UNIVERSIDADE FEDERAL DE RORAIMA
CENTRO DE CIÊNCIA E TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Luciano dos Santos Nascimento e Wesley Silva Araújo

UMA PESQUISA APROFUNDADA SOBRE ALGORITMOS DE ORDENAÇÂO

Boa Vista/RR
2025

Sumário

Introdução

A ordenação e a busca são problemas fundamentais na ciência da computação e desempenham um papel crucial no desempenho de diversas aplicações, desde bancos de dados até algoritmos de inteligência artificial. Para solucionar esses problemas, diversos algoritmos foram desenvolvidos, cada um com características específicas quanto à eficiência, complexidade e aplicabilidade.

Este trabalho tem como objetivo apresentar uma análise aprofundada dos principais algoritmos de ordenação e busca, abordando suas definições, funcionamento, vantagens, desvantagens e exemplos de implementação. Os algoritmos estudados incluem:

- Selection Sort
- Insertion Sort
- Bubble Sort
- Heap Sort
- Radix Sort
- Quick Sort
- Merge Sort
- Busca Sequencial
- Busca Binária

Cada um desses algoritmos será detalhado de forma teórica e prática, permitindo uma compreensão clara de seu funcionamento e aplicabilidade. Além disso, referências bibliográficas serão utilizadas para embasar a pesquisa e possibilitar um estudo mais aprofundado sobre o tema.

Dessa forma, este estudo contribui para a formação acadêmica e profissional ao fornecer um material de apoio que facilita a compreensão e implementação de algoritmos essenciais para a computação.

Selection Sort

Definição

Selection sort é um algoritmo de classificação simples. Esse algoritmo de classificação, como o insertion sort, é um algoritmo baseado em comparação no local em que a lista é dividida em duas partes, a parte classificada na extremidade esquerda e a parte não classificada na extremidade direita. Inicialmente, a parte classificada está vazia e a parte não classificada é a lista inteira.

O menor elemento é selecionado do array não classificado e trocado com o elemento mais à esquerda, e esse elemento se torna parte do array classificado. Esse processo continua movendo os limites do array não classificado por um elemento para a direita.

Este algoritmo não é adequado para grandes conjuntos de dados, pois suas complexidades médias e de pior caso são de $\mathbf{O(n^2)}$, onde **n** é o número de itens.

Algoritmo:

```c
#include <stdio.h>
void trocar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void selectionSort(int arr[], int tamanho) {
    for (int i = 0; i < tamanho - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < tamanho; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        trocar(&arr[min_idx], &arr[i]);
    }
}
void imprimirArray(int arr[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int tamanho = sizeof(arr) / sizeof(arr[0]);
    imprimirArray(arr, tamanho);
    selectionSort(arr, tamanho);
    imprimirArray(arr, tamanho);
    return 0;
}
```

Vantagem e desvantagens

Vantagens do Selection Sort

- **Fácil de entender e implementar** – O algoritmo é simples, tornando-o ideal para aprendizado inicial sobre ordenação.
- **Não requer memória extra significativa** – Ele é um algoritmo **in-place**, ou seja, não precisa de espaço adicional além da própria lista.
- **Bom para conjuntos pequenos de dados** – Para listas pequenas, seu desempenho pode ser aceitável, especialmente quando a estabilidade não é um critério importante.
- **Número fixo de trocas** – Em comparação com outros algoritmos como o Bubble Sort, ele faz menos trocas, o que pode ser útil em algumas situações.

Desvantagens do Selection Sort

- **Desempenho ruim para listas grandes** – Sua complexidade é **O(n²)** no pior, melhor e caso médio, tornando-o ineficiente para grandes volumes de dados.
- **Não é estável** – Se houver elementos repetidos, a posição relativa entre eles pode ser alterada, o que pode ser um problema em algumas aplicações.

- **Não é adaptativo** – Mesmo que a lista esteja parcialmente ordenada, o Selection Sort sempre executa o mesmo número de operações.
- **Mais lento que algoritmos mais eficientes** – Quick Sort, Merge Sort e Heap Sort possuem desempenho muito superior para listas maiores.

O Selection Sort pode ser útil para situações específicas, mas geralmente não é a melhor escolha para ordenação de grandes conjuntos de dados.

