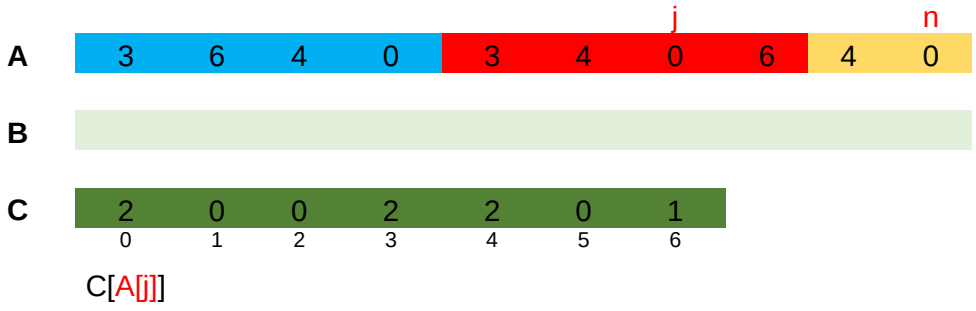


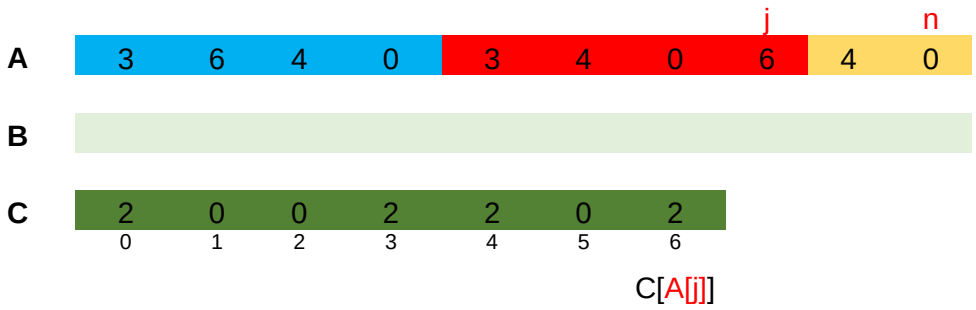


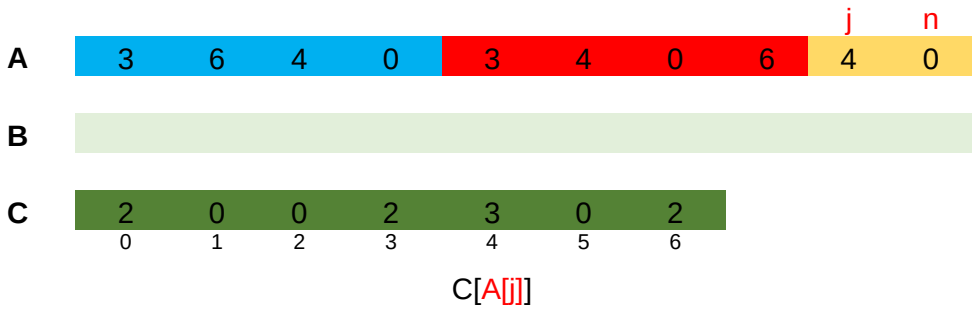


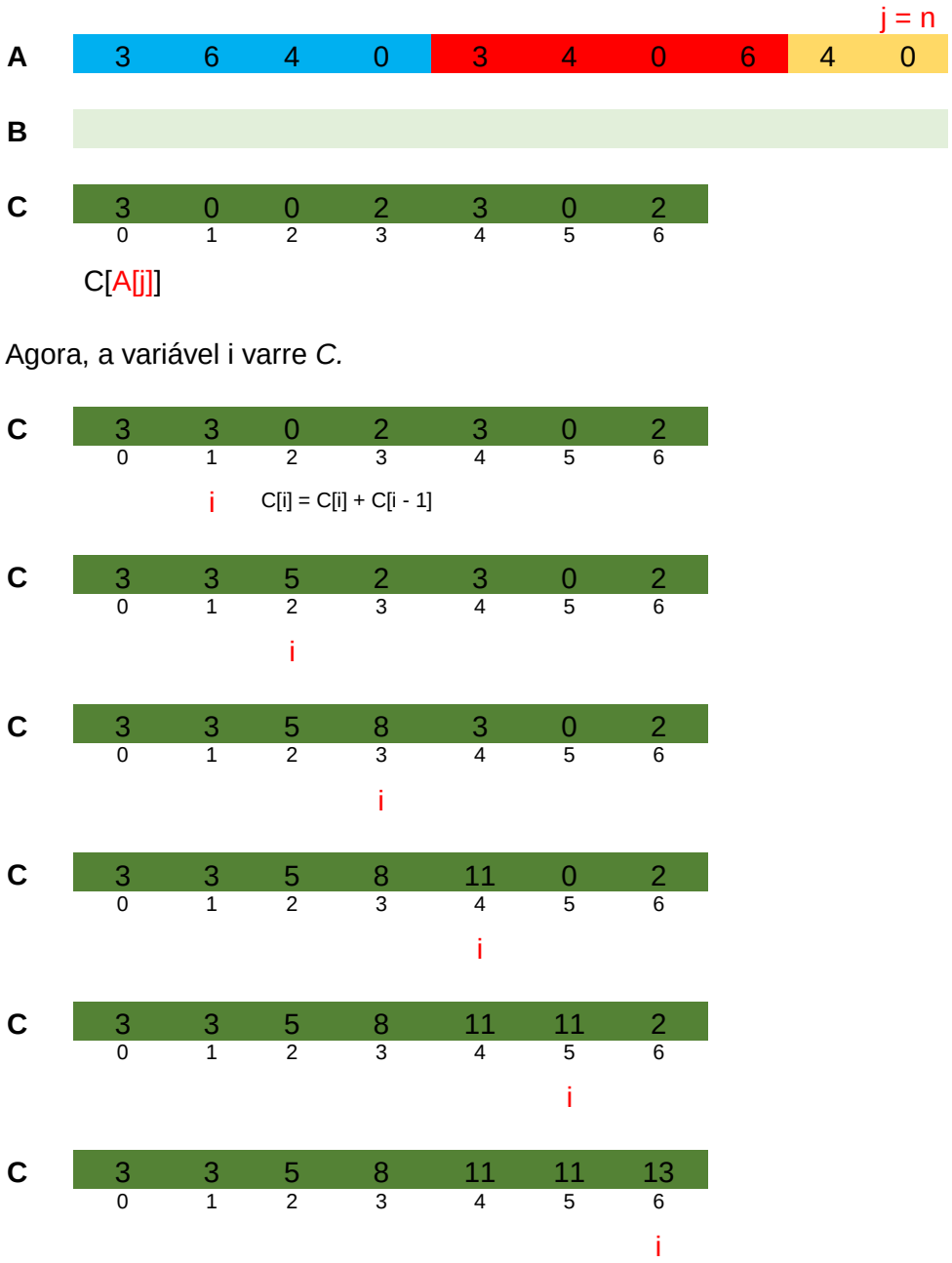

j
n
A
3 6 4 0 3 4 0 6 4 0
B
C
1 0 0 2 1 0 1
0 1 2 3 4 5 6
C[A[j]]


j
n
A
3 6 4 0 3 4 0 6 4 0
B
C
1 0 0 2 2 0 1
0 1 2 3 4 5 6
C[A[j]]


j
n
A
3 6 4 0 3 4 0 6 4 0
