

THIAGO GOMEZ DE SOUZA
LEONARDO RIBEIRO OLIVEIRA PALMEIRA

TRABALHO PRÁTICO 1

**RELATÓRIO SOBRE A ANÁLISE ASSINTÓTICA (SIMPLES) E
EMPÍRICA DOS MÉTODOS DE ORDENAÇÃO VISTOS EM SALA NA
DISCIPLINA DE ALGORITMOS E ESTRUTURAS DE DADOS 1.**

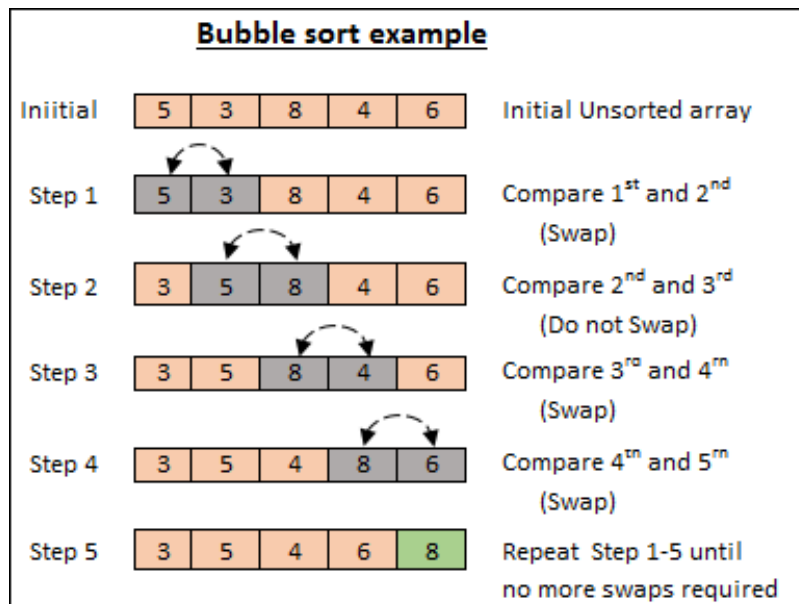
Professor: Hebert Coelho

Goiânia - GO
2023

ANÁLISE ASSINTÓTICA:

Método Bubble Sort:

Este é um método de ordenação simples que funciona comparando cada elemento do vetor com o elemento imediatamente seguinte. Se o elemento atual for maior do que o próximo, eles são trocados de lugar. Este processo é repetido até que o vetor esteja completamente ordenado. Análise assintótica: $O(n^2)$;



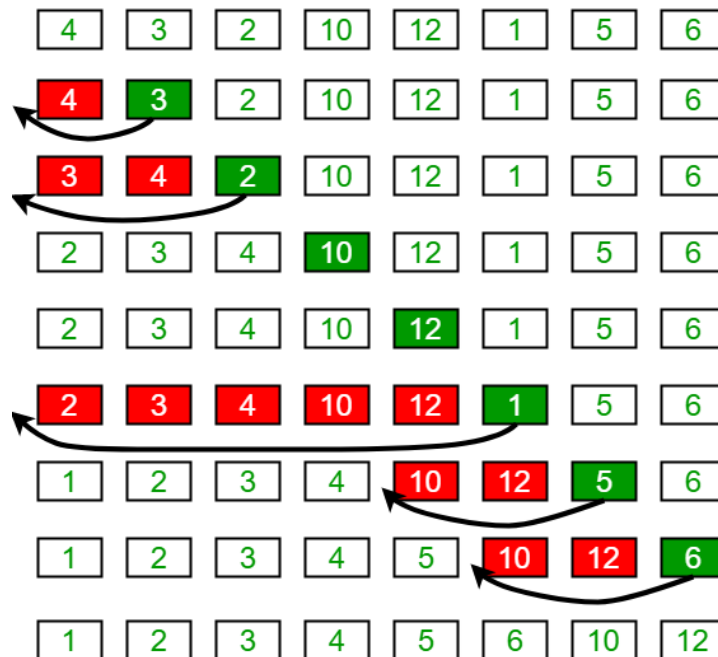
```
#include "Bubble_sort.h"

void bubble_sort(int* arr, int size) {
    int i, j, temp;
    for (i = 0; i < size-1; i++) {
        for (j = 0; j < size-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Método Insertion Sort:

Este método funciona dividindo o vetor em dois sub-vetores: um ordenado e um não ordenado. Cada elemento do subvetor não ordenado é inserido na posição correta no subvetor ordenado através de comparações e trocas. Este processo é repetido até que todo o vetor esteja ordenado. Análise assintótica: $O(n^2)$;

Insertion Sort Execution Example

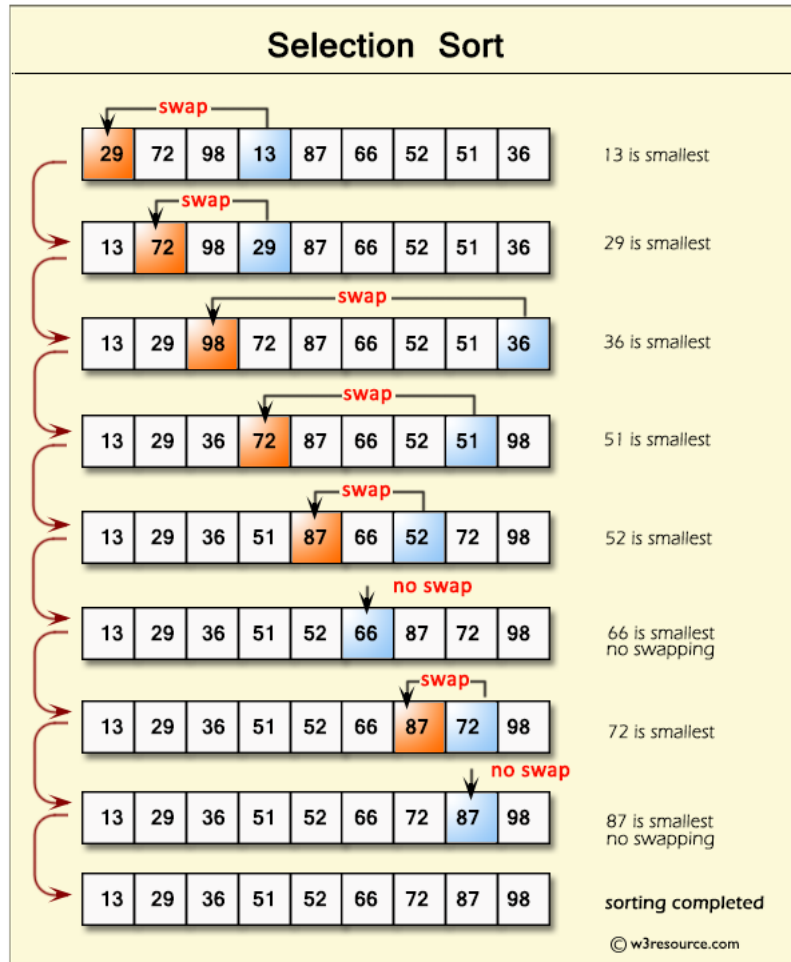


