

# Trabalho Computacional

## 02 - Pesquisa e Ordenação

### **Equipe 01:**

Claudia Ferreira de Souza  
Laís Carvalho Coutinho  
Joao Pedro Silva Fialho  
Rosana Celine Pinheiro Damaceno  
Thaís Araújo de Paiva  
Thalyta Lima Rodrigues

### **Docente:**

Adonias Caetano de Oliveira

### **Disciplina:**

Construção e Análise de  
Algoritmos

# Sumário

**01**

Questão A

**02**

Questão B

**03**

Questão 01

**04**

Questão 02

**05**

Questão 03

**06**

Questão 04

# 01

## Questão A

# Enunciado

**A)** Implemente em Python todos os dez algoritmos de ordenação ensinados em sala de aula, realizando experimentos que avaliem o tempo de execução para ordenar de acordo com as seguintes regras:

**I.** Serão nove vetores com os seguintes tamanhos para cada um: 1000, 3000, 6000, 9000, 12000, 15000, 18000, 21000, 24000.

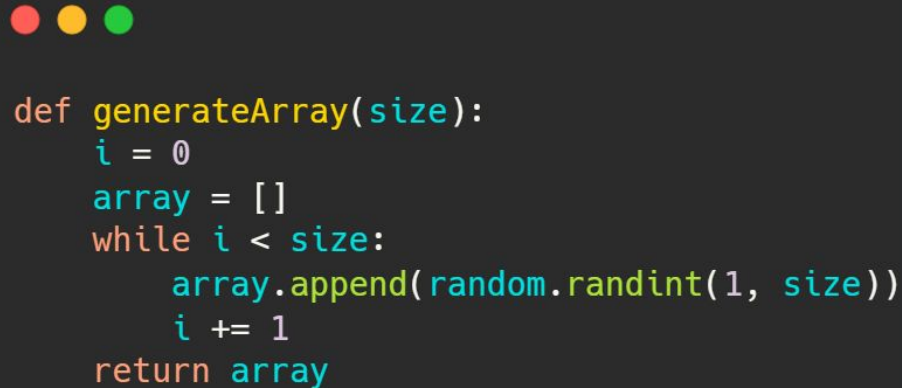
**II.** Os valores armazenados nos nove vetores serão números inteiros gerados aleatoriamente.

**III.** Usar a biblioteca “matplotlib.pyplot”

**IV.** Plotar um gráfico geral,, isto é, todas curvas de tempo de execução dos métodos de ordenação juntos, comparando o desempenho de execução dos algoritmos de acordo com o tamanho do vetor.

**V.** Apresente gráficos específicos que considere conveniente para facilitar a análise comparativa dos métodos, tais como gráficos por métodos com desempenho de execução semelhantes ou com base nos tamanhos.

# Implementação - Geração de array aleatório




```
def generateArray(size):  
    i = 0  
    array = []  
    while i < size:  
        array.append(random.randint(1, size))  
        i += 1  
    return array
```

# Implementação - Métodos de ordenação

```
def bubbleSort(array):  
    i = 0  
    while i < len(array):  
        j = 0  
        while j < len(array) - 1:  
            if array[j] > array[j + 1]:  
                temp = array[j]  
                array[j] = array[j + 1]  
                array[j + 1] = temp  
            j += 1  
        i += 1
```

```
def insertionSort(array):  
    for i in range(1, len(array)):  
        x = array[i]  
        j = i - 1  
  
        while(j >= 0 and array[j] > x):  
            array[j + 1] = array[j]  
            j -= 1  
  
        array[j + 1] = x
```

# Implementação - Métodos de ordenação



```
def selectionSort(array):  
    for i in range(len(array) - 1):  
        min = i  
        for j in range(i + 1,  
len(array)):if(array[j] < array[min]):  
            min = j  
        aux = array[i]  
        array[i] = array[min]  
        array[min] = aux
```

# Implementação - Métodos de ordenação

```
def mergeSort(array, left, right):  
    if(left < right):  
        mid = math.floor((left + right) / 2)  
        mergeSort(array, left, mid)  
        mergeSort(array, mid + 1, right)  
  
        tam = right - left + 1  
        fim1, fim2 = False, False  
        p1, p2 = left, mid + 1  
        temp = []
```

```
for i in range(tam):  
    if(not fim1 and not fim2):  
        if(array[p1] < array[p2]):  
            temp.append(array[p1])  
            p1 += 1  
        else:  
            temp.append(array[p2])  
            p2 += 1  
  
    if(p1 > mid):  
        fim1 = True  
    if(p2 > right):  
        fim2 = True  
    else:  
        if(not fim1):  
            temp.append(array[p1])  
            p1 += 1  
        else:  
            temp.append(array[p2])  
            p2 += 1
```

```
j, k = 0, left  
while(j < tam):  
    array[k] = temp[j]  
    j += 1  
    k += 1
```



# Implementação - Métodos de ordenação

```
def countingSort(array):  
    k = max(array)  
    B = [0 for w in range(len(array))]  
    C = [0 for w in range(k + 1)]  
  
    for i in range(len(array)):  
        C[array[i] + 1] += 1  
    for i in range(1, k):  
        C[i] += C[i - 1]  
    for i in range(len(array) - 1, -1,  
-1):  
        B[C[array[i] - 1] - 1] = array[i]  
        C[array[i] - 1] -= 1  
    for i in range(len(array)):  
        array[i] = B[i]
```

```
def quickSort(array, left, right):  
    if(right > left):  
        i = left  
        for j in range(left + 1, right + 1):  
            if(array[j] < array[left]):  
                i += 1  
                array[i], array[j] = array[j], array[i]  
        array[left], array[i] = array[i], array[left]  
  
        quickSort(array, left, i - 1)  
        quickSort(array, i + 1, right)
```