B
C
2 0 0 2 2 0 1
0 1 2 3 4 5 6
C[A[j]]

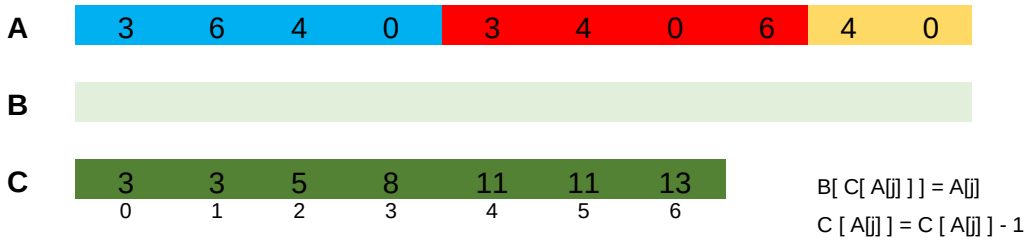

j
n
A
3 6 4 0 3 4 0 6 4 0
B
C
2 0 0 2 2 0 2
0 1 2 3 4 5 6
C[A[j]]


j
n
A
3 6 4 0 3 4 0 6 4 0
B
C
2 0 0 2 3 0 2
0 1 2 3 4 5 6
C[A[j]]

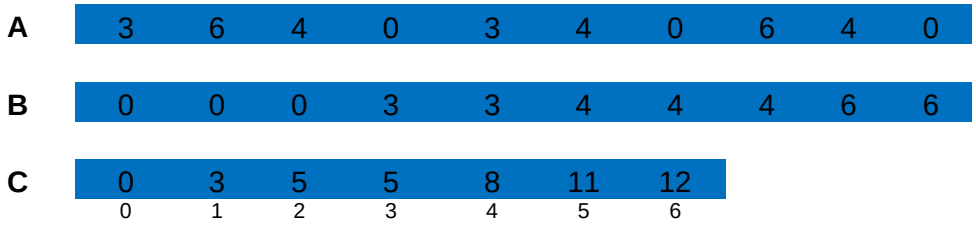


Agora, a variável i varre *C*.

Agora, a variável *j* varre *C*.