Bubble Sort

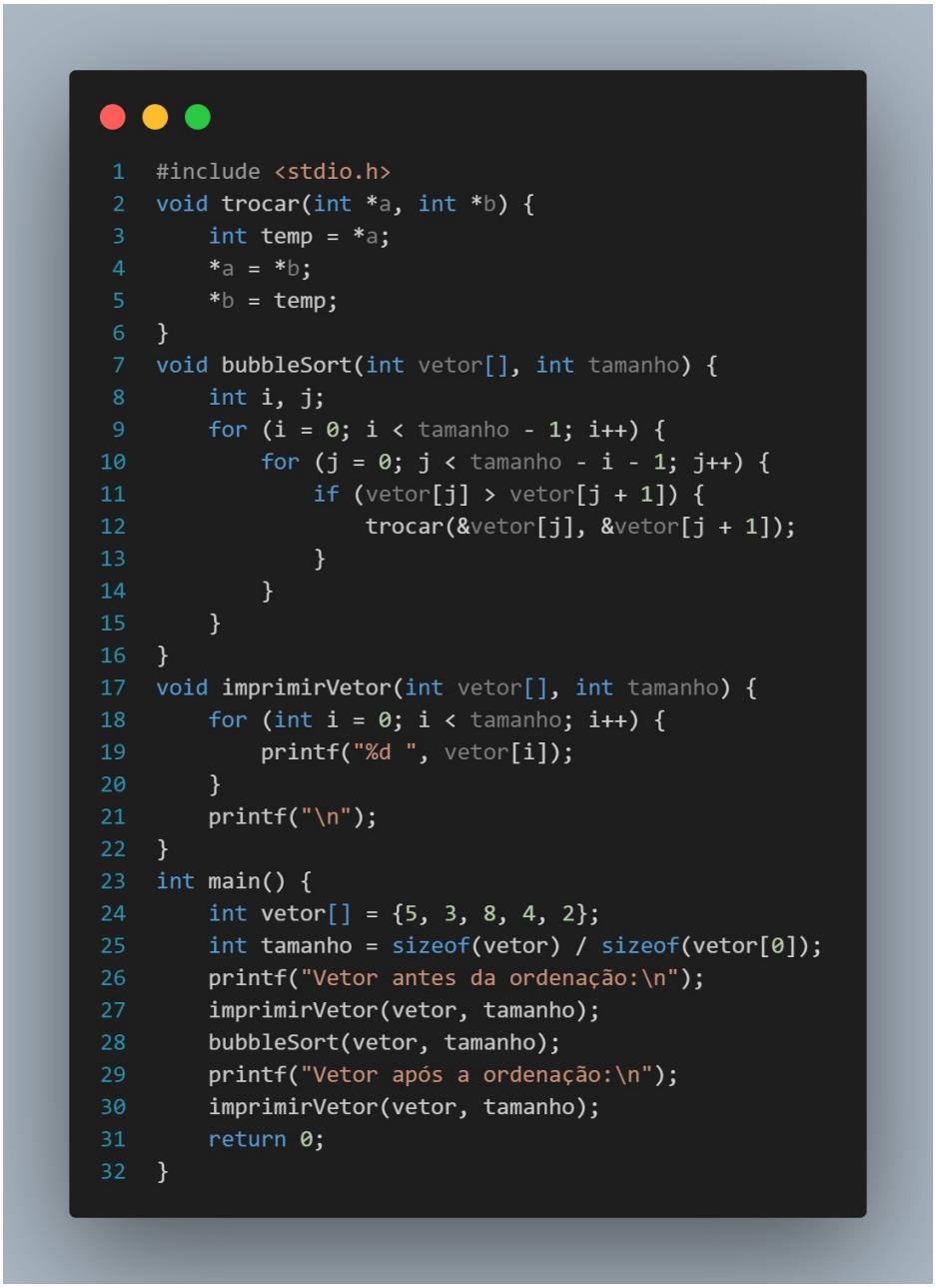
definição

O *Bubble sort* é um algoritmo simples de ordenação que recebe como entrada uma lista de elementos e produz uma lista ordenada de acordo com um critério. É um algoritmo popular, mas menos eficiente em relação a outros algoritmos de ordenação. O funcionamento do *Bubble sort* é baseado na comparação dos elementos da lista e na troca de posição dos mesmos quando necessário para atingir a ordenação pretendida. Ele atua localmente, ou seja, pode alterar a própria lista de entrada.

Passo a passo do algoritmo

1. Percorra a lista da esquerda para a direita.
2. Compare cada elemento com o elemento adjacente à sua direita.
3. Se o elemento à esquerda for maior que o elemento à direita, troque-os de posição.
4. Repita os passos 1a 3 até que a lista esteja ordenada.

Algoritimo:



```c
#include <stdio.h>
void trocar(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void bubbleSort(int vetor[], int tamanho) {
    int i, j;
    for (i = 0; i < tamanho - 1; i++) {
        for (j = 0; j < tamanho - i - 1; j++) {
            if (vetor[j] > vetor[j + 1]) {
                trocar(&vetor[j], &vetor[j + 1]);
            }
        }
    }
}
void imprimirVetor(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}
int main() {
    int vetor[] = {5, 3, 8, 4, 2};
    int tamanho = sizeof(vetor) / sizeof(vetor[0]);
    printf("Vetor antes da ordenação:\n");
    imprimirVetor(vetor, tamanho);
    bubbleSort(vetor, tamanho);
    printf("Vetor após a ordenação:\n");
    imprimirVetor(vetor, tamanho);
    return 0;
}
```

Vantagens do Bubble Sort

- Fácil de entender e implementar – O algoritmo é simples e ideal para aprendizado inicial sobre ordenação.
- Ordenação estável – Mantém a ordem relativa de elementos iguais, o que pode ser útil em algumas aplicações.

