Relatório Projeto 3 V1.0 AED 2024/2025

Nome: Vasco Guilherme Alves Nº Estudante: 2022228207
PL (inscrição): 06 Email: vasco.guilherme.alves@gmail.com

IMPORTANTE:

- **Os textos das conclusões devem ser manuscritos...** o texto deve obedecer a este requisito para não ser penalizado.
- Texto para além das linhas reservadas, ou que não seja legível para um leitor comum, não será tido em conta.
- **O relatório deve ser submetido num único PDF** que deve incluir os anexos. A não observância deste formato é penalizada.

1. Planeamento

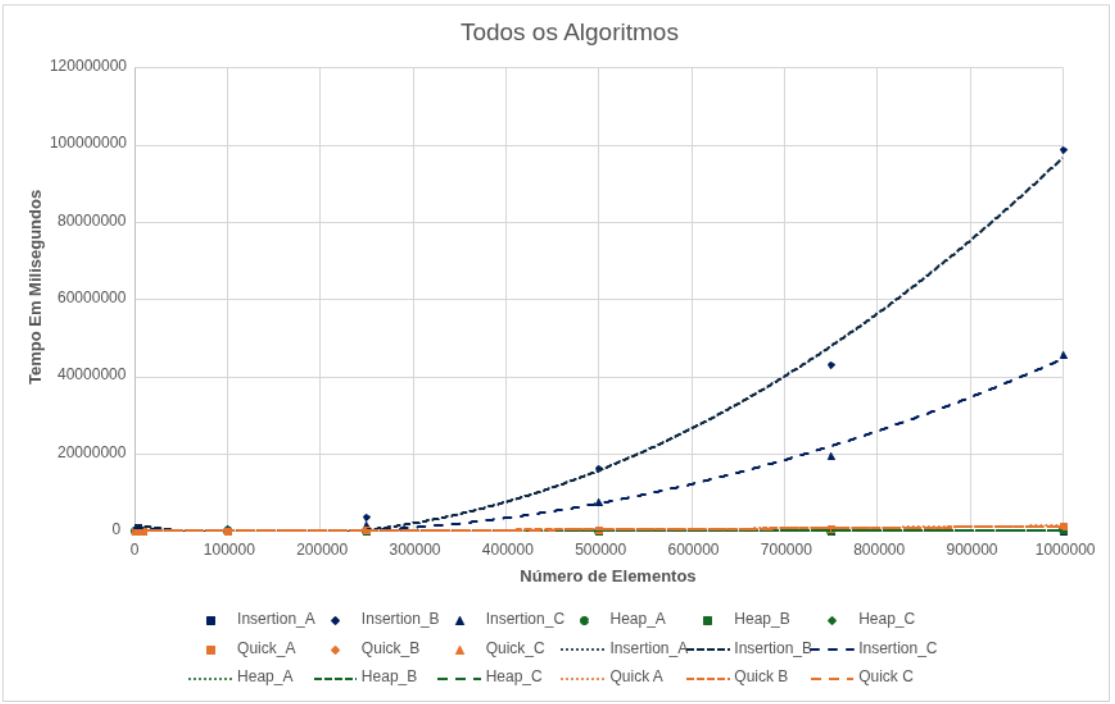
	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5	...
T1: Insertion Sort						
T2: Heap Sort						
T3: Quick Sort						
Finalização Relatório						

