Aluno: Félix Luiz Garção Filho

Matrícula: 20230024216

Algoritmos de ordenação

1.

- SelectionSort:

```
func SelectionSortIP(v []int) {
	for varredura := 0; varredura < len(v)-1; varredura++ { // varredura
		menor := varredura
		for j := varredura + 1; j < len(v); j++ { // comparar o menor com o resto do
			if v[j] < v[menor] {
				menor = j
			}
		}
		v[varredura], v[menor] = v[menor], v[varredura]

	}

}
```

- BubbleSort:

```
func BubbleSort(v []int) {
	trocou := false
	n := len(v)
	for varredura := 0; varredura < n-1; varredura++ {
		for j := 0; j < (n - varredura - 1); j++ { // Dica 2: sempre vai diminuindo e
			if v[j] > v[j+1] {
				v[j], v[j+1] = v[j+1], v[j]
				trocou = true
			} else {
				trocou = false
			}
		}
		if !trocou {
			return
		}
	}
}
```

- InsertionSort:

```
func InsertionSort(v []int) {
	for varredura := 1; varredura < len(v); varredura++ {
		for i := varredura; i > 0; i-- {
			if v[i-1] > v[i] {
				v[i], v[i-1] = v[i-1], v[i]
			} else {
				break
			}
		}
	}
}
```

- MergeSort:

```
func merge(v []int, e []int, d []int) { //funcao auxiliar para juntar os dois vetores
	iv := 0
	ie := 0
	id := 0
	for ie < len(e) && id < len(d) {
		if e[ie] < d[id] {
			v[iv] = e[ie]
			ie++
		} else {
			v[iv] = d[id]
			id++
		}
		iv++
	}
	for ie < len(e) {
		v[iv] = e[ie]
		ie++
		iv++
	}
	for id < len(d) {
		v[iv] = d[id]
		id++
		iv++
	}
}


func MergeSort(v []int) {
	if len(v) > 1 {
		mid := len(v)/2
		e := make([]int,mid)
		d := make([]int,len(v)-mid)

		iv := 0
		for ie:=0; ie < len(e); ie++{
			e[ie] = v[iv]
			iv++
		}

		for id:=0; id < len(d); id++{
			d[id] = v[iv]
			iv++
		}
```

```
                MergeSort(e)
                MergeSort(d)
                merge(v,e,d)
        }
}
```

- QuickSort:

```
func Partition(v [] int, ini int, fim int) int {
    pivot := v[fim]
    p_index := ini
    for i := ini; i < fim; i++ {
        if v[i] <= pivot {
            v[p_index], v[i] = v[i], v[p_index]
            p_index++
        }
        v[p_index], v[i] = v[i], v[p_index]
    }
    v[fim],v[p_index] = v[p_index],v[fim]
    return p_index
}

func QuickSort(v [] int, ini int, fim int) {
    if ini < fim {
        index_pivot := Partition(v,ini,fim)
        QuickSort(v,ini,index_pivot-1)
        QuickSort(v,index_pivot+1,fim)
    }
}
```

- CountingSort:

```
func CountingSort(v []int) []int{
	maior, menor := v[0],v[0]
	// achar o maior e o menor
	for i:=1; i<len(v); i++{
		if v[i] < menor {
			menor = v[i]
		}
		if v[i] > maior {
			maior = v[i]
		}
	}
	// vetor de contagem
	c := make([]int,maior-menor+1)
	for i:=0; i < len(v); i++ { // contar as ocorrencias de cada valor e atualizar o vetor
		c[v[i]-menor]++
	}

	for i :=0; i<len(c); i++{
		c[i+1] += c[i]
	}

	ord := make([]int, len(v))

	for i:=0; i<len(ord); i++{
		ord[c[v[i]-menor]-1] = v[i]
	}

	return ord
}
```

2.

- SelectionSort: o pior caso do algoritmo é O(n²), pois o algoritmo faz uma varredura no vetor e dentro dessa varredura, ele elege o indice "[varredura]" como o menor valor do vetor. e entao vai comparando ele com cada elemento lá dentro e caso haja um menor que v[varredura], ele troca o menor para este indice J e ao final de cada uma dessas comparações internas, ele permuta o v[varredura] e o v[menor] de posição. inclusive, mesmo se o vetor estiver ordenado ele fará essas comparações que não seriam necessarias. portanto o seu pior caso tambem sera o melhor caso.