- Pouco uso de memória – É um algoritmo in-place, ou seja, ordena os elementos diretamente no vetor sem precisar de memória extra significativa.
- Funciona bem para listas pequenas ou quase ordenadas – Com a otimização de parada antecipada, pode ser eficiente se a lista já estiver quase ordenada.

Desvantagens do Bubble Sort

- Desempenho ruim para listas grandes – Sua complexidade é $O(n^2)$ no pior e caso médio, tornando-o muito ineficiente para grandes volumes de dados.
- Número excessivo de comparações e trocas – Mesmo se a lista já estiver quase ordenada, ele continua fazendo muitas operações desnecessárias sem otimizações.
- Mais lento que outros algoritmos – Quick Sort, Merge Sort e Heap Sort possuem desempenho muito superior.
- Não é adaptativo sem otimizações – Sem a verificação de parada antecipada, ele sempre percorre todas as interações, mesmo se a lista já estiver ordenada.

O Bubble Sort é útil para aprendizado, mas raramente é usado na prática para ordenação de grandes conjuntos de dados devido ao seu baixo desempenho.

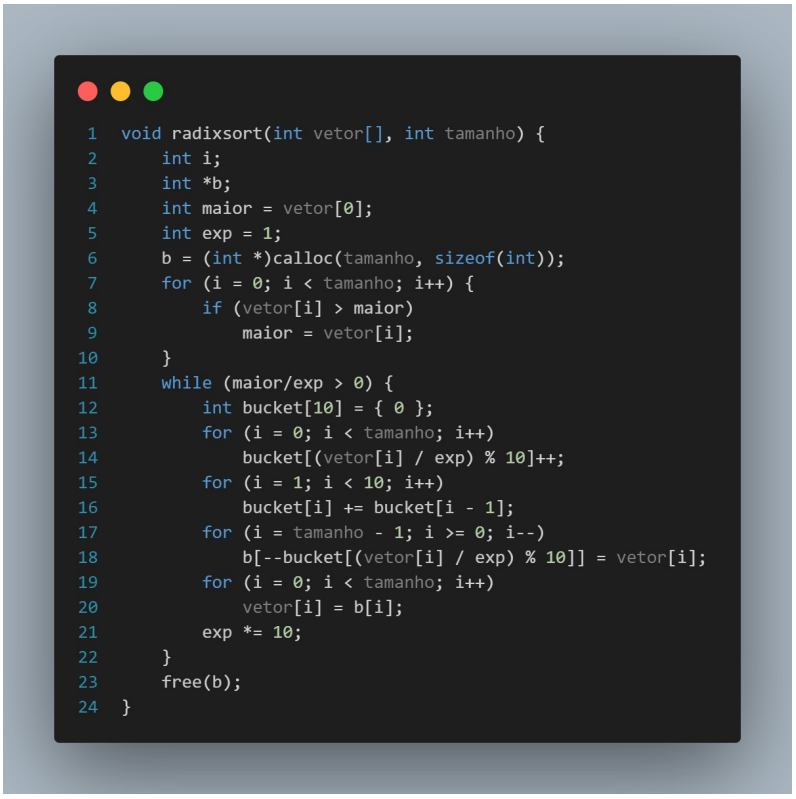
Radix sort

definição

O Radix sort é um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia

de caracteres ou número, e o *radix sort* ordena estas chaves em qualquer ordem relacionada com a lexicografia. Na ciência da computação, radix sort é um algoritmo de ordenação que ordena inteiros processando dígitos individuais. Como os inteiros podem representar strings compostas de caracteres (como nomes ou datas) e pontos flutuantes especialmente formatados, radix sort não é limitado somente a inteiros.

Algoritmo:



```c
void radixsort(int vetor[], int tamanho) {
    int i;
    int *b;
    int maior = vetor[0];
    int exp = 1;
    b = (int *)calloc(tamanho, sizeof(int));
    for (i = 0; i < tamanho; i++) {
        if (vetor[i] > maior)
            maior = vetor[i];
    }
    while (maior/exp > 0) {
        int bucket[10] = { 0 };
        for (i = 0; i < tamanho; i++)
            bucket[(vetor[i] / exp) % 10]++;
        for (i = 1; i < 10; i++)
            bucket[i] += bucket[i - 1];
        for (i = tamanho - 1; i >= 0; i--)
            b[--bucket[(vetor[i] / exp) % 10]] = vetor[i];
        for (i = 0; i < tamanho; i++)
            vetor[i] = b[i];
        exp *= 10;
    }
    free(b);
}
```

Aqui estão as **vantagens** e **desvantagens** do **Radix Sort**:

Vantagens do Radix Sort

- **Desempenho eficiente em casos específicos** – Para números inteiros ou strings de tamanho fixo, o Radix Sort pode ser mais rápido que algoritmos baseados em

comparações, como Quick Sort (**O(n log n)**), com uma complexidade de **O(nk)**, onde *n* é o número de elementos e *k* é o número de dígitos do maior número.