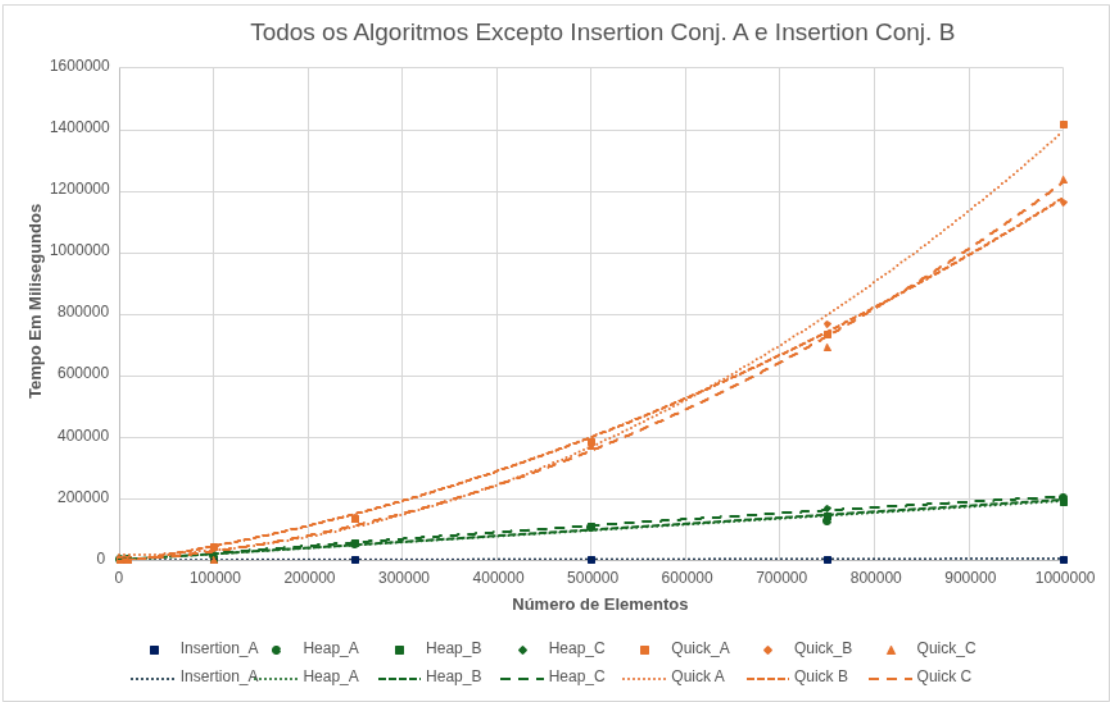
2. Recolha de Resultados *(tabelas)*

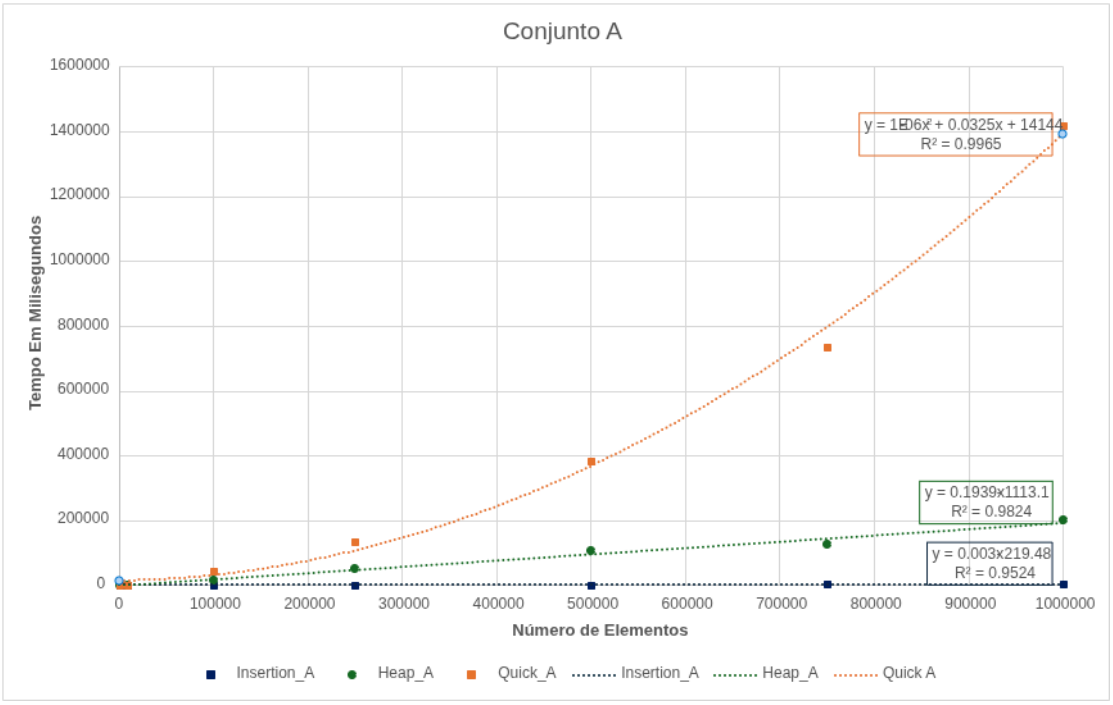
Keys	Insert. A	Insert. B	Insert. C	Heap A	Heap B	Heap C	Quick A	Quick B	Quick C
1000	0,002	0,063	0,035	0,101	0,094	0,101	0,003	0,067	0,032
10000	0,02	5,424	2,71	1,169	1,213	1,251	1,2	1,235	1,235
100000	0,302	519,586	274,338	13,775	14,23	16,249	44,972	37,245	37,76
250000	0,561	3,468,703	1,682,398	50,17	57,246	57,072	135,044	138,797	136,278
500000	1,297	16,193,054	7,474,055	107,026	111,256	107,588	388,43	385,595	370,121
750000	2,214	43,058,650	19,489,003	124,748	141,121	166,981	736,041	768,41	690,725
1000000	2,741	98,897,350	45,681,026	201,65	189,713	203,123	1,417,351	1,162,511	1,238,895