```
#include "Insertion_sort.h"
```

```
void insertion_sort(int* arr, int size) {  
    int i, key, j;  
    for (i = 1; i < size; i++) {  
        key = arr[i];  
        j = i-1;  
        while (j >= 0 && arr[j] > key) {  
            arr[j+1] = arr[j];  
            j = j-1;  
        }  
        arr[j+1] = key;  
    }  
}
```

Método Selection sort:

Este método funciona selecionando o menor elemento do vetor não ordenado e colocando-o no início do vetor ordenado. Este processo é repetido até que todo o vetor esteja ordenado. Análise assintótica: $O(n^2)$;



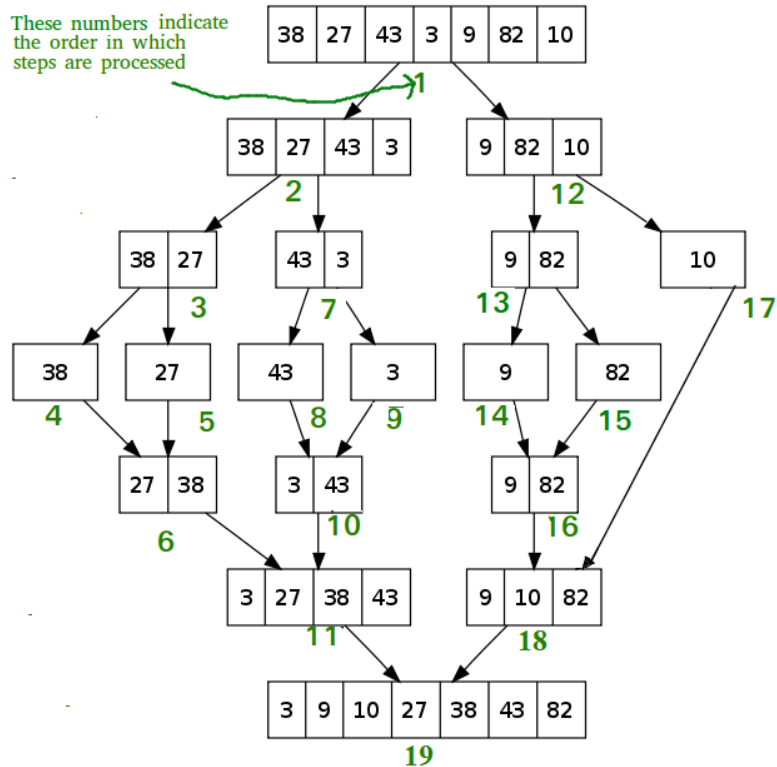
```
#include "Selection_sort.h"

void selection_sort(int* arr, int size) {
    int i, j, min_idx, temp;
    for (i = 0; i < size-1; i++) {
        min_idx = i;
        for (j = i+1; j < size; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Método Merge Sort:

Este método funciona dividindo o vetor em dois sub-vetores e ordenando-os individualmente antes de juntá-los novamente em um vetor ordenado completo.