Fazendo esse procedimento, chegamos em:



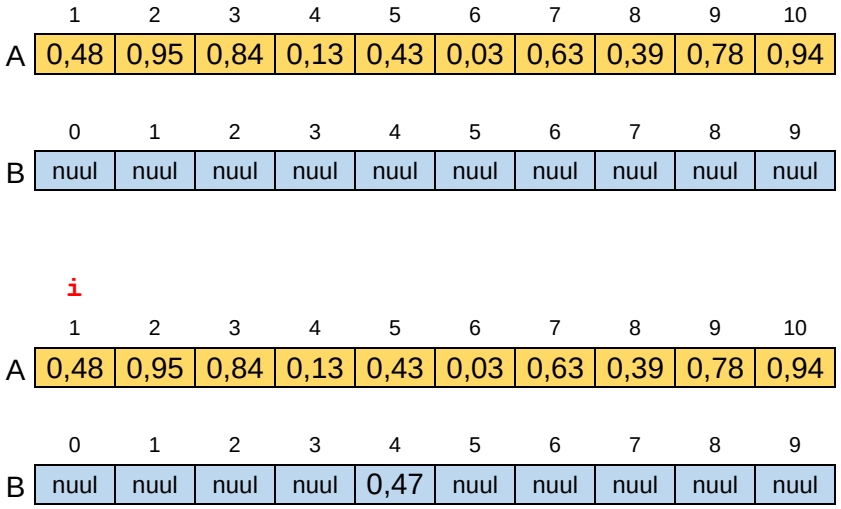
- **Bucket Sort**

É um método de ordenação estável. A ideia básica é a seguinte: divide-se os elementos do vetor a ser ordenado em baldes. Em seguida, ordena-se os elementos de cada baldes. Por fim, agrupa os elementos.

Desempenho: O(n).

Exemplo:

Cada elemento de A é inserido no Bucket correspondente:



i

	1	2	3	4	5	6	7	8	9	10
A	0,48	0,95	0,84	0,13	0,43	0,03	0,63	0,39	0,78	0,94

	0	1	2	3	4	5	6	7	8	9
B	nuul	nuul	nuul	nuul	0,47	nuul	nuul	nuul	nuul	0,95

i

	1	2	3	4	5	6	7	8	9	10
A	0,48	0,95	0,84	0,13	0,43	0,03	0,63	0,39	0,78	0,94

	0	1	2	3	4	5	6	7	8	9
B	nuul	nuul	nuul	nuul	0,47	nuul	nuul	nuul	0,84	0,95

i

	1	2	3	4	5	6	7	8	9	10
A	0,48	0,95	0,84	0,13	0,43	0,03	0,63	0,39	0,78	0,94

	0	1	2	3	4	5	6	7	8	9
B	nuul	0,13	nuul	nuul	0,47	nuul	nuul	nuul	0,84	0,95

i

	1	2	3	4	5	6	7	8	9	10
A	0,48	0,95	0,84	0,13	0,43	0,03	0,63	0,39	0,78	0,94

	0	1	2	3	4	5	6	7	8	9
B	nuul	0,13	nuul	nuul	0,47	nuul	nuul	nuul	0,84	0,95
					0,43					

i

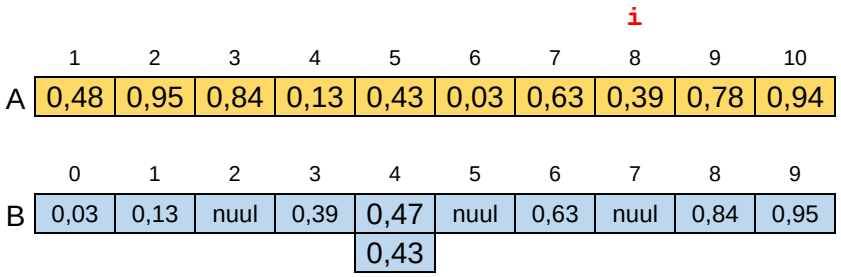
	1	2	3	4	5	6	7	8	9	10
A	0,48	0,95	0,84	0,13	0,43	0,03	0,63	0,39	0,78	0,94

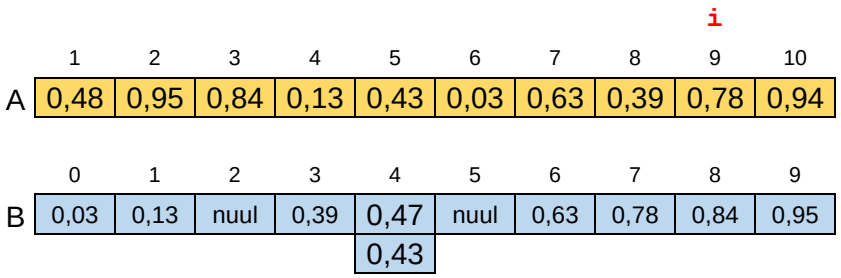
	0	1	2	3	4	5	6	7	8	9
B	0,03	0,13	nuul	nuul	0,47	nuul	nuul	nuul	0,84	0,95
					0,43					