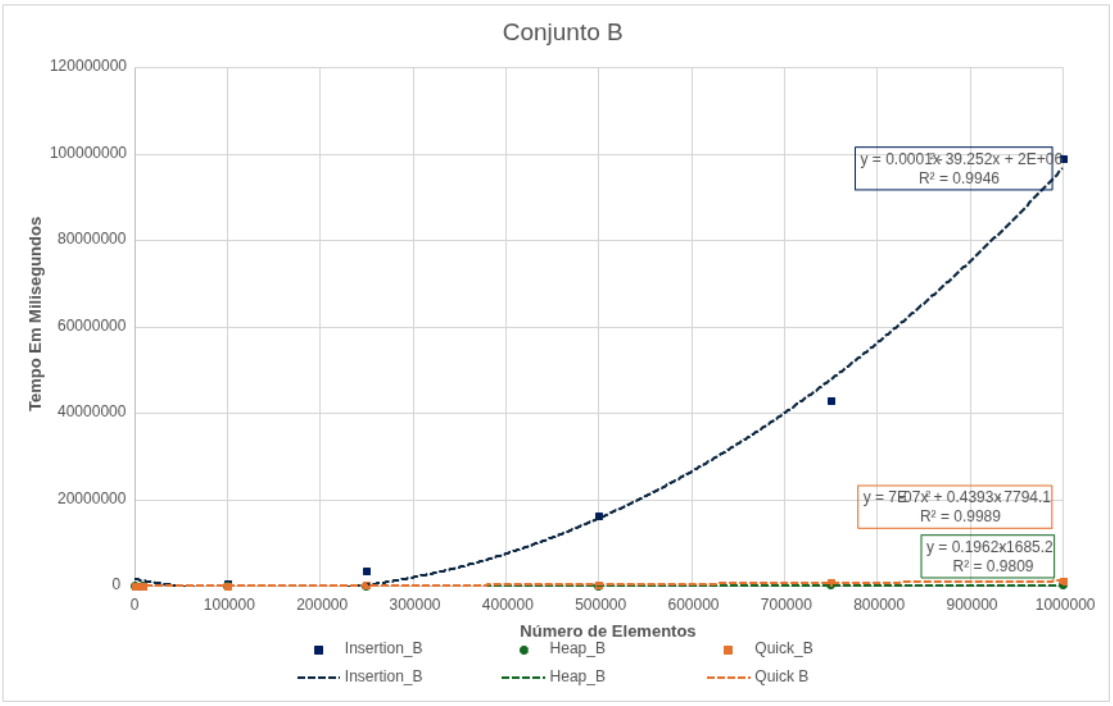
***Nota: Os tempos estão em nanosegundos e são números inteiros.**