# Implementação - Métodos de ordenação

```
def bucketSort(array):  
    b = [[] for _ in range(len(array))]  
    for i in range(len(array)):  
        array[i] /= len(array)  
    for i in range(len(array)):  
        bi = int(len(array) * array[i]) -  
1        b[bi].append(array[i])  
    index = 0  
    for v in b:  
        insertionSort(v)  
        for n in v:  
            array[index] = n * len(array)  
            index += 1
```

```
def radixSort(array):  
    max_num_dig = len(str(max(abs(x) for x in array)))  
    aux = [0] * len(array)  
  
    for w in range(max_num_dig):  
        position = [0] * 10  
        count = [0] * 10  
        for i in range(len(array)):  
            d = (array[i] // (10 ** w)) % 10  
            count[d] += 1  
        for j in range(1, 10):  
            position[j] = position[j - 1] + count[j - 1]  
        for i in range(len(array)):  
            d = (array[i] // (10 ** w)) % 10  
            aux[position[d]] = array[i]  
            position[d] += 1  
        for i in range(len(array)):  
            array[i] = aux[i]
```

# Implementação - Métodos de ordenação

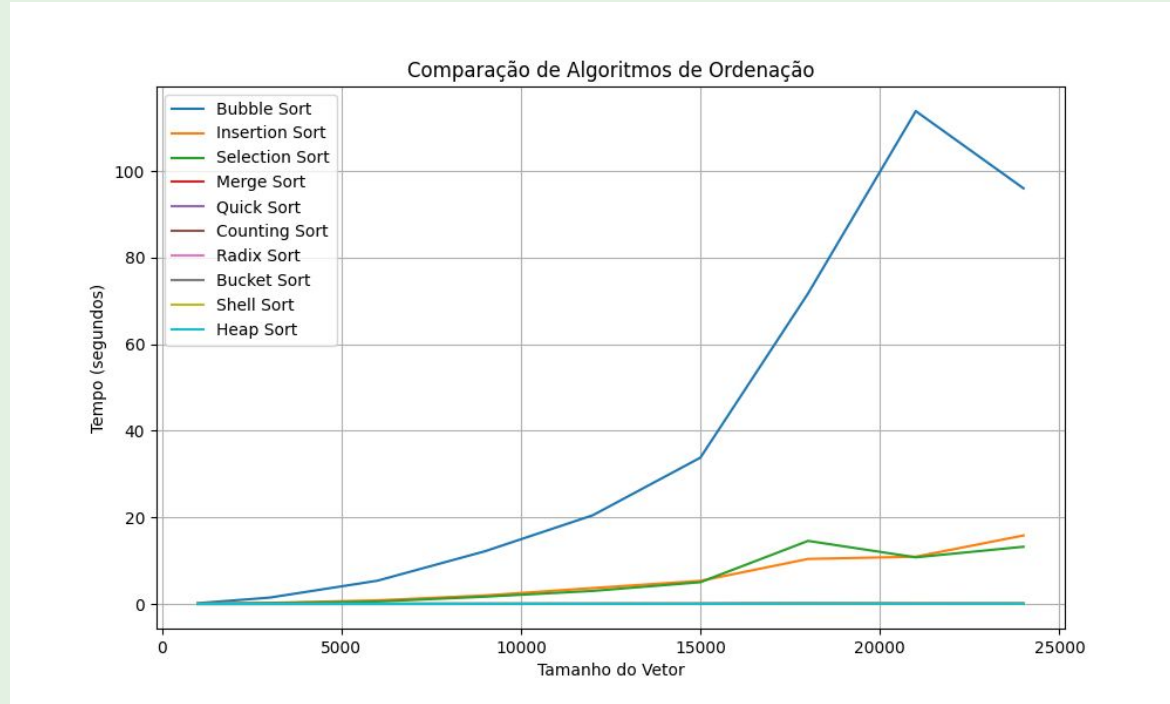
```
def shellSort(array):  
    n = len(array)  
    h = n // 2  
    count = 0  
    while h > 0:  
        for i in range(h, n):  
            c = array[i]  
            j = i  
            while j >= h and c < array[j - h]:  
                array[j] = array[j - h]  
                j -= h  
            array[j] = c  
            count += 1  
        h = int(h / 2.2)
```