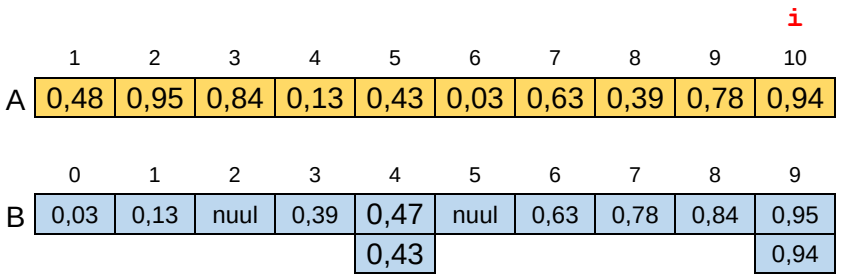
i

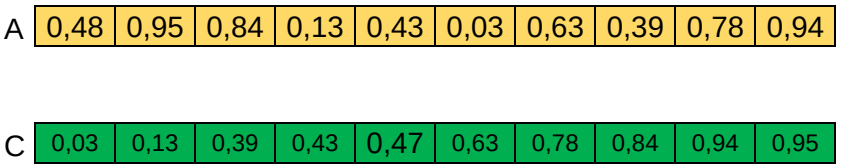
	1	2	3	4	5	6	7	8	9	10
A	0,48	0,95	0,84	0,13	0,43	0,03	0,63	0,39	0,78	0,94

	0	1	2	3	4	5	6	7	8	9
B	0,03	0,13	nuul	nuul	0,47	nuul	0,63	nuul	0,84	0,95
					0,43					









Por fim, os buckets concatenados geram o vetor C.

2. ANÁLISE EMPÍRICA

2.1. Descrição da máquina

- Notebook Dell Inspiron;
- RAM instalada: 4GB;
- Processador: Intel Core i5-5200U CPU 2.20GHz x 4;
- Intel HD Graphics 5500 (Broadwell GT2);
- Tipo de sistema: 64-bit;
- SO: Ubuntu 16.04 LTS;
- Ferramenta utilizada: **Code::Blocks 16.01**.

2.2. Recursos de linguagem

Foram utilizados os seguintes recursos da linguagem C:

- Alocação dinâmica de vetores;
- Structs;
- Ponteiros;
- Manipulação de arquivos (gravação, abertura e leitura);
- Função *rand/srand* para gerar as sequências de números de maneira aleatória.

2.3. Implementações

Determinação do tamanho máximo suportado pela máquina

Após diversos teste, foi concluído que o tamanho máximo suportado pela máquina foi 1,5 bilhão. Entretanto, embora este tenha sido o maior valor, foi necessário reduzir para 1,5 bilhão, tendo em vista que a máquina não suportou ordenar sequências desse tamanho (dois bilhões) – travava durante a execução, necessitando ser reiniciada – com a utilização de métodos com complexidade n^2. O tamanho máximo foi definido sobre as seguintes circunstâncias: vetor alocado dinamicamente, variável local. Resultados encontrados:

- Variável local sem alocação dinâmica: 1.999.999;

- Variável global sem alocação: 1.400.000.000;

- Variável local com alocação: 1.500.000.000;

Funções utilizadas

- **Código para gerar as sequências:**

```c
#include <stdio.h>
#include <stdlib.h>// para rand() e srand()
#include <time.h> //referente a funcao time()

int main(void) {
	int i, regnum;
	FILE *fptr;
	int TAM; //qtd de numeros que serao gerados

	scanf("%i", &TAM);

	srand(time(NULL)); //permite que a "semente" seja diferente em cada execucao

	fptr = fopen("seq.txt", "w"); //abre o arquivo para gravacao em modo texto

	for (i = 0; i < TAM; i++) {
		// gerando valores aleatorios na faixa de 0 a %x
		regnum = rand() % TAM;

		fprintf(fptr, "%i\n", regnum); //armazena no arquivo
	}

	fclose(fptr);

	/*O seguinte trecho pode ser utilizado para a realizacao de testes */

	// fptr = fopen("seq.txt", "r");
	//
	// while(fscanf(fptr, "%i\n", &regnum) != EOF)
	//    printf("%i\n", regnum);
	//
	//
	// fclose(fptr);

	 return 0;
}
```

- **Selection Sort**

```
void selection (int vetor[], int TAM) {

    int min, aux, i, j;

    for (i = 0; i < (TAM - 1); i++) {

        min = i;

        //Procura o menor elemento do conjunto de numeros desodernados
        for (j = (i + 1); j < TAM; j++) {
            if(vetor[j] < vetor[min]) {
              min = j;
            }
        }

        //Realiza a troca entre o menor elemento encontrado e a posição i
        if (vetor[i] != vetor[min]) {
            aux = vetor[i];
            vetor[i] = vetor[min];
            vetor[min] = aux;
        }
    }
}
```