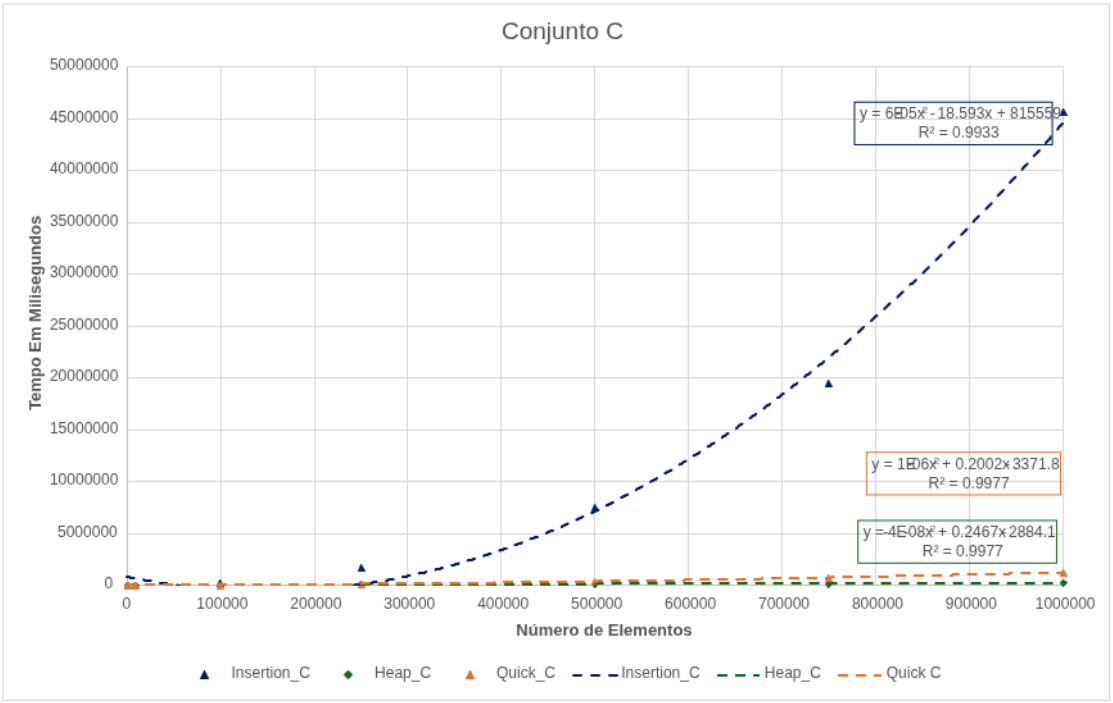
3. Visualização de Resultados *(gráficos)*