- Bubblesort: o pior caso do algoritmo é O(n²) (quando o vetor estiver completamente desordenado), pois, similarmente ao SelectionSort, se o vetor estiver desordenado, ele terá um comportamento que fará permutas durante o for interno (ele vai comparar o elemento atual com o próximo e caso o atual seja maior que o próximo, será realizada a permuta de posições entre eles)
- InsertionSort: o pior caso do algoritmo também é O(n²) (quando o vetor estiver ordenado decrescentemente), para cada novo elemento i que pegamos da parte não ordenada, temos que compará-lo com todos os elementos i-1 que já estavam na parte ordenada. Além disso, teremos que deslocar todos esses i-1 elementos.
- MergeSort: o pior caso do mergesort, tambem é seu melhor caso O(n log_n), pois Ele sempre divide o vetor em O$(\log n)$ níveis, e em cada um desses níveis, ele sempre faz um trabalho total de O(n) para intercalar todos os sub-vetores daquele nível (conquista). juntando a divisão e a conquista temos O(n log_n).
- CountingSort: o pior caso do couting é O(n + k), ele ocorre quando o valor máximo do vetor (k) é muito maior que a quantidade de elementos (n) [pois ele irá alocar uma area com o tamanho n+k]. Assim, a complexidade O(n+k) se aproxima de O(k), tornando o algoritmo lento e faminto por memória.

Algoritmo	Pior Caso	Melhor Caso
SelectionSort	O(n²)	Ω(n²)
BubbleSort	O(n²)	Ω(n)
InsertionSort	O(n²)	Ω(n)
MergeSort	O(n log_n)	Ω(n log_n)
CountingSort	O(n + k)	Ω(n + k)

3.

selectionsort:

Passo	Menor elemento encontrado	Troca realizada	Resultado parcial
1	1	1 ↔ 3	[1,6,2,5,4,3,7,3,10⁹]
2	2	2 ↔ 6	[1,2,6,5,4,3,7,3,10⁹]
3	3	3 ↔ 6	[1,2,3,5,4,6,7,3,10⁹]

Passo	Menor elemento encontrado	Troca realizada	Resultado parcial
4	3	3 ↔ 5	[1,2,3,3,4,6,7,5,10^9]
5	4	—	[1,2,3,3,4,6,7,5,10^9]
6	5	5 ↔ 6	[1,2,3,3,4,5,7,6,10^9]
7	6	6 ↔ 7	[1,2,3,3,4,5,6,7,10^9]
Resultado	—	—	[1,2,3,3,4,5,6,7,10^9]

bubblesort:

Passagem	Trocas realizadas	Vetor após passagem
1	7 trocas	[3,2,5,4,3,6,1,7]
2	6 trocas	[2,3,4,3,5,1,6,7]
3	5 trocas	[2,3,3,4,1,5,6,7]
4	4 trocas	[2,3,3,1,4,5,6,7]
5	3 trocas	[2,3,1,3,4,5,6,7]
6	2 trocas	[2,1,3,3,4,5,6,7]
7	1 troca	[1,2,3,3,4,5,6,7]
Resultado	—	[1,2,3,3,4,5,6,7]

insertionsort:

Passo	Subvetor já ordenado	Elemento inserido	Resultado parcial
1	[3]	6	[3,6,2,5,4,3,7,1]
2	[3,6]	2	[2,3,6,5,4,3,7,1]
3	[2,3,6]	5	[2,3,5,6,4,3,7,1]
4	[2,3,5,6]	4	[2,3,4,5,6,3,7,1]
5	[2,3,4,5,6]	3	[2,3,3,4,5,6,7,1]

Passo	Subvetor já ordenado	Elemento inserido	Resultado parcial
6	[2,3,3,4,5,6]	7	[2,3,3,4,5,6,7,1]
7	[2,3,3,4,5,6,7]	1	[1,2,3,3,4,5,6,7]

mergesort:

Etapa	Ação	Resultado parcial
1	Divide [3,6,2,5,4,3,7,1] em duas metades	[3,6,2,5], [4,3,7,1]
2	Divide até sobrar 1 elemento	[3],[6],[2],[5],[4],[3],[7],[1]
3	Intercala pares	[3,6], [2,5], [3,4], [1,7]
4	Intercala metades	[2,3,5,6], [1,3,4,7]
5	Intercala final	[1,2,3,3,4,5,6,7]

quicksort(sem pivo aleatorio):

Etapa	Subvetor	Pivô	Situação
1	[1,2,3,3,4,5,6,7]	7	Tudo menor
2	[1,2,3,3,4,5,6]	6	Tudo menor
3	[1,2,3,3,4,5]	5	Tudo menor
4	[1,2,3,3,4]	4	Tudo menor
5	[1,2,3,3]	3	Tudo menor ou igual
6	[1,2]	2	Tudo menor
Resultado	—	—	[1,2,3,3,4,5,6,7]

quicksort(com pivo aleatorio):