Análise assintótica: $O(n \log n)$;



```
#include "Merge_sort.h"
#include <stdlib.h>

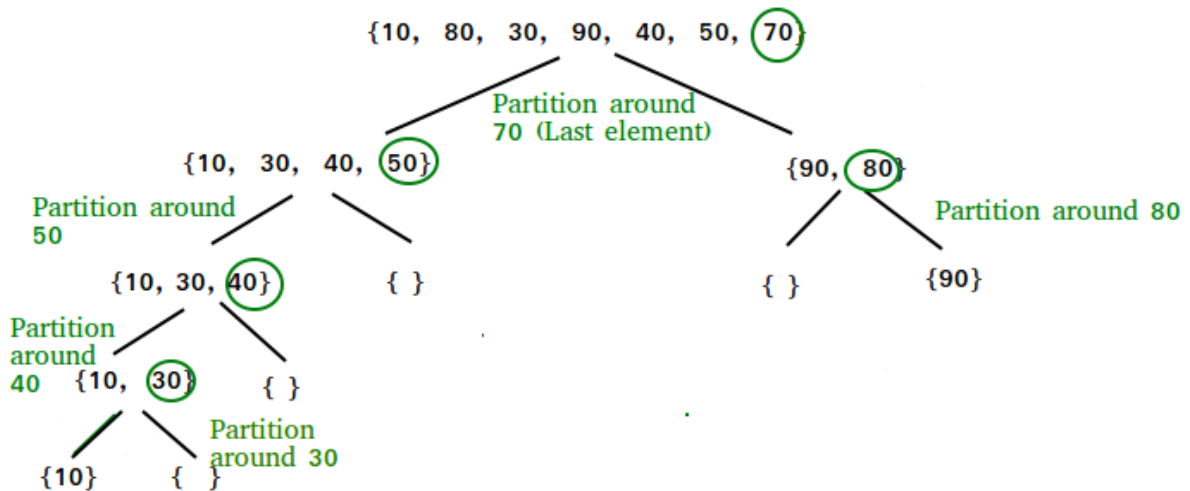
void merge(int *arr, int *size, int i, int m, int f) {
    int z, iv = i, ic = m + 1;
    for (z = i; z <= f; z++) size[z] = arr[z];
    z = i;
    while (iv <= m && ic <= f) {
        if (size[iv] <= size[ic]) arr[z++] = size[iv++];
        else arr[z++] = size[ic++];
    }
    while (iv <= m) arr[z++] = size[iv++];
    while (ic <= f) arr[z++] = size[ic++];
}

void sort(int *arr, int *size, int i, int f) {
    if (i >= f) return;
    int m = (i + f) / 2;
    sort(arr, size, i, m);
    sort(arr, size, m + 1, f);
    if (arr[m] <= arr[m + 1]) return;
    merge(arr, size, i, m, f);
}

void merge_sort(int *arr, int n) {
    int *size = malloc(sizeof(int) * n);
    sort(arr, size, 0, n - 1);
    free(size);
}
```

Método quick Sort:

Este método funciona escolhendo um "pivo" e dividindo o vetor em dois sub-vetores com base na posição do pivô. Os elementos menores que o pivô são colocados em um subvetor e os elementos maiores que o pivô são colocados em outro subvetor. Este processo é repetido recursivamente até que o vetor esteja completamente ordenado. Análise assintótica: $O(n \log n)$;



```
#include "Quick_sort.h"
#include <stdlib.h>

void troca(int vet[], int i, int j){
    int aux = vet[i];
    vet[i] = vet[j];
    vet[j] = aux;
}

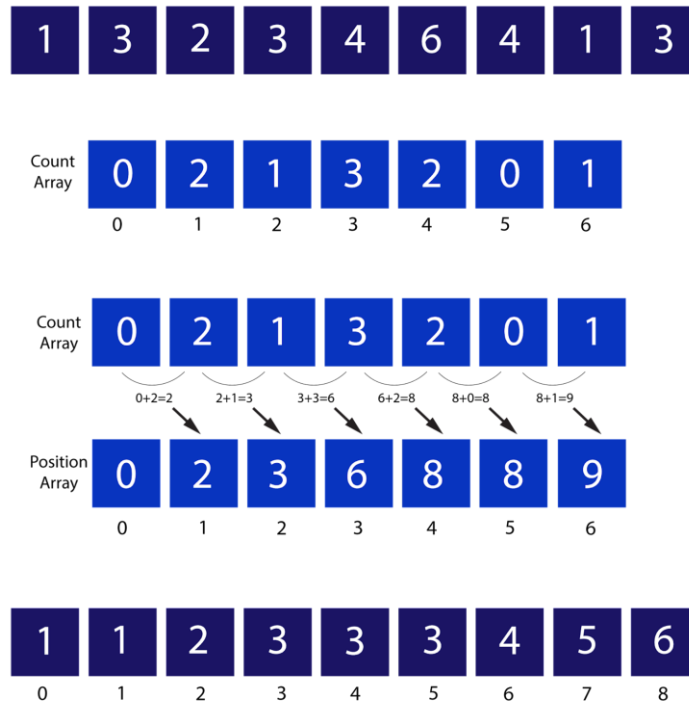
int particiona(int* vet, int inicio, int fim){
    int pivo, pivo_indice, i;
    pivo = vet[fim];
    pivo_indice = inicio;
    for(i = inicio; i < fim; i++){
        if(vet[i] <= pivo){
            troca(vet, i, pivo_indice);
            pivo_indice++;
        }
    }
    troca(vet, pivo_indice, fim);
    return pivo_indice;
}

int particiona_random(int* vet, int inicio, int fim){
    int pivo_indice = (rand() % (fim - inicio + 1)) + inicio;
    troca(vet, pivo_indice, fim);
    return particiona(vet, inicio, fim);
}

void quick_sort(int* vet, int inicio, int fim){
    if(inicio < fim){
        int pivo_indice = particiona_random(vet, inicio, fim);
        quick_sort(vet, inicio, pivo_indice - 1);
        quick_sort(vet, pivo_indice + 1, fim);
    }
}
```

Método Counting Sort:

Este é um método de ordenação não-comparativo que funciona contando a ocorrência de cada valor no vetor e usando essas contagens para determinar a posição final de cada valor no vetor ordenado. Análise assintótica: $O(n + k)$, onde k é o intervalo de valores no vetor;



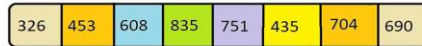
```
#include "Counting_sort.h"
#define MAX_VECTOR_SIZE 100000000

void counting_sort(int* arr, int size) {
    int i, j;
    int count[MAX_VECTOR_SIZE] = {0};
    for (i = 0; i < size; i++) {
        count[arr[i]]++;
    }
    int k = 0;
    for (i = 0; i < MAX_VECTOR_SIZE; i++) {
        for (j = 0; j < count[i]; j++) {
            arr[k++] = i;
        }
    }
}
```

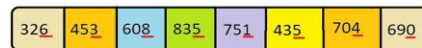
Método Radix Sort:

Este é outro método de ordenação não-comparativo que funciona classificando os elementos do vetor baseado em dígitos individuais (como o dígito das unidades, das dezenas, etc.). Análise assintótica: $O(nk)$, onde k é o número de dígitos nos valores do vetor

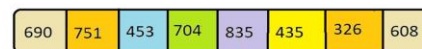
Let's Say The Given Array Is This :-



First, Consider The One's Place :-

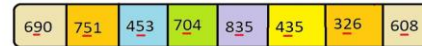


Now Sort the above array on the basis of digits on one's place

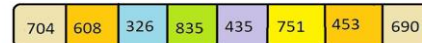


Observe That 835 has before 90 this is because it appeared before in the original array.

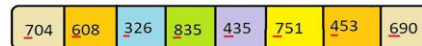
Now Consider the 10's Place :-



Now Sort the above array on the basis of digits on 10 's place



Now Consider the 100's Place :-



Now Sort the above array on the basis of digits on 100's place



Array Is Now Sorted

```
#include "Radix_sort.h"

int getMax(int array[], int size) {
    int max = array[0];
    for (int i = 1; i < size; i++)
        if (array[i] > max)
            max = array[i];
    return max;
}

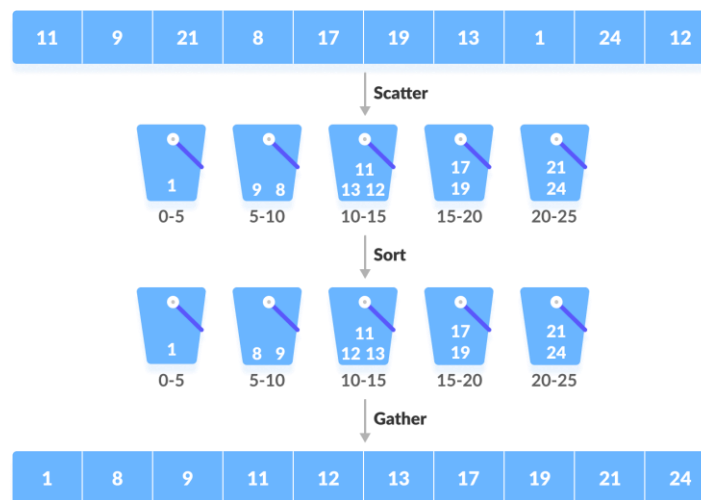
void countingSort(int array[], int size, int place) {
    int output[size + 1];
    int max = (array[0] / place) % 10;
    for (int i = 1; i < size; i++) {
        if ((array[i] / place) % 10 > max)
            max = array[i];
    }
    int count[max + 1];
    for (int i = 0; i < max; ++i)
        count[i] = 0;
    for (int i = 0; i < size; i++)
        count[(array[i] / place) % 10]++;
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (int i = size - 1; i >= 0; i--) {
        output[count[(array[i] / place) % 10] - 1] = array[i];
        count[(array[i] / place) % 10]--;
    }

    for (int i = 0; i < size; i++)
        array[i] = output[i];
}

void radix_sort(int* array, int size) {
    int max = getMax(array, size);
    for (int place = 1; max / place > 0; place *= 10)
        countingSort(array, size, place);
}
```