- **Ordenação estável** – Mantém a ordem relativa de elementos iguais, o que pode ser importante para alguns algoritmos ou problemas.
- **Não depende de comparações diretas** – Ao contrário de algoritmos como Quick Sort ou Merge Sort, o Radix Sort não faz comparações entre elementos, o que pode ser vantajoso em certos cenários de dados.
- **Boa performance com dados com poucos dígitos** – Se os números ou chaves a serem ordenados têm poucos dígitos, o Radix Sort pode ser muito mais eficiente em termos de tempo do que algoritmos de ordenação baseados em comparação.

Desvantagens do Radix Sort

- **Requer memória adicional** – O Radix Sort precisa de espaço extra para os "baldes" (buckets), o que pode ser um problema quando há restrições de memória.
- **Não é eficiente para números de ponto flutuante ou grandes números com muitos dígitos** – O algoritmo é mais eficiente quando os números são inteiros e de tamanho fixo. Para dados com grande variação em tamanho de chave ou números com muitos dígitos, ele pode se tornar menos eficiente.
- **Não é uma escolha para todos os tipos de dados** – O Radix Sort é ideal para inteiros e strings de tamanho fixo, mas não é adequado para tipos de dados mais complexos ou quando a comparação direta de valores é necessária.
- **Complexidade de implementação** – Embora o conceito seja simples, a implementação do Radix Sort pode ser mais complexa do que algoritmos baseados em comparações, especialmente devido à necessidade de manipulação dos dígitos e contagem.

O **Radix Sort** é particularmente útil quando há grandes volumes de dados inteiros ou strings e quando esses dados têm um número fixo de dígitos. No entanto, sua eficiência depende das características dos dados a serem ordenados e pode não ser adequado em todos os casos.

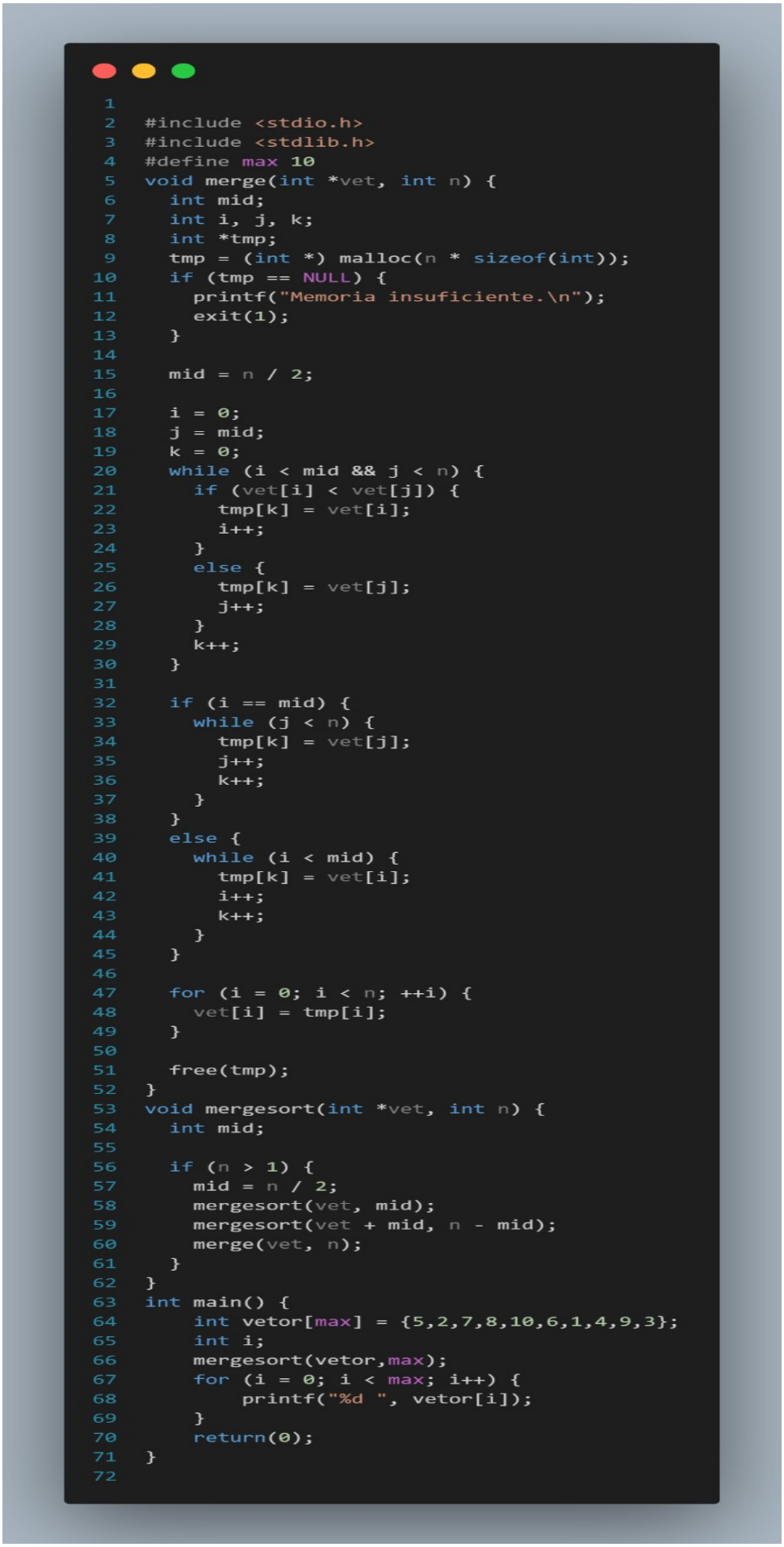
Merge Sort: O que é?