Etapa	Subvetor	Pivô (aleatório)	Resultado parcial
1	[1,2,3,3,4,5,6,7]	4	[1,2,3,3] + [4] + [5,6,7]
2	[1,2,3,3]	2	[1] + [2] + [3,3]

Etapa	Subvetor	Pivô (aleatório)	Resultado parcial
3	[5,6,7]	6	[5] + [6] + [7]
Resultado	—	—	[1,2,3,3,4,5,6,7]

coutingsort:

Passo	Descrição	Resultado
1	Contagem de ocorrências	[1,1,2,1,1,1,1]
2	Soma cumulativa	[1,2,4,5,6,7,8]
3	Preenchimento do vetor ordenado	[1,2,3,3,4,5,6,7]
Resultado final	—	[1,2,3,3,4,5,6,7]

4.

```
============================================================
A. Para vetores de tamanho PEQUENO (n=100)
============================================================
InsertionSort (O(n^2))   [Pequeno, Aleatório]     : 0s
MergeSort (O(n log n)) [Pequeno, Aleatório]       : 0s
CountingSort (O(n+k))  [Pequeno, Aleatório]       : 0s
---
InsertionSort (O(n^2))   [Pequeno, Ordenado]      : 0s
MergeSort (O(n log n)) [Pequeno, Ordenado]        : 0s
CountingSort (O(n+k))  [Pequeno, Ordenado]        : 0s
---
InsertionSort (O(n^2))   [Pequeno, Reverso]       : 0s
MergeSort (O(n log n)) [Pequeno, Reverso]         : 0s
CountingSort (O(n+k))  [Pequeno, Reverso]         : 0s

============================================================
B/C. Para vetores de tamanho GRANDE
     e Performance Estável do MergeSort (Ponto C)
============================================================
... Testando O(n^2) com n=50000 ...
InsertionSort (O(n^2))   [GRANDE, Aleatório]      : 635.5523ms
InsertionSort (O(n^2))   [GRANDE, Ordenado]       : 0s
InsertionSort (O(n^2))   [GRANDE, Reverso]        : 1.2538196s

... Testando O(n log n) e O(n+k) com n=200000 ...
MergeSort (O(n log n)) [GRANDE, Aleatório]        : 56.3757ms
MergeSort (O(n log n)) [GRANDE, Ordenado]         : 27.6086ms
MergeSort (O(n log n)) [GRANDE, Reverso]          : 24.2945ms
---
CountingSort (O(n+k))  [GRANDE, Aleatório]        : 41.7177ms
CountingSort (O(n+k))  [GRANDE, Ordenado]         : 20.0044ms
CountingSort (O(n+k))  [GRANDE, Reverso]          : 18.8403ms

============================================================
D. Pior Caso do QuickSort (n=200000)
============================================================
QuickSort NAIVE (Pivô Fixo) [GRANDE, Aleatório]   : 20.392ms
QuickSort NAIVE (Pivô Fixo) [GRANDE, Ordenado]    : 33.1296268s
QuickSort NAIVE (Pivô Fixo) [GRANDE, Reverso]     : 29.3271037s
---
```

```
QuickSort RANDOMIZADO         [GRANDE, Aleatório] : 22.095ms
QuickSort RANDOMIZADO         [GRANDE, Ordenado]  : 8.5348ms
QuickSort RANDOMIZADO         [GRANDE, Reverso]   : 16.9407ms

=========================================================
E. Desempenho do CountingSort (Bom vs. Mau Caso)
=========================================================
... Caso BOM: n=1000000 (grande), k=1000 (pequeno) ...
CountingSort (Bom Caso: n=1M, k=1k)               : 14.8294ms
MergeSort (Baseline: n=1M)                        : 169.2947ms

... Caso RUIM: n=200000 (médio), k=50000000 (GIGANTE) ...
CountingSort (Mau Caso: n=200k, k=50M)            : 413.823ms
MergeSort (Baseline: n=200k)                      : 36.8256ms
```

5.

a) ele não “enxerga” que o vetor já está ordenado, ele irá ficar iterando no vetor e fazendo comparações desnecessariamente.

b) pois há uma condição de parada no algoritmo que consegue dizer se o vetor está ordenado (if !trocou{break;})

c) no for dentro da varredura, ele é capaz de interrompê-lo caso os elementos atuais e anteriores estejam ordenados(caso v[i-1] < v[i]) ele interrompe a execução do laço.