Método Bucket Sort:

Este é um método de ordenação não-comparativo que funciona distribuindo os elementos do vetor em "buckets" (ou recipientes) baseado em um valor de "bucket number" para cada elemento. Os elementos são, então, ordenados dentro de cada bucket e finalmente concatenados para formar o vetor ordenado. Análise assintótica: $O(n + k)$, onde k é o número de buckets;

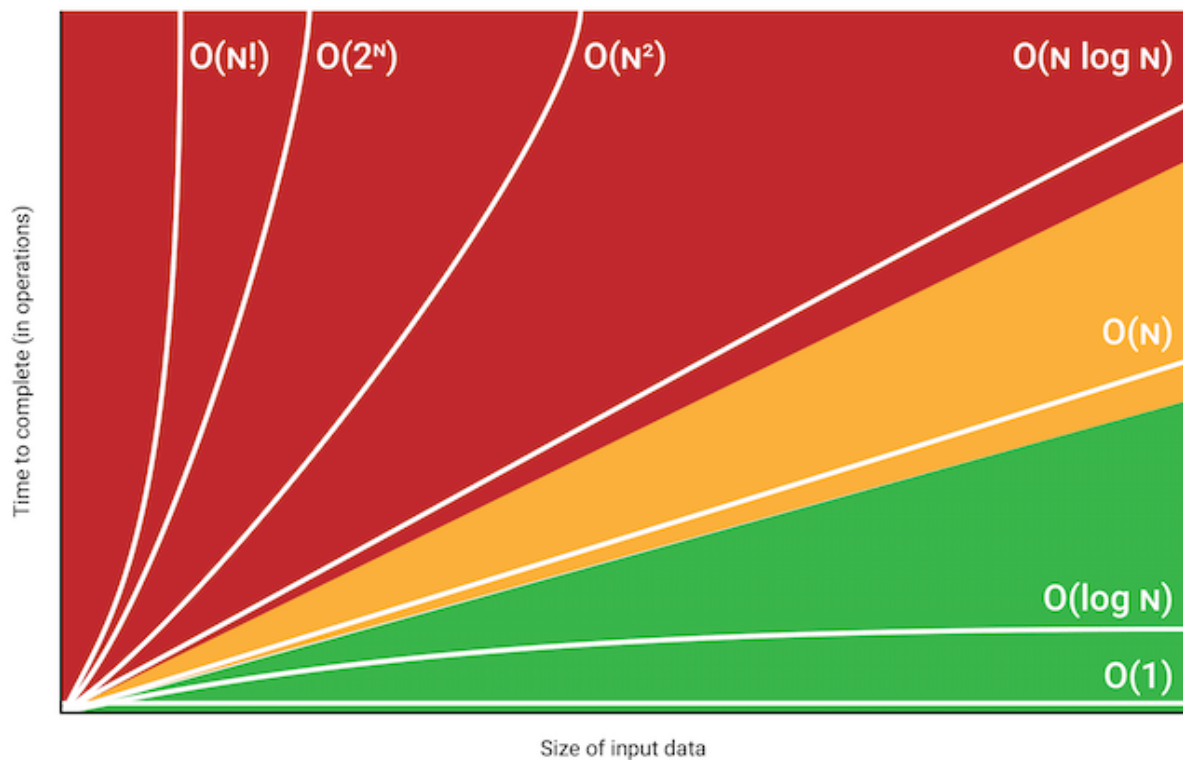


```
#include "Bucket_sort.h"
#include <stdlib.h>
#define NUM_BUCKETS 10

int getMaxi(int a[], int n)
{
    int max = a[0];
    for (int i = 1; i < n; i++)
        if (a[i] > max)
            max = a[i];
    return max;
}

void bucket_sort(int a[], int n)
{
    int max = getMaxi(a, n);
    int bucket[max], i;
    for (int i = 0; i <= max; i++)
    {
        bucket[i] = 0;
    }
    for (int i = 0; i < n; i++)
    {
        bucket[a[i]]++;
    }
    for (int i = 0, j = 0; i <= max; i++)
    {
        while (bucket[i] > 0)
        {
            a[j++] = i;
            bucket[i]--;
        }
    }
}
```