- **Insertion Sort**

```
void insertion (int vetor [], int n) {

    int i, key, j, cont = 0;

    for (i = 1; i < n; i++) {
       key = vetor[i]; //key = auxiliar
       j = i - 1;

      //deslocamento dos menores elementos para frente
       while ((j >= 0) && (vetor[j] > key)) {
           vetor[j + 1] = vetor[j];
           j = j - 1;
       }

       vetor[j + 1] = key;
    }
}
```

- **Bubble Sort**

```
void bubble (int vetor[], int n) {

    int i, fim, aux;

    for(fim = n - 1; fim > 0; fim--) {

      //enquanto fim /= de 0, a cada iteração um element maior é deslocado
      //para o fim do vetor
        for(i = 0; i < fim; i++) {

            //Realização da troca
            if(vetor[i] > vetor[i + 1]) {
                aux = vetor[i];
                vetor[i] = vetor[i + 1];
                vetor[i + 1] = aux;
            }
        }
    }
}
```

- **Quick Sort**

```
//funcao chamada pela funcao quick
int partition(int vetor[], int p, int r) {

    int aux, pivo = vetor[r], i = p - 1, j;

   //recua posicao da direita para a esquerda
    for (j = p; j <= r - 1; j++) {

      //troca esquerda e direita
        if(vetor[j] <= pivo) {
            i = i + 1;
            aux = vetor[i];
            vetor[i] = vetor[j];
            vetor[j] = aux;
        }
    }

    aux = vetor[i + 1];
    vetor[i + 1] = vetor[r];
    vetor[r] = aux;

    return (i + 1);
}

void quick(int vetor[], int p, int r) {

    int q, j;

    if (p < r) {

        q = partition (vetor, p, r); //separa os dados em particoes
        quick(vetor, p, q - 1); //chama a funcao para um das metades
        quick(vetor, q + 1, r); //chama a funcao para a outra metade
    }
}
```

- **Merge Sort**

```
void merge(int *V, int inicio, int meio, int fim){
    int *temp, p1, p2, tamanho, i, j, k;
    int fim1 = 0, fim2 = 0;

    tamanho = fim - inicio + 1;
    p1 = inicio;
    p2 = meio + 1;

    temp = (int *) malloc(tamanho * sizeof(int));

    if (temp != NULL) {
        for (i = 0; i < tamanho; i++) {
            if (!fim1 && !fim2) {

             //combina ordenando
                if (V[p1] < V[p2])
                    temp[i] = V[p1++];

                else
                    temp[i] = V[p2++];
            //os dois if's verifica se o vetor acabou
                if (p1 > meio)
                    fim1 = 1;
                if (p2 > fim)
                    fim2 = 2;
            }

             //copia o que sobrar
             else {
                if (!fim1)
                    temp[i] = V[p1++];

                else
                    temp[i] = V[p2++];
             }

        }

       //copiar do auxiliar para o original
        for (j = 0, k = inicio; j < tamanho; j++, k++)
            V[k] = temp[j];
    }

    free(temp);
}

void mergeSort(int *V, int inicio, int fim){

    if(inicio < fim){
        int meio = floor(inicio + fim) / 2;

        mergeSort(V, inicio, meio); //chama a funcao para uma das metades
        mergeSort(V, meio + 1, fim); //chama a funcao para a outra metade
        merge(V, inicio, meio, fim); //combina as 2 metades de forma ordenada
    }
}
```

- **Counting Sort**

```
void counting (int vetor[], int n) {

    int i, j, k;
    int baldes[TAM];

      //inicializa todas as posições do vetor balde com o valor 0
      for (i = 0; i < TAM; i++)
        baldes[i] = 0;

      //vare o vetor principal e contabiliza
      //a ocorrencia dos elementos no vetor baldes
      for (i = 0; i < n; i++)
        baldes[vetor[i]]++;

      //vare o vetor baldes verificando a ocorrencia dos elementos
      //e add na posição correta em vetor
      for(i = 0, j = 0; j < TAM; j++) {
        for (k = baldes[j]; k > 0; k--)
            vetor[i++] = j;
    }
}
```

- **Bucket Sort**

```
void bubble (int vetor[], int n) {

    int i, fim, aux;

    for(fim = n - 1; fim > 0; fim--) {

      //enquanto fim /= de 0, a cada iteração um element maior é deslocado
      //para o fim do vetor
        for(i = 0; i < fim; i++) {

            //Realização da troca
            if(vetor[i] > vetor[i + 1]) {
                aux = vetor[i];
                vetor[i] = vetor[i + 1];
                vetor[i + 1] = aux;
            }
        }
    }
}


void bucketSort(int v[],int tam){

        int num_bucket;
        num_bucket = tam / 10;


         struct balde b[num_bucket];
         int i, j, k;
 /* 1 */ for (i=0; i < num_bucket;i++)   //inicializa todos os "topo"
                b[i].topo=0;
```

```
 /* 2 */ for(i=0;i<tam;i++){     //verifica em que balde o elemento deve ficar
                j=(num_bucket)-1;
                while(1){
                        if(j<0)
                                break;
                        if(v[i]>=j*10){
                                b[j].balde[b[j].topo]=v[i];
                                (b[j].topo)++;
                                break;
                        }
                        j--;
                }
        }

 /* 3 */ for(i=0;i<num_bucket;i++)  //ordena os baldes
                if(b[i].topo)
                        bubble(b[i].balde,b[i].topo);

        i=0;
 /* 4 */ for(j=0;j<num_bucket;j++){  //põe os elementos dos baldes de volta no vetor
                for(k=0;k<b[j].topo;k++){
                        v[i]=b[j].balde[k];
                        i++;
                }
        }

      bubble(v, tam);
 }
```