d) ele vai quebrando o vetor até que hajam vários microvetores com tamanho igual a 1 e após isso ele vai mesclando os microvetores de forma ordenada até que resulte no vetor original(dividir para conquistar)

e) O pior caso do QuickSort ($O(n^2)$) ocorre quando o pivô divide o vetor de forma muito desigual. por exemplo, escolhendo sempre o primeiro elemento em um vetor já ordenado. Assim, uma partição fica vazia e a outra contém quase todos os elementos. Isso faz o algoritmo realizar n + (n−1) +

(n−2) + ... + 1 ≈ n²/2 comparações. Ou seja, a recursão fica profunda e o desempenho degrada.

f) basta randomizar o pivô

g) o CountingSort só é melhor que MergeSort e QuickSort quando os dados são inteiros e com intervalo pequeno.

6. e)

7. a)

8. a)

9. d)

10. d)

11. e)

Árvores de Busca Binária

1.

a) a principal diferença entre uma arvore binaria e uma bst é que a BST possui uma regra de ordenação dos nós (esquerda < nó < direita) e a busca é mais eficiente $Log_2(n)$

b) é a distância do nó até a raiz

c) Uma árvore de busca binária cuja estrutura é composta por todos os

níveis (exceto possivelmente o último) cheios.

d) se calcula fazendo a altura da subarvore esquerda - direita. se for == 0, ela é perfeitamente balanceada, caso contrario, ela vai ser desbalanceada

e) a cada nível terá $n = 2^{altura+1} - 1$, aplicando log e passando o 1 somandg: $log_2(n+1) = h + 1$, que entao terá: $h = log_2(n+1) - 1$. Portanto $h = O(log(n))$

f)

Operação	Árvore desbalanceada
Add()	O(n)
Search()	O(n)
Min()/Max()	O(n)
Percursos (Pre/In/Pos/Level)	O(n)
Height()	O(n)
Remove()	O(n)

g)

Operação	Complexidade de tempo (árvore completa)
Add(value)	O(log n)
Search(value)	O(log n)
Min()	O(log n)
Max()	O(log n)
PreNav()	O(n)
InNav()	O(n)
PosNav()	O(n)
LevelNav()	O(n)

Operação	Complexidade de tempo (árvore completa)
Height()	O(log n) *(ou O(1) se armazenada)*
Remove(value)	O(log n)

h) Casos:

- O nó a ser removido é uma folha (não tem filhos):
  - (Este é o caso mais simples. Basta remover o nó e fazer o pai dele apontar para nulo)
- O nó a ser removido possui um filho:
  - contorna-se o nó a ser removido, fazendo com que o pai dele aponte diretamente para o seu único filho
- O nó a ser removido possui dois filhos:
  - Precisa encontrar um sucessor (maior que todos na subárvore esquerda ou o menor da direita)
  - Copiar o valor: copiamos o valor do sucessor, para o nó que queremos remover
  - Remover o Sucessor: Precisamos remover o nó original(que está duplicado)
  - Sabemos que o nó R original é o sucessor, e ele sempre terá 0 ou 1 filho(nunca 2)(e isso nos leva pro primeiro caso ou pro segundo)