Algoritmo	Complexidade		
	Melhor	Médio	Pior
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$
Quick Sort	$O(n \cdot \log(n))$	$O(n \cdot \log(n))$	$O(n^2)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$




















ANÁLISE EMPÍRICA:

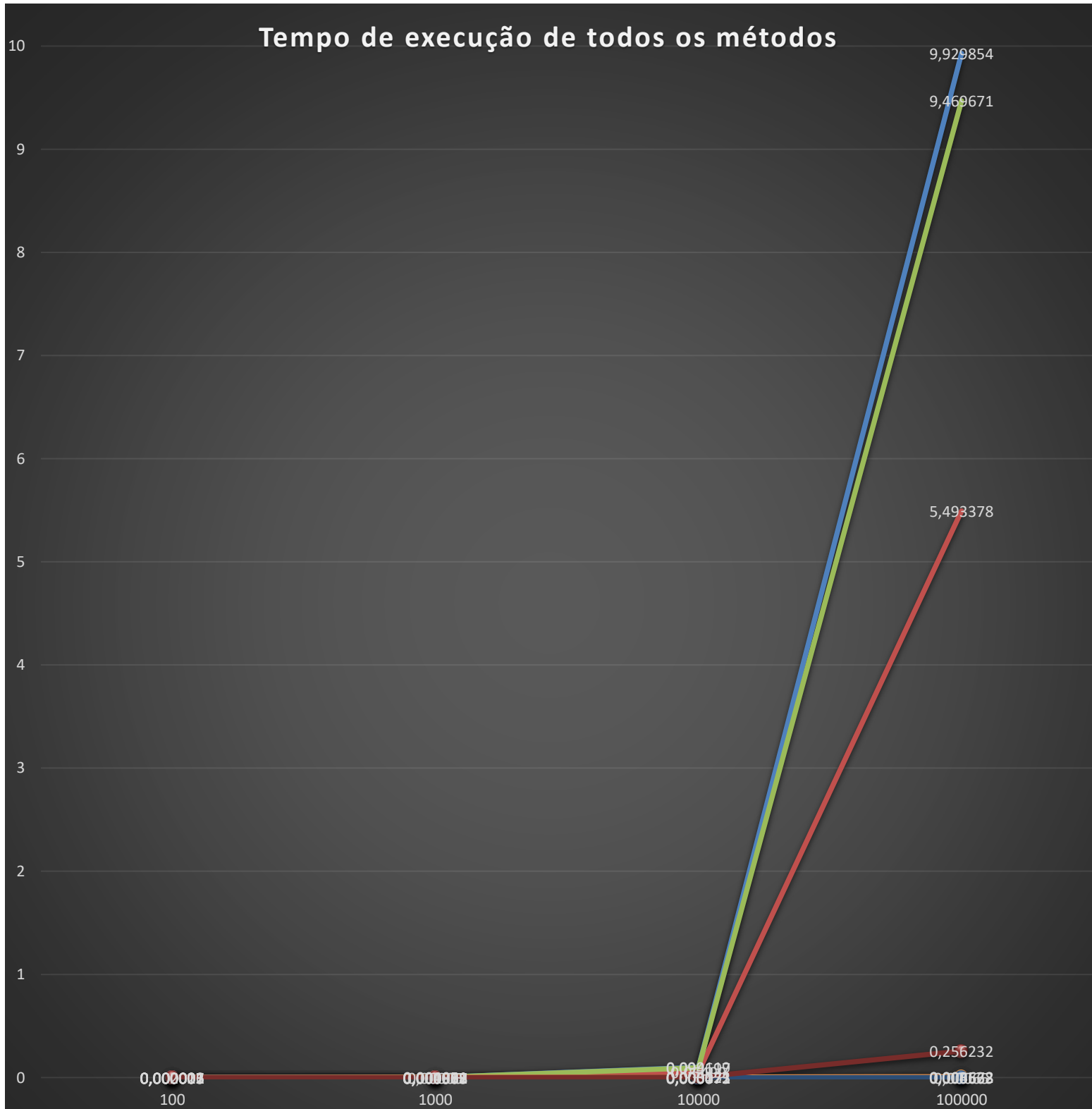
A convenção em linguagem C de todos os métodos de ordenação foram preparados em dois arquivos para implementar uma TAD;

Arquivo .H: protótipos das funções, tipos de ponteiro, e dados globalmente acessíveis. Aqui é definida a interface visível pelo usuário.