# Implementação - Métodos de ordenação

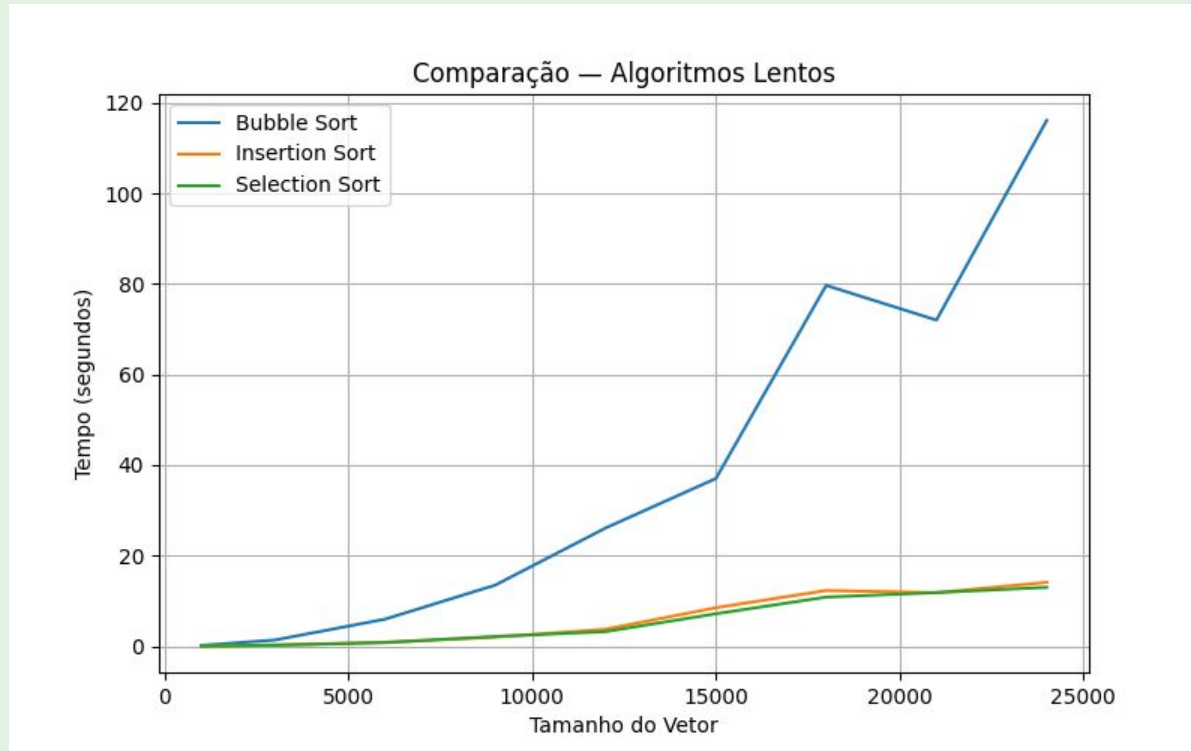
```
def heapSort(array):  
    n = len(array)  
    i = n // 2  
    while True:  
        if i > 0:  
            i -= 1  
            t = array[i]  
        else:  
            n -= 1  
            if n <= 0:  
                return  
            t = array[n]  
            array[n] = array[0]  
        pai = i  
        filho = i * 2 + 1
```

```
while filho < n:  
    if (filho + 1 < n) and (array[filho + 1] >  
array[filho]): filho += 1  
    if array[filho] > t:  
        array[pai] = array[filho]  
        pai = filho  
        filho = pai * 2 + 1  
    else:  
        break  
    array[pai] = t
```