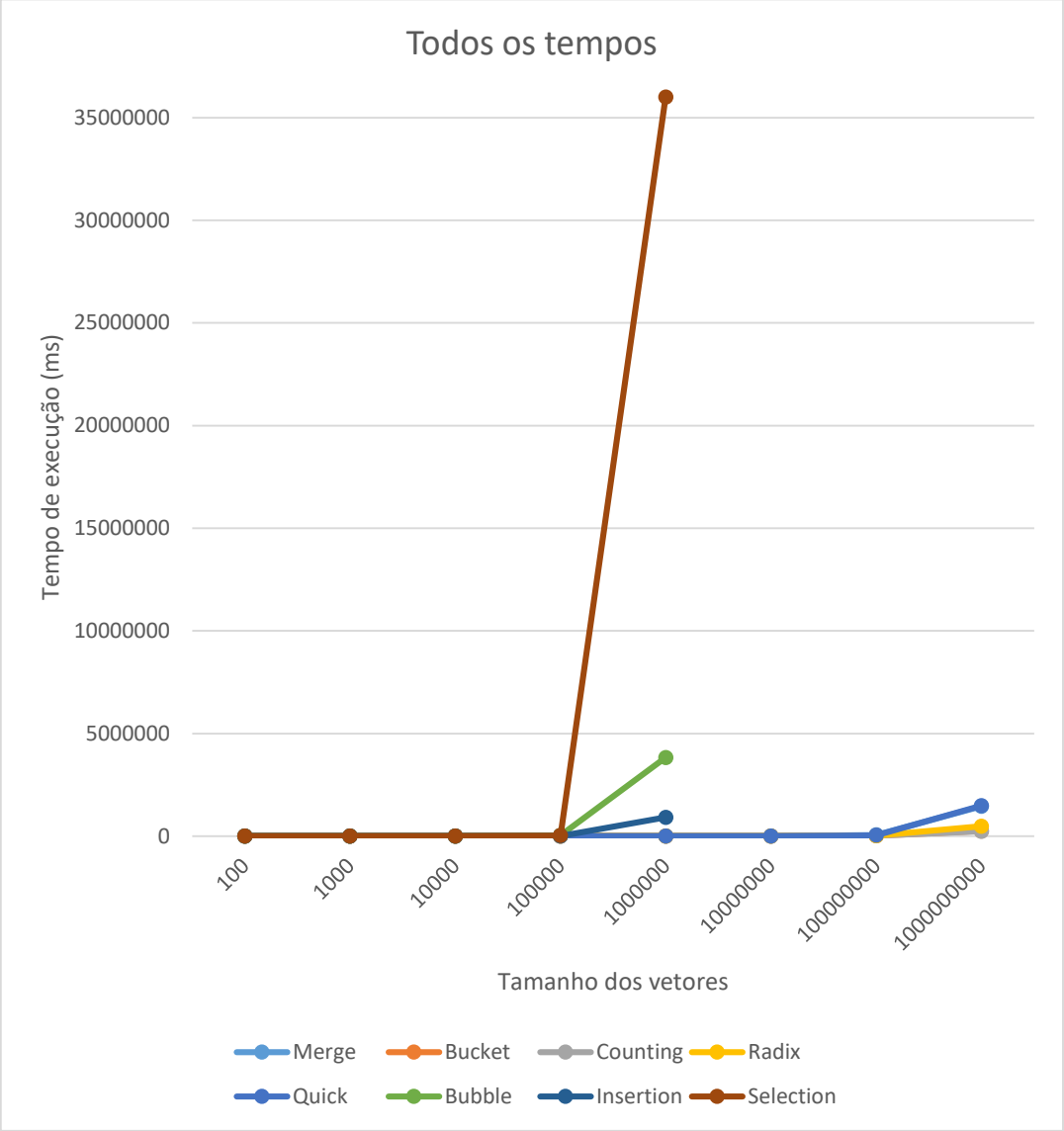
- **Radix Sort**

```
void radix(int *vet, int n) {
      int i, exp = 1, m = 0, bucket[n], temp[n];

      //busca o maior elemento do vetor
      for(i = 0; i < n; i++) {
            if(vet[i] > m) {
                  m = vet[i];
            }
      }

      while((m/exp) > 0) {

            //inicializa o vetor bucket com o valor 0
            for (i = 0; i < n; i++) {
                  bucket[i] = 0;
            }

            for (i = 0; i < n; i++) {
                  bucket[(vet[i] / exp) % 10]++;
            }

            for( i = 1; i < n; i++) {
                  bucket[i] += bucket[i-1];
            }

            for (i = (n - 1); i >= 0; i--) {
```