2.

```
func createNode(val int) *Node {
	return &Node{
		val: val,
	}
}

func (t *bst) add(val int) {
	if t.root == nil {
		t.root = createNode(val)
	} else {
		t.root.AddNode(val)
	}

	t.inserted++
}

func (no *Node) AddNode(val int) {
	if val <= no.val {
		if no.left == nil {
			no.left = createNode(val)
		} else {
			no.left.AddNode(val)
		}
	} else {
		if no.right == nil {
			no.right = createNode(val)
		} else {
			no.right.AddNode(val)
		}
	}
}

func (t *bst) search(val int) bool {
	if t.root == nil {
		return false
	} else {
		return t.root.searchNode(val)
	}
}
```

```
func (no *Node) searchNode(val int) bool {
	if val == no.val {
		return true
	} else if val < no.val {
		if no.left == nil {
			return false
		} else {
			return no.left.searchNode(val)
		}
	} else {
		if no.right == nil {
			return false
		} else {
			return no.right.searchNode(val)
		}
	}
}

func (t *bst) min() (int, error) {
	if t.root == nil {
		return -1, errors.New("bst vazia")
	} else {
		return t.root.min(), nil
	}
}

func (no *Node) min() int {
	val := no
	for val.left != nil {
		val = val.left
	}
	return val.val
}

func (t *bst) max() (int, error) {
	if t.root == nil {
		return -1, errors.New("bst vazia")
	}
	val := t.root
	for val.right != nil {
		val = val.right
	}
	return val.val, nil
```

```
}

func (t *bst) height() int {
	if t.root == nil {
		return 0
	} else {
		return t.root.height()
	}
}

func (no *Node) height() int {
	h_NodeLeft := 0

	if no.left == nil && no.right == nil {
		return 0
	}

	if no.left != nil {
		h_NodeLeft = 1 + no.left.height()
	}

	h_NodeRight := 0
	if no.right != nil {
		h_NodeRight = 1 + no.right.height()
	}

	if h_NodeLeft >= h_NodeRight {
		return h_NodeLeft
	} else {
		return h_NodeRight
	}
}

func (no *Node) preOrder() {
	fmt.Println(no.val)
	if no.left != nil {
		no.left.preOrder()
	}
	if no.right != nil {
		no.right.preOrder()
	}
}
```

```
func (no *Node) inOrder() {
	if no.left != nil {
		no.left.inOrder()
	}
	fmt.Println(no.val)
	if no.right != nil {
		no.right.inOrder()
	}
}

func (no *Node) posOrder() {
	if no.left != nil {
		no.left.posOrder()
	}
	if no.right != nil {
		no.right.posOrder()
	}
	fmt.Println(no.val)
}

func (t *bst) remove(val int) error {
	if t.root == nil {
		return errors.New("bst vazia")
	} else {
		t.root.removeNode(val)
		t.inserted--
		return nil
	}

}

func (no *Node) removeNode(val int) *Node {
	if val < no.val {
		if no.left != nil {
			no.left = no.left.removeNode(val)
		} else {
			return nil

		}
	} else if val > no.val {
		no.right = no.right.removeNode(val)
	} else {
```

```
                // achemo o noh
                if no.left == nil && no.right == nil {
                        return nil
                } else if no.left != nil && no.right == nil {
                        // filho a esquerda
                        return no.left
                } else if no.left == nil && no.right != nil {
                        // filho a direita
                        return no.right
                } else {
                        // dois filhos, acha o menor da direita e bota la
                        min := no.right.min()
                        no.val = min
                        no.right = no.right.removeNode(min)
                        return no
                }
        }

}
```

3.

- **F,E,L,I,X,U,Z,G,A,R**

a) Pré ordem (R,E,D): F,E,A,L,I,G,X,U,R,Z

B) Em ordem (E,R,D): A,E,F,G,I,L,R,U,X,Z

c) Pós ordem (E,D,R): A,E,G,I,R,U,Z,X,L,F

d) Em níveis: F,E,L,A,I,X,G,U,Z,R

4.

a)pre ordem (RED): 47,30,28,5,21,7,42,54,63,75,86

b) em ordem (ERD): 5,7,21,28,30,42,47,54,63,75,86

c)pos-ordem (EDR): 7,21,5,28,42,30,86,75,63,54,47

d) por níveis: 47,30,54,28,42,63,5,75,21,86,7

e) altura de 30: 1

f) profundidade de 30: 4

5.

```
func (t *bst) IsBST() bool {
          if t.root == nil {
                    return true
          } else {
                    return t.root.IsBST() // func auxiliar pro no
          }
}

func (no *Node) IsBST() bool {
          if no.left != nil {
                    if no.left.val > no.val {
                              return false
                    }
                    if !no.left.IsBST() {
                              return false
                    }
          }
```

```
        if no.right != nil {
                if no.right.val < no.val {
                        return false
                }
                if !no.right.IsBST() {
                        return false
                }
        }
        return true
}
```

6.

```
func (no *Node) Size() int{
        if no == nil {
                return 0
        }
        return 1 + no.left.Size() + no.right.Size()
}
```

7.

```
func (no *Node) Par() int{
        if no == nil {
                return 0
        }
        if no.val %2 ==0 {
                return 1 + no.left.Par() + no.right.Par()
        }
        return 0 + no.left.Par() + no.right.Par() // caso o no atual seja impar
}
```

8.

```
func convertToBalancedBst(arr []int) *Node {
          if len(arr) == 0 {
                    return nil
          }
          mid := len(arr) / 2 // escolhe o elemento do meio como raiz
          root := &Node{val: arr[mid]}
          root.left = convertToBalancedBst(arr[:mid]) // os elementos da esquerda formam a subár
          root.right = convertToBalancedBst(arr[mid+1:]) // o mesmo para a direita
          return root
}
```

9. D) 1 e 2 apenas

10. e)