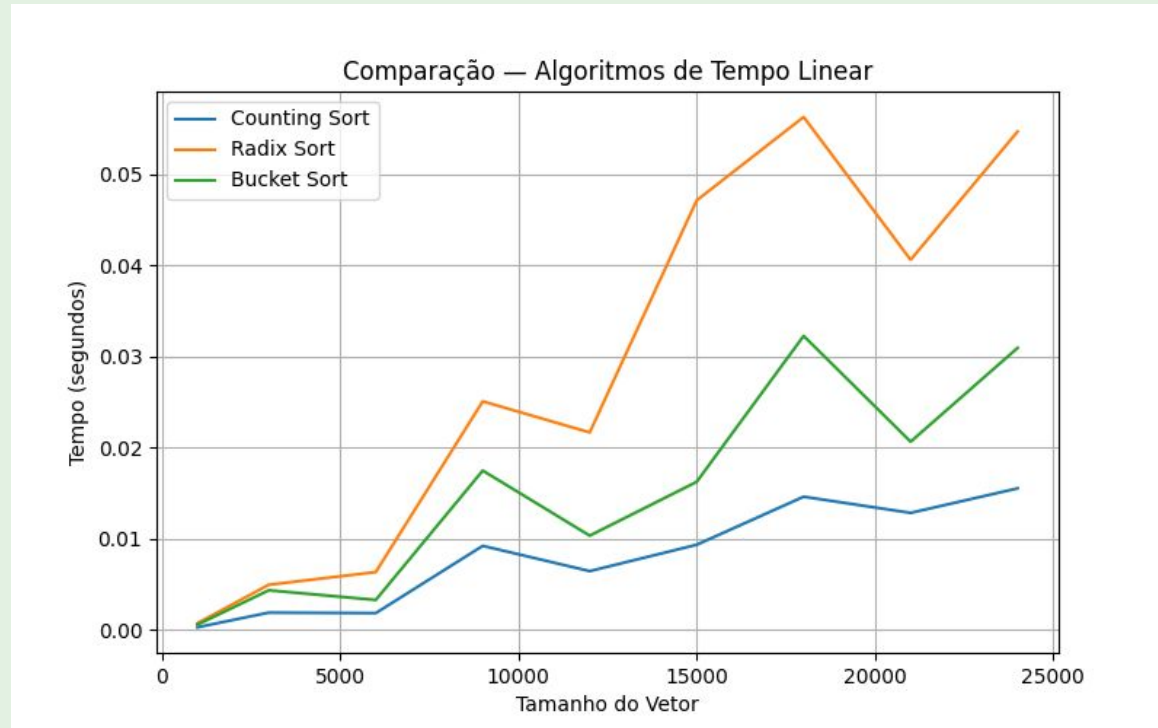
# Análise geral de tempo de execução



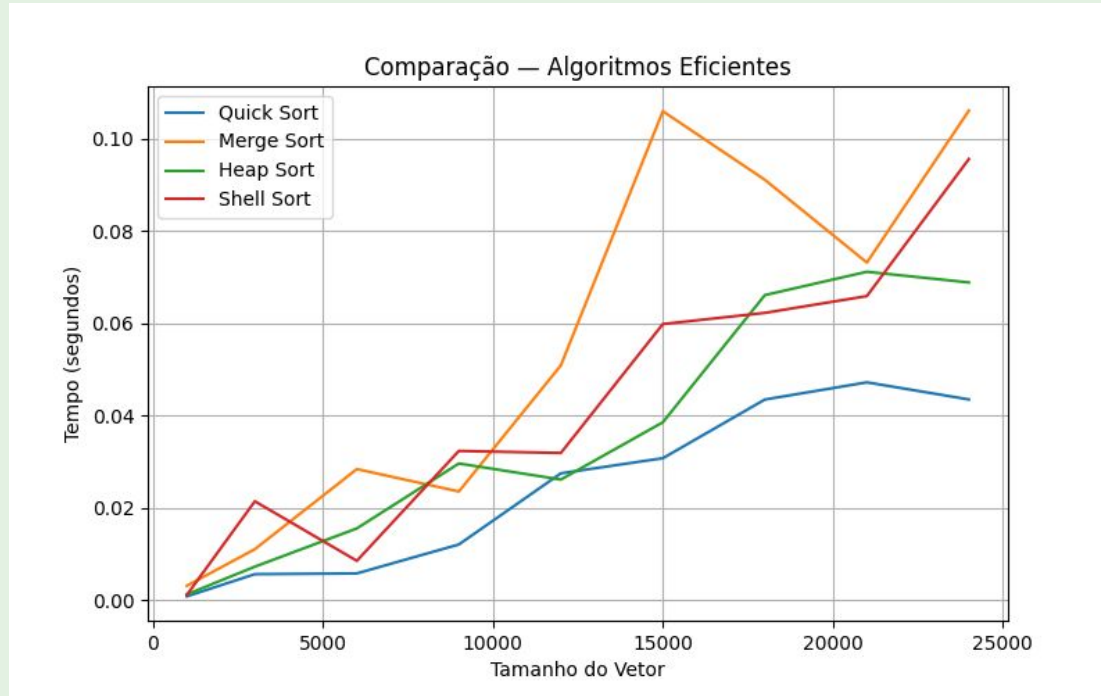
# Análise de tempo de execução - Algoritmos lentos



# Análise de tempo de execução - Algoritmos de tempo linear



# Análise de tempo de execução - Algoritmos eficientes





# 02

## Questão B

# Enunciado

**B)** Implemente em Python a busca linear convencional e com sentinela, busca binária convencional e rápida realizando experimentos que avaliem o tempo de execução para encontrar uma chave de acordo com as seguintes regras:

- I.** Serão nove vetores com os seguintes tamanhos para cada um: 1000, 3000, 6000, 9000, 12000, 15000, 18000, 21000, 24000.
- II.** Os valores armazenados nos nove vetores serão números inteiros gerados aleatoriamente.
- III.** Aplique um método de ordenação linear nas buscas do tipo binária.
- IV.** O elemento chave a ser buscado pode ser informado pelo usuário ou gerado aleatoriamente.
- V.** Usar a biblioteca “matplotlib.pyplot”.
- VI.** Conforme as Figuras 3 e 4, plote um gráfico comparando o tempo de execução dos algoritmos de acordo com o tamanho do vetor.
- VII.** Faça comentários objetivos e sucintos sobre os gráficos.

# Buscas Lineares

## Como funciona uma busca linear convencional?

Algoritmo que localiza um valor específico (chave de busca) em uma lista, verificando cada elemento um a um. Tem complexidade  $O(n)$ .

## Como funciona uma busca linear com sentinela?

É uma variação da busca linear que adiciona uma chave no final do vetor (sentinela), eliminando a necessidade de testar o fim da lista a cada iteração. Isso reduz o número de comparações e acelera ligeiramente o processo. Também tem complexidade  $O(n)$ .

```
def linear_search(arr, key):  
    for i in range(len(arr)):  
        if arr[i] == key:  
            return i  
    return -1  
  
def linear_search_sentinel(arr, key):  
    arr.append(key)  
    i = 0  
    while arr[i] != key:  
        i += 1  
    arr.pop()  
    return i if i != len(arr) else -1
```

# Buscas Binárias

## Como funciona uma busca binária?

Funciona apenas em listas ordenadas. A cada iteração, compara a chave com o elemento do meio: se for menor, continua buscando na metade inferior; se maior, na metade superior. A sua complexidade é  $O(\log n)$ .