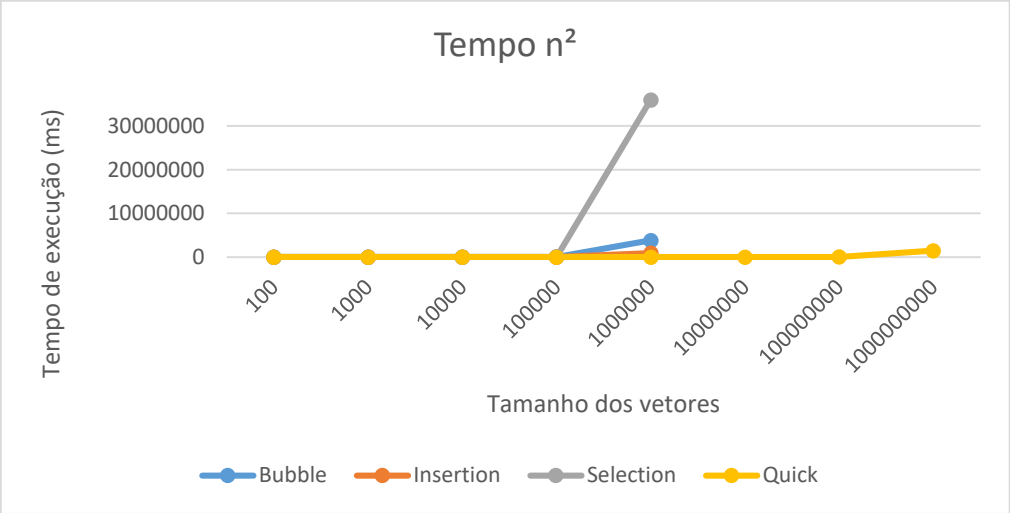
```
            temp[--bucket[(vet[i] / exp) % 10]] = vet[i];
        }

        for (i = 0; i < n; i++) {
            vet[i] = temp[i];
        }

        exp *= 10;
    }
}
```

2.4. Gráficos

2.4.1. Todos os métodos



2.4.2 Apenas métodos com tempo n²



Selection

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	1,5
30	1000	4,7
20	10000	217,5
40	100000	27863,22
20	1000000	3657863,22
30	10000000	Não executado completamente
25	100000000	
20	1000000000	

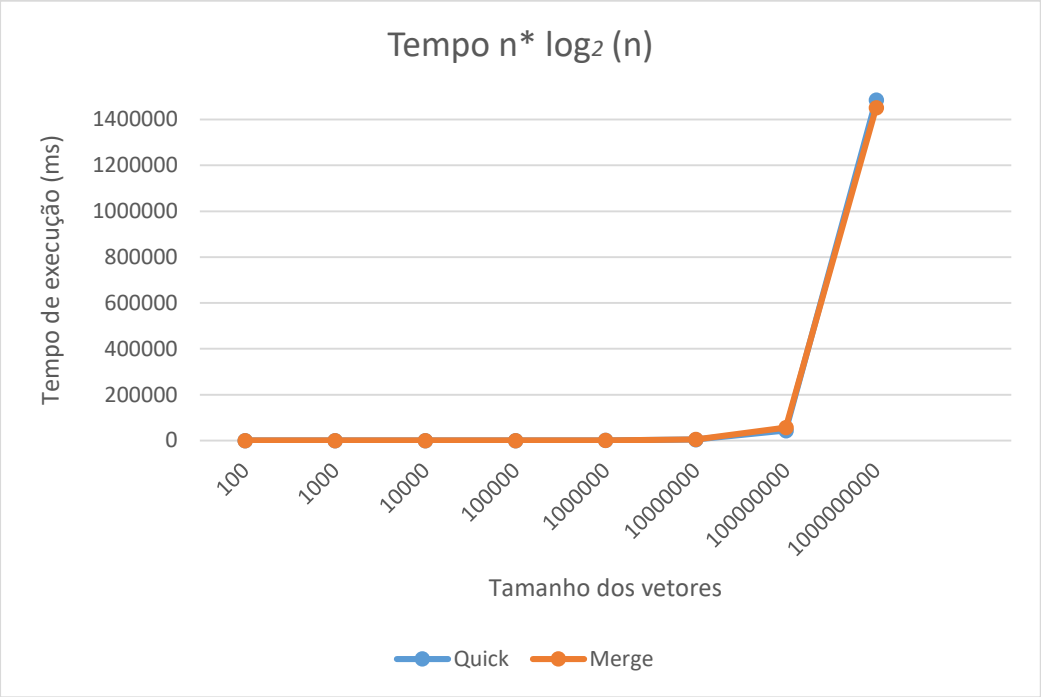
Insertion

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	2
30	1000	5,5
20	10000	223
40	100000	8947,75
20	1000000	915987,5
30	10000000	Não executado completamente
25	100000000	
20	1000000000	