Merge Sort é um algoritmo de ordenação baseado na técnica de dividir e conquistar. Ele divide repetidamente o vetor ou lista em duas metades até que cada sublista tenha apenas um elemento e, então, começa a combinar essas sublistas de forma ordenada. O processo de combinação é feito através de uma operação de mesclagem (merge), onde duas sublistas ordenadas são fundidas em uma única lista ordenada.

Como funciona?

1. **Divisão:** O vetor é repetidamente dividido em duas metades, até que cada sublista tenha apenas um elemento.
2. **Mesclagem:** As sublistas são combinadas de volta, mas de forma ordenada. O algoritmo compara os elementos das sublistas e os coloca na ordem correta.

Algoritmo em c:



```c

#include <stdio.h>
#include <stdlib.h>
#define max 10
void merge(int *vet, int n) {
  int mid;
  int i, j, k;
  int *tmp;
  tmp = (int *) malloc(n * sizeof(int));
  if (tmp == NULL) {
    printf("Memoria insuficiente.\n");
    exit(1);
  }

  mid = n / 2;

  i = 0;
  j = mid;
  k = 0;
  while (i < mid && j < n) {
    if (vet[i] < vet[j]) {
      tmp[k] = vet[i];
      i++;
    }
    else {
      tmp[k] = vet[j];
      j++;
    }
    k++;
  }

  if (i == mid) {
    while (j < n) {
      tmp[k] = vet[j];
      j++;
      k++;
    }
  }
  else {
    while (i < mid) {
      tmp[k] = vet[i];
      i++;
      k++;
    }
  }

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

  free(tmp);
}
void mergesort(int *vet, int n) {
  int mid;

  if (n > 1) {
    mid = n / 2;
    mergesort(vet, mid);
    mergesort(vet + mid, n - mid);
    merge(vet, n);
  }
}
int main() {
    int vetor[max] = {5,2,7,8,10,6,1,4,9,3};
    int i;
    mergesort(vetor,max);
    for (i = 0; i < max; i++) {
        printf("%d ", vetor[i]);
    }
    return(0);
}

```

Vantagens do Merge Sort

- **Desempenho garantido:** Sua complexidade no pior, melhor e caso médio é **O(n log n)**, o que o torna mais eficiente que algoritmos como Bubble Sort ou Selection Sort, que possuem complexidade **O(n²)**.
- **Estabilidade:** O Merge Sort é um algoritmo **estável**, o que significa que ele mantém a ordem relativa de elementos iguais.
- **Funciona bem para grandes volumes de dados:** Por ter uma complexidade eficiente e garantida, o Merge Sort é adequado para ordenação de grandes conjuntos de dados.
- **Adaptável para listas vinculadas:** Ao contrário de outros algoritmos, o Merge Sort pode ser adaptado de maneira eficiente para listas encadeadas, além de vetores.
- **Desempenho previsível:** O Merge Sort não é influenciado pela ordenação prévia do conjunto de dados. Sempre terá a mesma performance, independentemente da ordem inicial.