## Como funciona uma busca binária rápida?

É a versão recursiva da busca binária. Usa chamadas de função para dividir o problema, mantendo a mesma lógica e complexidade ( $O(\log n)$ ), mas com estrutura de código diferente mais limpa.

```
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1

def fast_binary_search(arr, key, low=0, high=None):
    if high is None:
        high = len(arr) - 1
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == key:
        return mid
    elif arr[mid] > key:
        return fast_binary_search(arr, key, low, mid - 1)
    else:
        return fast_binary_search(arr, key, mid + 1, high)
```

# O que esperar da comparação?

A **Busca Linear Convencional** e a **Busca Linear com Sentinela** apresentam crescimento linear com o tamanho do vetor.

- No entanto, a **busca linear com sentinela** tende a ser um pouco mais rápida, pois otimiza o laço e diminui o número de comparações.

A **Busca Binária Convencional** e a **Busca Binária Rápida** mantêm desempenho mesmo com vetores maiores, pois eliminam metade dos elementos a cada passo.

- Entre elas, a **busca binária convencional** tende a ser mais rápida e eficiente pois evita o custo das chamadas recursivas.

**Conclusão esperada:** Métodos binários devem apresentar um menor tempo quando feitos para grandes volumes de dados. Seguindo do **maior para o menor tempo de execução**, temos:

**Linear convencional > Linear com sentinela > Binária Rápida > Binária convencional**

# 03

## Questão 01

# Enunciado

1) Considere a seguinte estrutura na linguagem C:

```
struct pessoa{  
    int Matricula;  
    char Nome[30];  
    float Nota;  
};
```

I. Implemente uma função em C que dado um array de tamanho N dessa estrutura (necessita de alocação dinâmica de memória), ordene o array pelo campo escolhido pelo usuário. Cada campo deve ser ordenado por um método distinto.

# Algoritmos de Ordenação

## Radix Sort

Algoritmo que ordena números por dígitos usando ordenação por contagem, com complexidade  $O(d \cdot (n+k))$ . Escolhido para **matrículas** por lidar bem com inteiros de tamanho variável e alto desempenho para grandes conjuntos.

### Como funciona uma ordenação em tempo linear com radix sort?

Ordena os números processando um dígito por vez, do menos significativo para o mais significativo. Em cada etapa, usa um método estável para organizar os números conforme o dígito atual, preservando a ordem dos elementos com dígitos iguais. Ao final, o array fica totalmente ordenado, mesmo sem fazer comparações diretas entre números.

```
int obterDigito(int valor, int w, int base) {
    int i = -1, digito;
    do {
        i++;
        digito = valor % base; // Obtém o dígito menos significativo
        valor /= base;         // Remove o último dígito
    } while (i != w);         // Repete até chegar na posição desejada
    return digito;
}
```



# Algoritmos de Ordenação - Radix Sort

```
void ordenarPorRadix(struct pessoa *pessoas, int n) {
    int base = 10;
    int num_digitos = 0;

    int maior = pessoas[0].matricula;
    for (int i = 1; i < n; i++)
        if (pessoas[i].matricula > maior)
            maior = pessoas[i].matricula;

    while (maior > 0) {
        maior /= 10;
        num_digitos++;
    }

    // Vetores auxiliares
    struct pessoa *aux = malloc(n * sizeof(struct pessoa));
    int *count = malloc(base * sizeof(int));
    int *posicao = malloc(base * sizeof(int));
}
```

```
// Para cada dígito (unidade, dezena, centena...)
for (int w = 0; w < num_digitos; w++) {
    for (int j = 0; j < base; j++) {
        count[j] = 0;
        posicao[j] = 0;
    }
    // Conta quantos elementos possuem cada dígito na posição w
    for (int i = 0; i < n; i++) {
        int d = obterDigito(pessoas[i].matricula, w, base);
        count[d]++;
    }
    // Calcula as posições finais de cada dígito no vetor ordenado
    for (int j = 1; j < base; j++) {
        posicao[j] = posicao[j-1] + count[j-1];
    }
    // Coloca os elementos na ordem correta em aux
    for (int i = 0; i < n; i++) {
        int d = obterDigito(pessoas[i].matricula, w, base);
        aux[posicao[d]++] = pessoas[i];
    }
    // Copia o vetor auxiliar de volta para pessoas
    for (int i = 0; i < n; i++) {
        pessoas[i] = aux[i];
    }
}

free(aux);
free(count);
free(posicao);
}
```

# Algoritmos de Ordenação

## Merge Sort

Algoritmo que divide e conquista o vetor, mesclando ordenadamente com complexidade  $O(n \log n)$ . Escolhido para **nomes** por sua estabilidade e eficiência em ordenação de strings.

### Como funciona uma ordenação por divisão e conquista com mergesort?

Divide o array em partes menores até cada uma ter um único elemento. Depois, junta essas partes ordenando durante a união. Esse processo é repetido até o array inteiro estar ordenado. Essa técnica de dividir para conquistar garante eficiência e mantém a ordem relativa de elementos iguais.

```
// Função recursiva do Merge Sort
void ordenarPorMerge(struct pessoa *pessoas, int esquerda, int direita) {
    if (esquerda < direita) {
        int meio = floor((esquerda + direita) / 2);

        // Ordena a primeira metade
        ordenarPorMerge(pessoas, esquerda, meio);

        // Ordena a segunda metade
        ordenarPorMerge(pessoas, meio + 1, direita);

        // Mescla as duas metades já ordenadas
        mesclarPorNome(pessoas, esquerda, meio, direita);
    }
}
```