Arquivo .C: declaração do tipo de dados e implementação das suas funções. Aqui é ficará

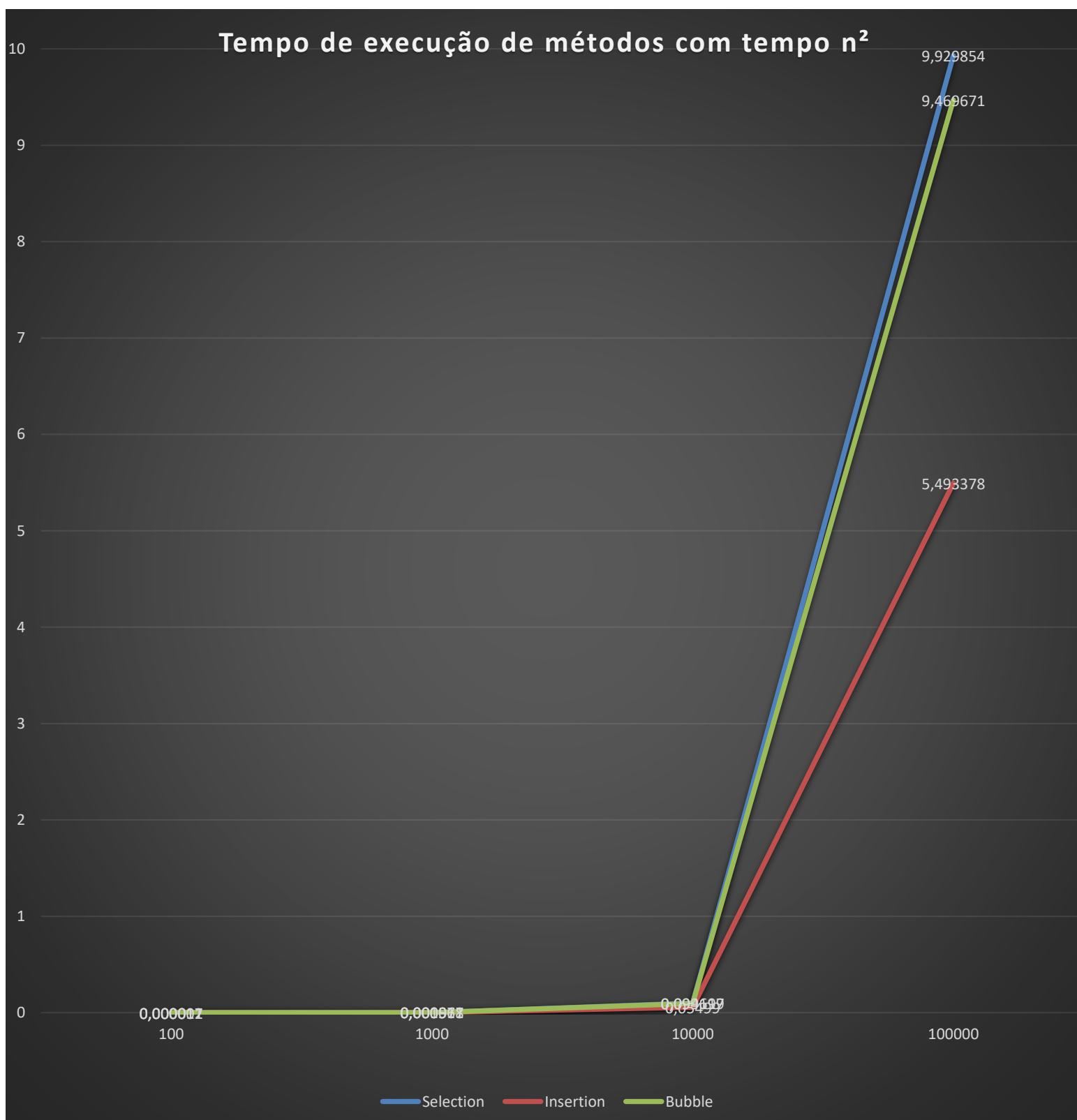
-  Bubble_sort.c
-  Bubble_sort.h
-  Bucket_sort.c
-  Bucket_sort.h
-  Counting_sort.c
-  Counting_sort.h
-  Insertion_sort.c
-  Insertion_sort.h
-  main.c
-  Merge_sort.c
-  Merge_sort.h
-  Quick_sort.c
-  Quick_sort.h
-  Radix_sort.c
-  Radix_sort.h
-  Selection_sort.c
-  Selection_sort.h

Tamanho	Selection	Insertion	Bubble	Merge	Radix	Counting	Bucket	Quick
100	0,000012	0,000007	0,000011	0,000001	0,000005	0,00202	0,000003	0,000006
1000	0,001017	0,000568	0,000971	0,000009	0,00005	0,00405	0,000012	0,000088
10000	0,099119	0,05499	0,094697	0,000071	0,000475	0,00612	0,000072	0,003153
100000	9,929854	5,493378	9,469671	0,000663	0,004672	0,00854	0,00065	0,256232