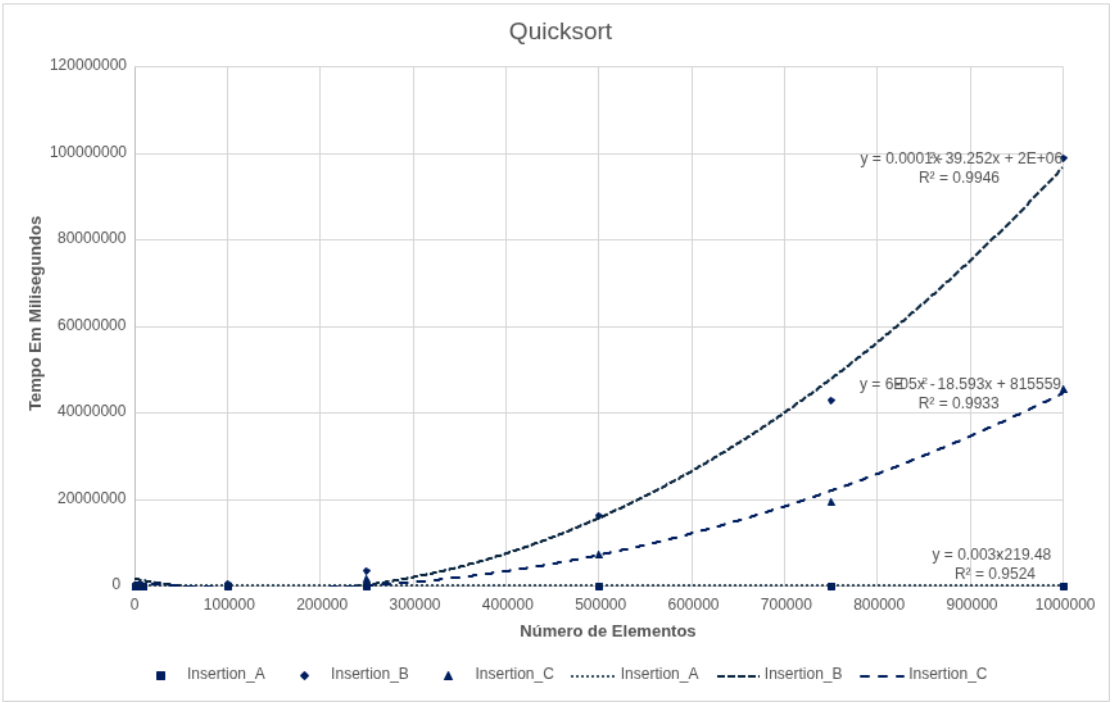


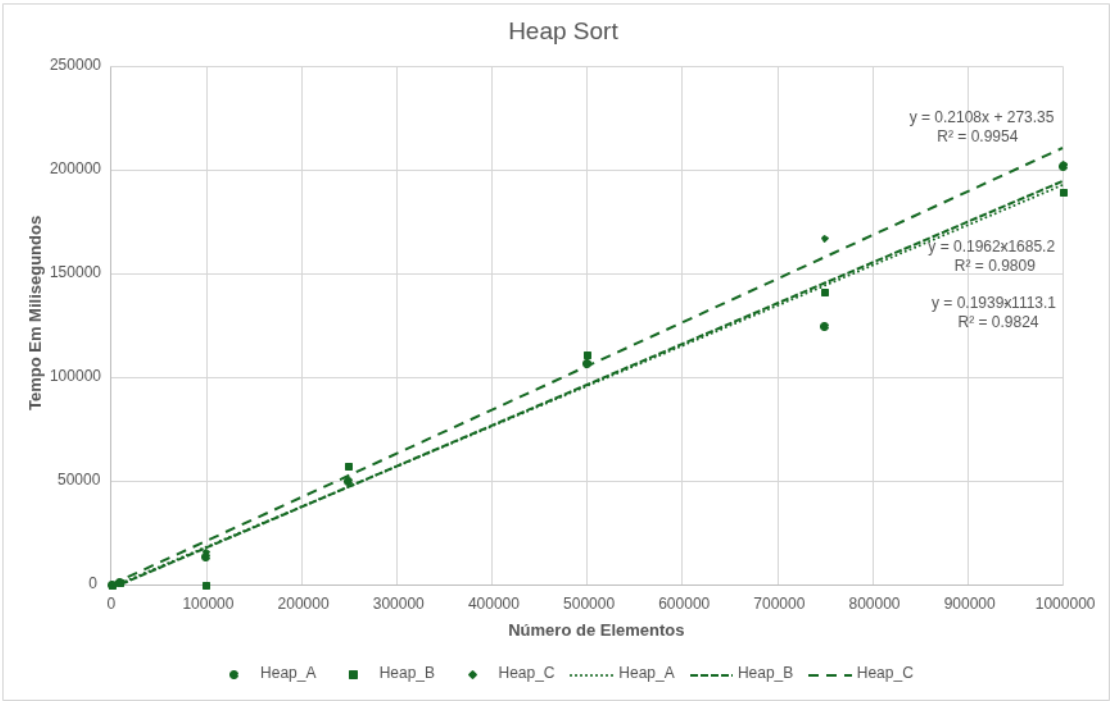


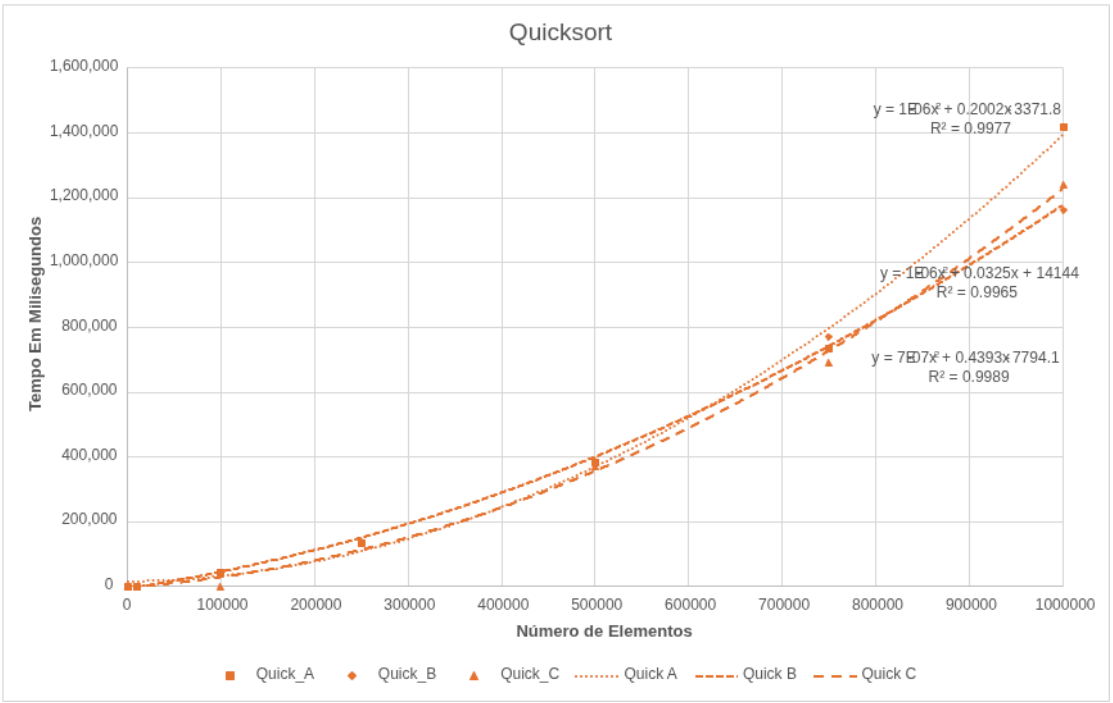

Conjunto A
Tempo Em Milisegundos
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000
Número de Elementos
y = 1E-06x² + 0.0325x + 14144
R² = 0.9965
y = 0.1939x1113.1
R² = 0.9824
y = 0.003x219.48
R² = 0.9524
Insertion_A
Heap_A
Quick_A
Insertion_A
Heap_A
Quick A


Conjunto B
Tempo Em Milisegundos
120000000
100000000
80000000
60000000
40000000
20000000
0
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
1000000
Número de Elementos
y = 0.0001x² - 39.252x + 2E+06
R² = 0.9946
y = 7E-07x² + 0.4393x7794.1
R² = 0.9989
y = 0.1962x1685.2
R² = 0.9809
Insertion_B
Heap_B
Quick_B
Insertion_B
Heap_B
Quick B


Conjunto C
Tempo Em Milisegundos
Número de Elementos
y = 6E-05x² - 18.593x + 815559
R² = 0.9933
y = 1E-06x² + 0.2002x - 3371.8
R² = 0.9977
y = -4E-08x² + 0.2467x - 2884.1
R² = 0.9977
Insertion_C