Desvantagens do Merge Sort

- **Uso de memória adicional:** O Merge Sort não é um algoritmo **in-place**, o que significa que ele exige espaço extra para armazenar as sublistas durante o processo de mesclagem. Isso pode ser um problema quando a memória disponível é limitada.
- **Mais lento em listas pequenas:** Embora tenha uma complexidade **O(n log n)**, o Merge Sort pode ser mais lento que algoritmos simples, como Insertion Sort, quando aplicado a listas pequenas, devido ao custo de suas operações auxiliares.
- **Complexidade na implementação de listas vinculadas:** Embora seja possível aplicar o Merge Sort em listas vinculadas, a implementação pode ser mais complexa em comparação com vetores ou arrays.

Insertion Sort

Insertion Sort ou ordenação por inserção é o método que percorre um vetor de elementos da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda. É considerado um método de ordenação estável.

Um método de ordenação é estável se a ordem relativa dos itens iguais não se altera durante a ordenação.

O funcionamento do algoritmo é bem simples: consiste em cada passo a partir do segundo elemento selecionar o próximo item da sequência e colocá-lo no local apropriado de acordo com o critério de ordenação.

Funcionamento:

1. O algoritmo começa com um subarray ordenado de um único elemento, que é o primeiro elemento da lista a ser ordenada.
2. O algoritmo então percorre a lista a partir do segundo elemento.
3. Para cada elemento da lista, o algoritmo compara-o com os elementos do subarray ordenado.
4. Se o elemento for menor que qualquer um dos elementos do subarray ordenado, o algoritmo o insere na posição correta no subarray ordenado.
5. O algoritmo repete os passos 2 a 4 até que todos os elementos da lista tenham sido processados.

Algoritmo:

```
int * insertionSort(int original[], int length) {
	int i, j, atual;

	for (i = 1; i < length; i++) {
		atual = original[i];
		j = i - 1;

		while ((j >= 0) && (atual <
original[j])) {
			original[j + 1] = original[j];
			j = j - 1;
		}

		original[j + 1] = atual;
	}

	return (int * )original;
}
```

Vantagens e Desvantagens:

Vantagens:

- É um bom método quando se desejar adicionar poucos elementos em um arquivo já ordenado, pois seu custo é linear.
- O algoritmo de ordenação por inserção é estável.
- é de simples implementação, leitura e manutenção

Desvantagens:

- Alto custo de movimentação de elementos no vetor.

Heap Sort

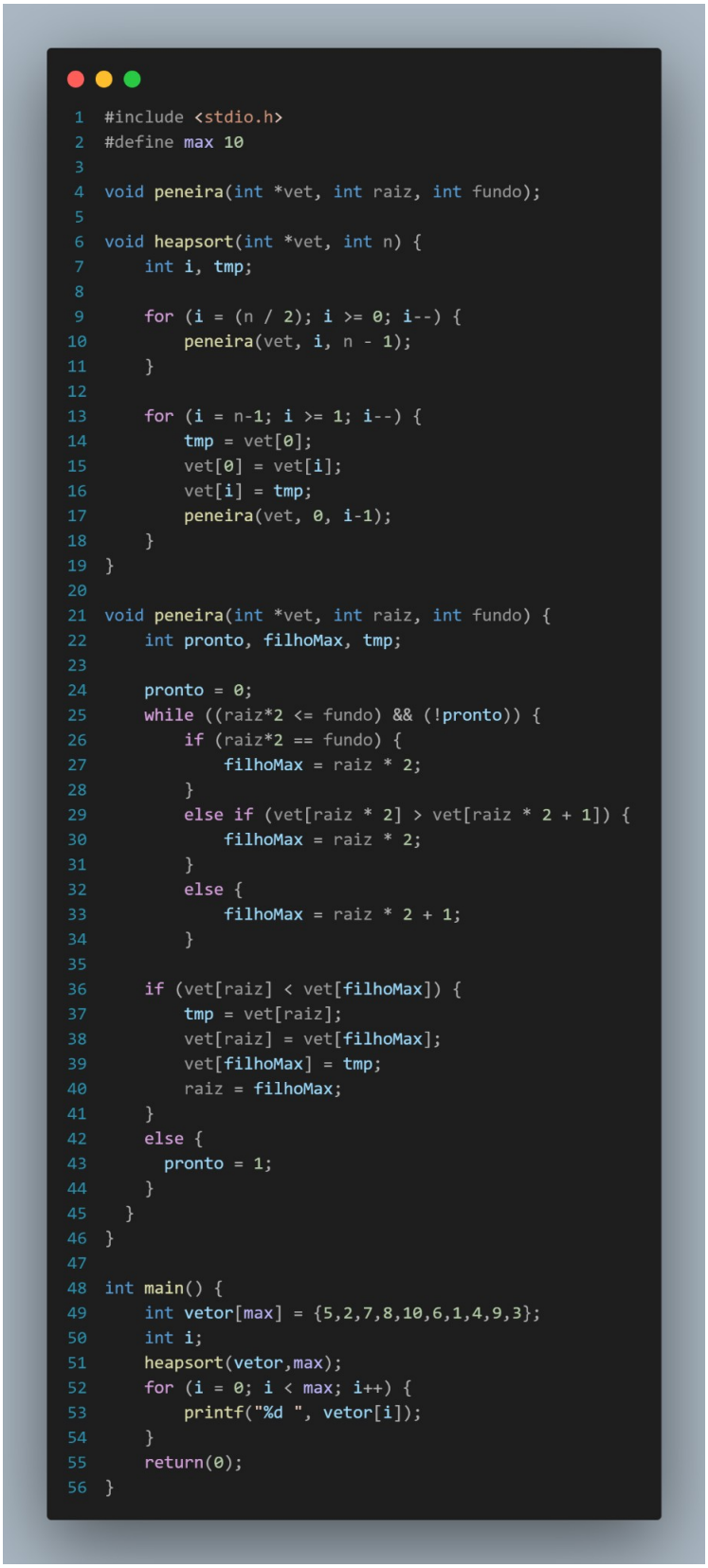
O Heapsort é um algoritmo de ordenação que se baseia na estrutura de dados conhecida como *heap*. Um *heap* é uma árvore binária especial em que cada nó possui um valor maior ou igual ao valor de seus filhos (se for um *heap* máximo) ou menor ou igual (se for um *heap* mínimo). A chave para o sucesso do *Heapsort* está na construção de um *heap* máximo a partir dos elementos do *array* que será ordenado.