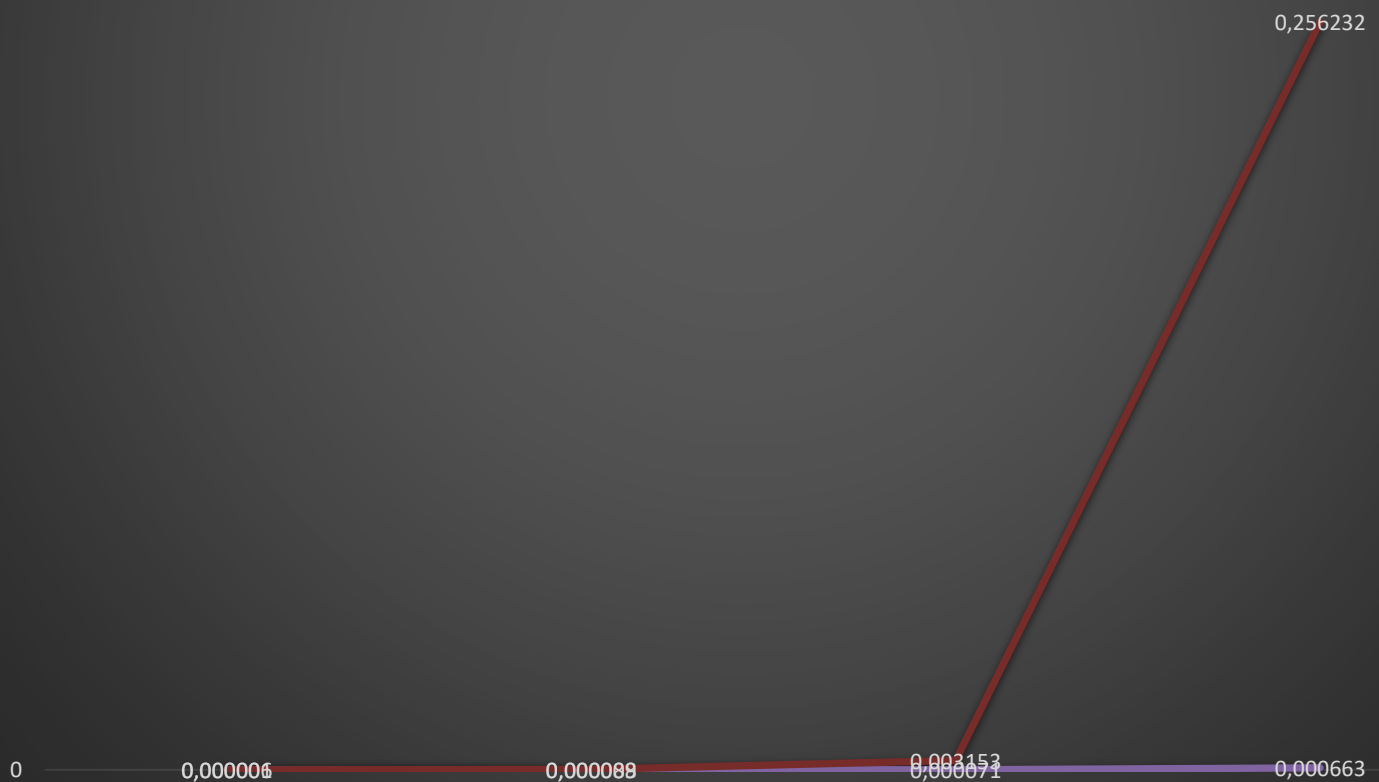
Tamanho	Selection	Insertion	Bubble
100	0,000012	0,000007	0,000011
1000	0,001017	0,000568	0,000971
10000	0,099119	0,05499	0,094697
100000	9,929854	5,493378	9,469671

Tempo de execução de métodos com tempo n^2



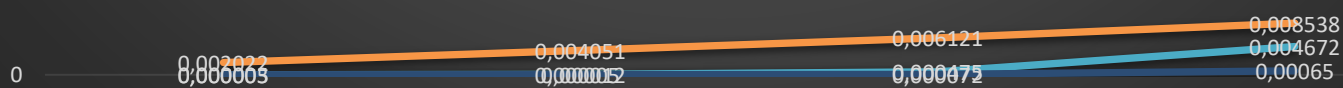
Tamanho	Merge	Quick
100	0,000001	0,000006
1000	0,000009	0,000088
10000	0,000071	0,003153
100000	0,000663	0,256232

Tempo de execução de métodos com tempo $n \cdot \log_2(n)$



Tamanho	Radix	Counting	Bucket
100	0,000005	0,00202	0,000003
1000	0,00005	0,00405	0,000012
10000	0,000475	0,00612	0,000072
100000	0,004672	0,00854	0,00065

Tempo de execução de métodos com tempo n



Especificações da máquina que rodou os testes:

Placa-mãe: PCWARE IPMH310G

Memória: 16gb 2333mhz

Processador: Intel® Core™ i5-9400F CPU @ 2.90GHz x 6

Placa de vídeo: AMD® Radeon rx 5500 xt