# Algoritmos de Ordenação - Merge Sort

```
// Função que mescla dois subvetores já ordenados (usando nomes como critério)
void mesclarPorNome(struct pessoa *pessoas, int esquerda, int meio, int direita) {
    int tamanho = direita - esquerda + 1;
    struct pessoa *temp = malloc(tamanho * sizeof(struct pessoa));

    int p1 = esquerda;    // Ponteiro da metade esquerda
    int p2 = meio + 1;    // Ponteiro da metade direita
    int i = 0;

    // Compara os elementos das duas metades e coloca em ordem
    while (p1 <= meio && p2 <= direita) {
        if (strcmp(pessoas[p1].nome, pessoas[p2].nome) < 0)
            temp[i++] = pessoas[p1++];
        else
            temp[i++] = pessoas[p2++];
    }

    // Copia os elementos restantes (se houver)
    while (p1 <= meio) temp[i++] = pessoas[p1++];
    while (p2 <= direita) temp[i++] = pessoas[p2++];

    // Copia de volta para o vetor original
    for (i = 0; i < tamanho; i++)
        pessoas[esquerda + i] = temp[i];

    free(temp);
}
```

# Algoritmos de Ordenação

## Shell Sort

Algoritmo a qual melhora o Insertion Sort ordenando elementos afastados, com complexidade intermediária. Escolhido para **notas** por eficiência com números reais e simplicidade de implementação.

## Como funciona uma ordenação com mergesort?

Começa ordenando elementos distantes entre si usando um intervalo grande, o que reduz deslocamentos longos. Depois, vai diminuindo esse intervalo progressivamente até ordenar elementos vizinhos, finalizando com uma ordenação simples para deixar o array completamente ordenado. Isso acelera a ordenação em relação a métodos simples como o Insertion Sort.

```
// Implementação do Shell Sort para ordenar por nota (ordem decrescente)
void ordenarPorShell(struct pessoa *pessoas, int n) {
    int gap = 1;

    // Calcula o maior intervalo inicial (gap)
    do {
        gap = 3 * gap + 1;
    } while (gap < n);

    // Enquanto houver intervalos para comparar
    do {
        gap /= 3;

        // Faz inserção com gap
        for (int i = gap; i < n; i++) {
            struct pessoa temp = pessoas[i];
            int j = i - gap;

            while (j >= 0 && temp.nota > pessoas[j].nota) {
                pessoas[j + gap] = pessoas[j];
                j -= gap;
            }
            pessoas[j + gap] = temp;
        }
    } while (gap > 1);
}
```

# 04

## Questão 02

# Enunciado

**2) Implementar o algoritmo Bubble-Sort para ordenar uma lista encadeada de números. Linguagens permitidas: C, C++, Java, Python e Ruby.**

# Algoritmos de Ordenação

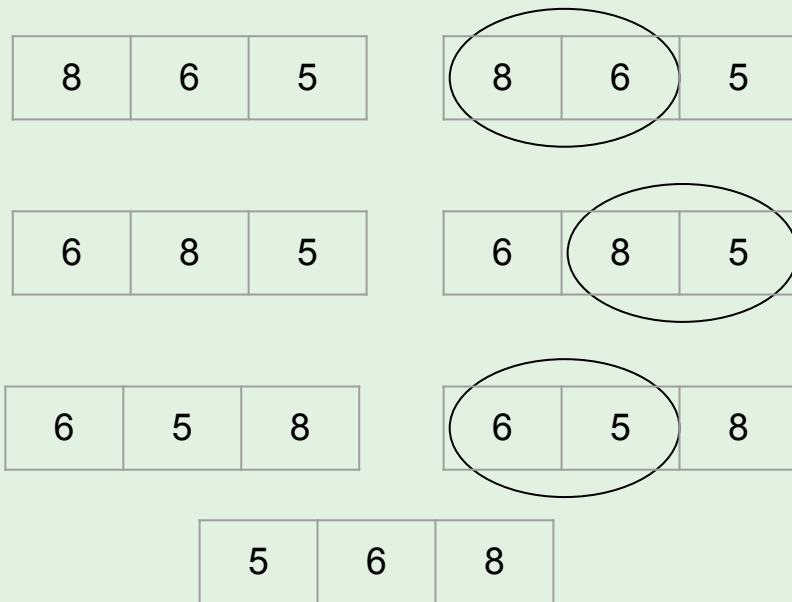
## Bubble-Sort

### O que é?

O algoritmo **Bubble Sort**, também conhecido como ordenação por flutuação, é um algoritmo de ordenação simples que compara elementos adjacentes de uma lista.

### Como funciona?

O **Bubble Sort** classifica os elementos comparando elementos adjacentes da esquerda para a direita em uma matriz e trocando os elementos se estiverem fora de ordem.