O algoritmo Heapsort segue os seguintes passos:

1. Construção do Heap Máximo: A primeira etapa envolve a transformação do array desordenado em um heap máximo. Isso é feito começando pelo último elemento do array e movendo-se em direção ao primeiro. A cada passo, o elemento é colocado na posição correta para que o heap máximo seja mantido.

2. Ordenação: Uma vez que o heap máximo tenha sido construído, o maior elemento estará sempre na raiz da árvore (posição 0 do array). Esse elemento é trocado com o último elemento do array, que agora está na posição correta. Em seguida, o heap é reconstruído sem o último elemento, que já está na posição correta. Esse processo é repetido até que todos os elementos estejam na posição correta.

Algoritmo:



```c
#include <stdio.h>
#define max 10

void peneira(int *vet, int raiz, int fundo);

void heapsort(int *vet, int n) {
    int i, tmp;

    for (i = (n / 2); i >= 0; i--) {
        peneira(vet, i, n - 1);
    }

    for (i = n-1; i >= 1; i--) {
        tmp = vet[0];
        vet[0] = vet[i];
        vet[i] = tmp;
        peneira(vet, 0, i-1);
    }
}

void peneira(int *vet, int raiz, int fundo) {
    int pronto, filhoMax, tmp;

    pronto = 0;
    while ((raiz*2 <= fundo) && (!pronto)) {
        if (raiz*2 == fundo) {
            filhoMax = raiz * 2;
        }
        else if (vet[raiz * 2] > vet[raiz * 2 + 1]) {
            filhoMax = raiz * 2;
        }
        else {
            filhoMax = raiz * 2 + 1;
        }

    if (vet[raiz] < vet[filhoMax]) {
        tmp = vet[raiz];
        vet[raiz] = vet[filhoMax];
        vet[filhoMax] = tmp;
        raiz = filhoMax;
    }
    else {
      pronto = 1;
    }
   }
}

int main() {
    int vetor[max] = {5,2,7,8,10,6,1,4,9,3};
    int i;
    heapsort(vetor,max);
    for (i = 0; i < max; i++) {
        printf("%d ", vetor[i]);
    }
    return(0);
}
```

Vantagens do Heap Sort

O heap sort oferece vantagens significativas em determinados cenários:

- Eficiência em Grandes Conjuntos de Dados: Sua complexidade de tempo O(n log n) o torna eficiente para ordenar grandes quantidades de dados.
- Uso em Aplicações de Baixo Nível: O heap sort encontra aplicações em sistemas operacionais e algoritmos de gerenciamento de memória, onde a eficiência é crucial.