Bubble

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	1,5
30	1000	5,3
20	10000	410
40	100000	31801
20	1000000	3825671,5
30	10000000	Não executado completamente
25	100000000	
20	1000000000	

2.4.3. Apenas métodos com tempo n * log_2 (n)



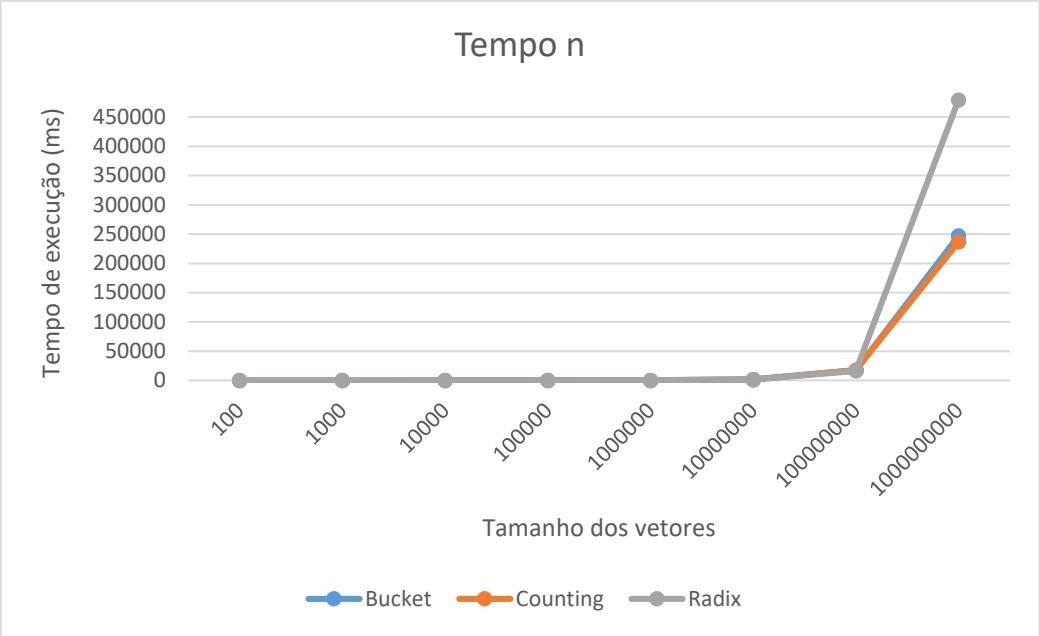
Quick

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	1,5
30	1000	5,5
20	10000	7
40	100000	49,5
20	1000000	424
30	10000000	4021,25
25	100000000	43265,5
20	1000000000	1486031,5

Merge

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	1,5
30	1000	2,5
20	10000	6,5
40	100000	76
20	1000000	581,5
30	10000000	4912
25	100000000	56772,5
20	1000000000	1451683

2.4.4. Apenas métodos com tempo n



Bucket

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	2
30	1000	4,25
20	10000	170,66
40	100000	195
20	1000000	385,5
30	10000000	1825,5
25	100000000	17488
20	1000000000	247061,5

Counting

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	1,3
30	1000	2,5
20	10000	4,1
40	100000	31,5
20	1000000	278,5
30	10000000	1314,9
25	100000000	17682,7
20	1000000000	236696,5

Radix

Número de vetores	Tamanho dos vetores	Tempo médio (ms)
20	100	1,3
30	1000	2,5
20	10000	4,5
40	100000	46,5
20	1000000	405,5
30	10000000	1665
25	100000000	16501,5
20	1000000000	479416,5

BIBIOGRAFIA

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. Introduction to Algorithms, 3a edição, MIT Press, 2009.

FEOFILOFF, P. Algoritmos em Linguagem C 2ª edição, Campus/ Elsevier. 2008 – 2009.

Slides da professora Nádia Félix e dos professores Hebert Coelho e Wanderley Alencar utilizados nas aulas da disciplina AED 1.

Slides do professor Jairo Souza da UFJF. Disponível em: http://www.ufjf.br/jairo_souza/files/2009/12/2-Ordena%C3%A7%C3%A3o-QuickSort.pdf

Slides do professor Túlio Toffolo da UFPO. Disponível em: http://www.decom.ufop.br/toffolo/site_media/uploads/2013-1/bcc202/slides/19._ordenacao_em_tempo_linear.pdf

Slides dos professores Adiano Jorge, Diogenes Laertius, João Vitor, José Arnóbio, Pedro Victor da UFAL. Disponível em: https://www.passeidireto.com/arquivo/6257293/radix-sort-aplicacao-e-exemplos

Site Wikipedia.

Site Khan Academy.

Site Programação Descomplicada (programacaodescomplicada.wordpress.com).