# Resolução - Python

```
2
3 class Node: 1 usage
4     def __init__(self, value):
5         self.value = value
6         self.next = None
7
8 class List: 1 usage
9     def __init__(self):
10         self.start = None
11         self.size = 0
12
13     def insert(self, value): 6 usages
14         new = Node(value)
15         if not self.start:
16             self.start = new
17         else:
18             current = self.start
19             while current.next:
20                 current = current.next
21             current.next = new
22             self.size += 1
```

```
23
24 def bubbleSort(self): 1 usage
25     if self.size < 2:
26         return
27
28     for _ in range(self.size):
29         current = self.start
30         while current and current.next:
31             if current.value > current.next.value:
32                 current.value, current.next.value = current.next.value, current.value
33             current = current.next
34
35 def show(self): 2 usages
36     current = self.start
37     while current:
38         print(current.value, end=" -> ")
39         current = current.next
40     print("None")
```



# Resolução - Phyton

```
42     l = List()
43
44     l.insert(5)
45     l.insert(6)
46     l.insert(-9)
47     l.insert(9)
48     l.insert(0)
49     l.insert(4)
50
51     l.show()
52     l.bubbleSort()
53     l.show()
```

```
5 -> 6 -> -9 -> 9 -> 0 -> 4 -> None
-9 -> 0 -> 4 -> 5 -> 6 -> 9 -> None
```

# 05

## Questão 03

# Enunciado

- Implemente em Java os algoritmos de busca sequencial e binária para listas simplesmente encadeadas.

# Busca sequencial (linear)

- Percorre todos os elementos
- Utilizada quando não há informações adicionais sobre os dados a serem pesquisados.
- Algoritmo que localiza um valor específico (chave de busca) em uma lista, verificando cada elemento um a um. Tem complexidade  $O(n)$ .

## A busca termina quando:

- Quando o elemento é encontrado
- Ou quando toda a estrutura foi percorrida e o elemento  $x$  não foi encontrado

# Busca Binária

- A Busca é mais eficiente se os dados estiverem ordenados.
- Elimina o maior número possível de elementos em futuras buscas
- Escolha da mediana dos elementos, porque ela elimina em qualquer caso metade dos elementos da lista.

# Lista Simplesmente encadeada

```
1
2 public class Node {
3     int elemento; // armazena o dado do nó
4     Node proximo; // Referência para o próximo nó da lista
5
6
7     public Node(int elemento) {
8         this.elemento = elemento;
9         this.proximo = null;
10    }
11
12 }
13
```

# Busca sequencial (linear)

```
1 //busca sequencial
2 public boolean busca_Sequencial(int valor) {
3     int count = 0;
4     Node pAtual = inicio;
5
6     while (pAtual != null) {
7         count++;
8         if (pAtual.elemento == valor) { // Se o valor do nó atual for igual ao buscado
9             System.out.println("Quantidade de comparações Busca sequencial:"+count);
10            return true; // Valor encontrado
11        }
12        pAtual = pAtual.proximo; // Vai para o próximo nó
13
14    }
15    return false;
16 }
```

# Busca Binária

```
1 //Ponteiros (slow e fast) para encontra nó do meio
2 public Node meio(Node inicio,Node ultimo) {
3     if(inicio==null) {
4         return null;
5     }
6
7     Node slow = inicio;
8     Node fast = inicio;
9
10    while (fast.proximo != ultimo && fast.proximo.proximo != ultimo) {
11        //fast avança dois nós
12        fast = fast.proximo.proximo;
13
14        //slow avança um nó
15        slow = slow.proximo;
16    }
17
18    return slow;
19 }
```

```
1 // busca binaria
2 private boolean busca_Binaria(Node inicio,Node ultimo,int element) {
3     int count = 0;
4     while(inicio != ultimo) {
5         // Encontra o nó do meio entre inicio e ultimo
6         Node m= meio(inicio,ultimo);
7
8         // Se não houver meio (encerra a busca)
9         if(m==null) {
10             return false;
11         }
12         count++;
13
14         // Caso o elemento do meio seja o valor procurado
15         if(m.elemento== element) {
16             System.out.println("Quantidade de comparações Busca binária:"+count);
17             return true; // Elemento encontrado
18         }
19
20         // Se o elemento do meio for menor que o valor procurado
21         else if(m.elemento< element) {
22             //busca continua na metade direita da lista.
23             inicio=m.proximo;
24         }
25
26         // Se o elemento do meio for maior que o valor procurado
27         else {
28             //busca continua na metade esquerda da lista.
29             ultimo= m;
30         }
31     }
32     return false;
33 }
34 }
```



# 06

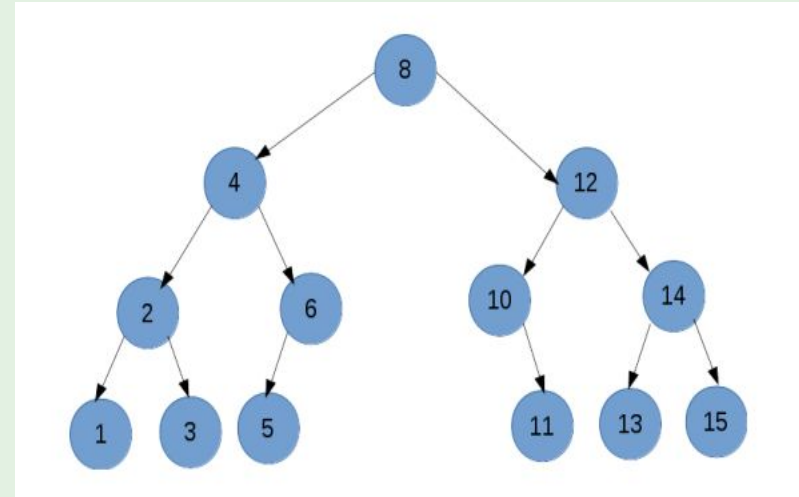
## Questão 04

# Enunciado

- Implemente a estrutura de dados Árvore Binária em Python ou Java ou C++ usando conceitos de orientação a objetos. A sua classe deve ter os seguintes métodos:
  - a) **getNosFolha:** retorna uma lista de nós folhas;
  - b) **getGrau(int nó):** retorna o grau de um nó;
  - c) **altura:** retorna altura da árvore;
  - d) **profundidade:** retorna o valor de profundidade da árvore