Heap_C
Quick_C
Insertion_C
Heap_C
Quick C


Quicksort
Tempo Em Milisegundos
Número de Elementos
y = 0.0001x² - 39.252x + 2E+06
R² = 0.9946
y = 6E-05x² - 18.593x + 815559
R² = 0.9933
y = 0.003x² - 219.48
R² = 0.9524
Insertion_A
Insertion_B
Insertion_C
Insertion_A
Insertion_B
Insertion_C


Heap Sort
Tempo Em Milisegundos
Número de Elementos
y = 0.2108x + 273.35
R² = 0.9954
y = 0.1962x1685.2
R² = 0.9809
y = 0.1939x1113.1
R² = 0.9824
Heap_A
Heap_B
Heap_C
Heap_A
Heap_B
Heap_C


Quicksort
Tempo Em Milisegundos
Número de Elementos
y = 1E-06x² + 0.2002x 3371.8
R² = 0.9977
y = -1E-06x² + 0.0325x + 14144
R² = 0.9965
y = 7E-07x² + 0.4393x 7794.1
R² = 0.9989
Quick_A
Quick_B
Quick_C
Quick A
Quick B
Quick C

4. Conclusões *(as linhas no template representam a extensão máxima de texto manuscrito)*

4.1 Tarefa 1

O algoritmo Insertion sort, no pior caso, tem complexidade $O(n^2)$, o que é comprovado pela regressão nos gráficos anteriores.