# Árvore Binária

- Uma **árvore binária** é uma estrutura de dados hierárquica na qual cada nó pode ter no máximo dois filhos, chamados de filho esquerdo e filho direito. Ela é amplamente usada em algoritmos de busca, ordenação e organização de dados.
- **Grau:** é o número de filhos que um nó possui. Em uma árvore binária, o grau máximo de um nó é 2.
- **Profundidade:** é a quantidade de arestas do nó até a raiz da árvore. A raiz tem profundidade 0.
- **Altura:** é a maior profundidade entre todos os nós da árvore. Em outras palavras, é a distância do nó mais profundo até a raiz.



# Classe Principal

- Primeiramente, devemos deixar os métodos que iremos chamar “públicos”, para que não ocorra de o usuário não usar um método privado e que possa ser usado de forma errada.
- O método push é usado para adicionar elementos à árvore binária, caso já existe um valor, ele não irá duplicar, ele simplesmente retornará um ponteiro para “nullptr” como forma de tratamento de erros;

```
template <typename T> class Arvore {
public:
    T valor;
    Arvore *esquerda;
    Arvore *direita;

    Arvore<T>(T valor) {
        this->valor = valor;
        this->esquerda = nullptr;
        this->direita = nullptr;
    }

    /* Arvore *push(T valor) { ... }*/

    /*std::vector<T> getNosFolha() { ... }*/

    struct FindResult {
        Arvore *node;
        int nivel;
    };

    /* FindResult find(T valor, int nivel = 0) { ... }*/

    /*int getGrau(T valor) { ... };*/

    int getProfundidade(T valor) { return this->find(valor).nivel; }

    int getAltura() { return getAlturaHelper(this); }

private:
    /* int getAlturaHelper(Arvore *node) { ... }*/
    /* void getNosFolhaHelper(std::vector<T> *out, Arvore<T> *node) { ... };*/
};
```

# Funções Genéricas

## Find

```
struct FindResult {
    Arvore *node;
    int nivel;
};

FindResult find(T valor, int nivel = 0) {
    if (valor == this->valor)
        return FindResult{this, nivel};
    if (valor < this->valor)
        return this->esquerda->find(valor, nivel + 1);
    if (valor > this->valor)
        return this->direita->find(valor, nivel + 1);
    return FindResult{nullptr, 0};
}
```

## Push

```
Arvore *push(T valor) {
    // Valor repetido
    if (this->valor == valor)
        return nullptr;

    if (valor < this->valor) {
        if (this->esquerda == nullptr) {
            return this->esquerda = new Arvore(valor);
        } else {
            return this->esquerda->push(valor);
        }
    } else {
        if (this->direita == nullptr) {
            return this->direita = new Arvore(valor);
        } else {
            return this->direita->push(valor);
        }
    }
    return nullptr;
};
```

# Funções Solicitadas

## GetNosFolha

```
std::vector<T> getNosFolha() {
    static auto out = std::vector<T>();
    out.clear();
    getNosFolhaHelper(&out, this);
    return out;
}

void getNosFolhaHelper(std::vector<T> *out, Arvore<T> *node) {
    if (node->esquerda != nullptr) {
        getNosFolhaHelper(out, node->esquerda);
    }
    out->push_back(node->valor);
    if (node->direita != nullptr) {
        getNosFolhaHelper(out, node->direita);
    }
};
```

## GetGrau e GetProfundidade

```
int getGrau(T valor) {
    auto find = this->find(valor).node;
    return (find->esquerda != nullptr) + (find->direita != nullptr);
};

int getProfundidade(T valor) { return this->find(valor).nivel; }
```

# Funções Solicitadas

## GetAltura



```
int getAltura() { return getAlturaHelper(this); }

int getAlturaHelper(Arvore *node) {
    if (node == nullptr)
        return -1; // altura de árvore vazia é -1
    int alturaEsquerda = getAlturaHelper(node->esquerda);
    int alturaDireita = getAlturaHelper(node->direita);
    return 1 +
        (alturaEsquerda > alturaDireita ? alturaEsquerda : alturaDireita);
};
```

# Valor Duplicado

Utilizando a árvore binária, implemente um programa que imprime os números duplicados da lista abaixo.

14	18	4	9	7	15	3	4	20	9	5
----	----	---	---	---	----	---	---	----	---	---

```
int main() {  
    auto raiz = Arvore(10);  
  
    auto arr = {14, 18, 4, 9, 7, 15, 3, 4, 20, 9, 5};  
    for (auto &elem : arr) {  
        auto node = raiz.push(elem);  
        if (node == nullptr)  
            printf("O valor %d é repetido e já está na árvore.\n", elem);  
    }  
}
```

```
→ Codigos-CANA git:(main) x clang++ ./questao4.cpp  
→ Codigos-CANA git:(main) x ./a.out  
O valor 4 é repetido e já está na árvore.  
O valor 9 é repetido e já está na árvore.  
→ Codigos-CANA git:(main) x
```



# Obrigada pela Atenção

Alguma dúvida?