Para o conjunto A, observamos melhores resultados que todos os outros algoritmos, pois já está parcialmente ordenado, logo obtemos o melhor caso de $O(n)$.

Para os conjuntos restantes obtive a pior performance devido a sua complexidade média de $O(n^2)$.

4.2 Tarefa 2

O algoritmo Heapsort, tem complexidade $O(n \log n)$ pois cria um heap em $O(n)$ e faz o sort em $O(\log n)$.

A complexidade é $O(n \log n)$ no pior caso e $O(n)$ no pior caso, logo tem a melhor performance em geral dentos os algoritmos.

Como implementação quicksort mais avançada poderia ser mais rápida em prática, mas neste caso o heapsort foi o algoritmo mais rápido.

4.3 Tarefa 3

O algoritmo quicksort é sempre mais rápido que o Insertion sort pois foi optimizado para utilizalo para partições pequenas. A complexidade média é $n \log n$

O quicksort, ~~tem~~ mesmo com as 4 otimizações ~~tipo~~ desejadas, degenera para valores elevados de N.

O best-case é $O(n)$ devido à implementação do Dutch National Flag.

Anexo B - Código de Autor

```
/*
 * Hardware Original: TOSHIBA SATELLITE_C50-A PSCG6P-01YAR1,
 * CPU: Intel i5-3320M (4) @ 3.300GHz,
 * GPU: Intel 3rd Gen Core processor Graphics Controller
 * RAM: 7821MiB, SSD SATA3 1TB
 *
 * Aluno: Vasco Alves, 2022228207
*/

/* algoritmos/heapsort.go */
package algoritmos

/* Helper */
func heapify(arr []int, n int, i int) {

	/* propriedade de heap
	 arvore binaria implicita  */
	largest := i
	left := 2*i + 1
	right := 2*i + 2

	if left < n && arr[left] > arr[largest] {
		largest = left
	}
	if right < n && arr[right] > arr[largest] {
		largest = right
	}

	if largest != i {
		arr[i], arr[largest] = arr[largest], arr[i]
		heapify(arr, n, largest)
	}
}

func HeapSort(arr []int) {
	n := len(arr)

	// Build max heap
	for i := n/2 - 1; i >= 0; i-- {
		heapify(arr, n, i)
	}

	// One by one extract elements
	for i := n - 1; i > 0; i-- {
		arr[0], arr[i] = arr[i], arr[0]
		heapify(arr, i, 0)
	}
}
```

```
/* algoritmos/insertion.go */
package algoritmos

func InsertionSortN(arr []int, start int, end int) {
	/* clamp end idx */
	if end >= len(arr)-1 {
		end = len(arr)-1
	}
	/* clamp start idx */
	if start < 0 {
		start = 0
	}

	for i := start+1; i <= end ; i++ {
		key := arr[i]

		j := i - 1
		for j >= 0 && arr[j] > key {
			arr[j+1] = arr[j]
			j--
		}

		arr[j+1] = key
	}
}

func InsertionSort(arr []int) {
	InsertionSortN(arr, 0, len(arr)-1)
}

/* algoritmos/quicksort.go */
package algoritmos

var (
	threshhold int = 1000
)

func swap(arr []int, i int, j int) {
	arr[i], arr[j] = arr[j], arr[i]
}

func partition(arr []int, low int, high int) (int,int) {

	/* Optimização 3: mediana de 3 */
    mid := low + (high-low)/2
    if arr[low] > arr[mid] {
        swap(arr, low, mid)
    }
    if arr[low] > arr[high] {
        swap(arr, low, high)
    }
    if arr[mid] > arr[high] {
        swap(arr, mid, high)
    }
    swap(arr, mid, high)
    pivot := arr[high]

	/* Optimização 4: dutch national flag */
	i := low      // primeiro elmento
	j := low      // elemento atual
	k := high - 1 // ultimo elemento
```

```
	for j <= k {
		if arr[j] < pivot {
			swap(arr, i, j)
			i++
			j++
		} else if arr[j] > pivot {
			swap(arr, j, k)
			k--
		} else {
			j++
		}
	}

	swap(arr, j, high)
	return i, j
}


func quicksort(arr []int, low int, high int) {

	if low >= high || low < 0 {
		return
	}

	/* Optimzação 1: Insertion sort para len < threshhold */
	if ( (1+high-low) <= threshhold) {
		InsertionSortN(arr, low, high)
		return
	}

	/* Optimiazação 2 */
	left, right := partition(arr, low, high)
	quicksort(arr, low, left-1)   // Sort elements less than pivot
	quicksort(arr, right+1, high) // Sort elements greater than pivot
}

func QuickSort(arr []int) {
	quicksort(arr, 0, len(arr)-1)
}

/* ./main.go */
package main

import (
	"log"
	"math/rand"
	"prj3/algoritmos"
	"sort"
	"time"
	"os"
	"io"
)

type ConjuntoGen struct {
	nome string
	fn	func(int)[]int
}

type Algoritmo struct {
	nome string
	fn	func([]int)
}
```

```
var (
	media int = 10
	tamanhos = []int{1000, 10000, 100000, 250000, 500000, 750000, 1000000}
	conjuntos = []ConjuntoGen{
		{"Conjunto A", ConjuntoA},
		{"Conjunto B", ConjuntoB},
		{"Conjunto C", ConjuntoC},
	}
	algos = []Algoritmo{
		{"Insertion Sort", algoritmos.InsertionSort},
		{"Heap Sort", algoritmos.HeapSort},
		{"Quicksort", algoritmos.QuickSort},
	}
)

func isSorted(arr []int) (bool) {
	i := 0
	for arr[i] <= arr[i+1] {
		i++
		if i == len(arr)-1 {
			return true
		}
	}
	return false
}

func gerarConjuntoRepetido(size int, repetir float64) ([]int) {
	if repetir > 1 || repetir < 0 {
		panic("[gerarConjuntoRepetido] repetir deve ser uma percentagem ente
0 e 1 válida")
	}

	base := make([]int, 0, size)
	numUnicos := int(float64(size) * repetir)
	numRepetidos := size - numUnicos

	for i := 0; i < numUnicos; i++ {
		base = append(base, i)
	}
	// Adicionar repetições aleatórias
	for i := 0; i < numRepetidos; i++ {
		valor := base[rand.Intn(len(base))]
		base = append(base, valor)
	}

	return base
}

func ConjuntoA(size int) ([]int) {
	arr := gerarConjuntoRepetido(size, 0.95)
	sort.Ints(arr)
	return arr
}

func ConjuntoB(size int) ([]int) {
	arr := gerarConjuntoRepetido(size, 0.95)
	sort.Sort(sort.Reverse(sort.IntSlice(arr)))
	return arr
}

func ConjuntoC(size int) ([]int) {
	arr := gerarConjuntoRepetido(size, 0.95)
```

```
	rand.Shuffle(len(arr), func(i, j int) {
		arr[i], arr[j] = arr[j], arr[i]
	})
	return arr
}

func TestAlgorithm(alg func([]int), conjGen func(int) ([]int), size int, attempts
int) (time.Duration) {

	totalTime := time.Duration(0)

	conj := conjGen(size)

	for i := 0; i < attempts; i++ {
		arr := make([]int, len(conj))
		copy(arr, conj)

		start := time.Now()
		alg(conj)
		if (!isSorted(conj)) {
			panic("Conjunto não ordenado?")
		}
		totalTime += time.Since(start)

		// log.Print(conj)
	}

	return totalTime / time.Duration(attempts)
}

func main() {

	logFile, err := os.OpenFile("log.txt", os.O_CREATE | os.O_APPEND |
os.O_RDWR, 0666)
	if err != nil {
		panic(err)
	}
	mw := io.MultiWriter(os.Stdout, logFile)
	log.SetOutput(mw)


	for _, tamanho := range tamanhos {
		for _, algoritmo := range algos {
			for _, conjunto := range conjuntos {
				res := TestAlgorithm(algoritmo.fn, conjunto.fn,
tamanho, media)

log.Printf("[RESULTADO]\t%s\t%s\tSIZE=%d\tAVG=%d\tMédia Final = %.3fms\n",
					algoritmo.nome, conjunto.nome, tamanho, media,
float64(res)/float64(time.Millisecond))
			}
		}
	}

	log.Print("Done!\n")
}


/* Fim do ficheiro */
```