Desvantagens do Heap Sort

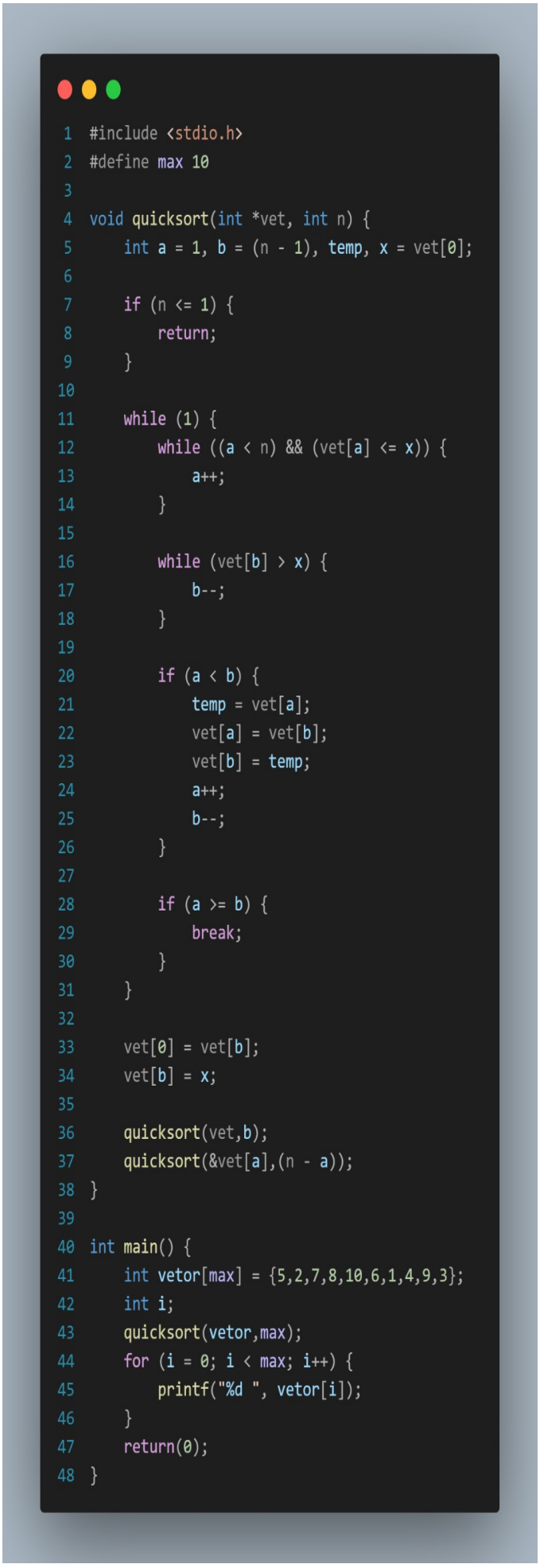
No entanto, o heap sort também tem suas desvantagens:

- Falta de Estabilidade: Ele não preserva a ordem relativa de elementos com valores iguais, tornando-o inadequado em situações onde a estabilidade é necessária.
- Complexidade de Implementação: A implementação do heap sort é mais complexa em comparação com alguns outros algoritmos de ordenação, como o bubble sort, o insertion sort e o selection sort.

Quick Sort

Quicksort é um algoritmo de ordenação eficiente e amplamente utilizado na ciência da computação. Desenvolvido por Tony Hoare em 1960, ele utiliza a técnica de divisão e conquista para ordenar elementos em uma lista. O algoritmo funciona escolhendo um ‘pivô’ e particionando os elementos em duas sublistas: uma contendo elementos menores que o pivô e outra com elementos maiores. Esse processo é repetido recursivamente até que a lista esteja completamente ordenada.

Algoritmo:



```c
#include <stdio.h>
#define max 10

void quicksort(int *vet, int n) {
    int a = 1, b = (n - 1), temp, x = vet[0];

    if (n <= 1) {
        return;
    }

    while (1) {
        while ((a < n) && (vet[a] <= x)) {
            a++;
        }

        while (vet[b] > x) {
            b--;
        }

        if (a < b) {
            temp = vet[a];
            vet[a] = vet[b];
            vet[b] = temp;
            a++;
            b--;
        }

        if (a >= b) {
            break;
        }
    }

    vet[0] = vet[b];
    vet[b] = x;

    quicksort(vet,b);
    quicksort(&vet[a],(n - a));
}

int main() {
    int vetor[max] = {5,2,7,8,10,6,1,4,9,3};
    int i;
    quicksort(vetor,max);
    for (i = 0; i < max; i++) {
        printf("%d ", vetor[i]);
    }
    return(0);
}
```

Vantagens do Quick Sort

Uma das principais vantagens do Quicksort é sua eficiência. Em média, ele possui uma complexidade de tempo de O(n log n), o que o torna mais rápido do que outros algoritmos de ordenação, como o Bubble Sort ou o Insertion Sort. Além disso, o Quicksort é um algoritmo in-place, o que significa que ele não requer espaço adicional significativo para a ordenação, tornando-o ideal para grandes conjuntos de dados.

Desvantagens do Quick Sort

Apesar de suas vantagens, o Quicksort também possui desvantagens. Em casos extremos, como quando a lista já está ordenada ou quase ordenada, sua complexidade de tempo pode degradar para $O(n^2)$. Para mitigar esse problema, técnicas como a escolha aleatória do pivô ou a implementação de uma versão híbrida que utiliza outro algoritmo para listas pequenas podem ser empregadas.

Busca Sequencial e Busca Binária

A busca sequencial compara cada posição da estrutura em busca do elemento requisitado onde ele, para caso atinja ao final da array ou encontre o elemento procurado.

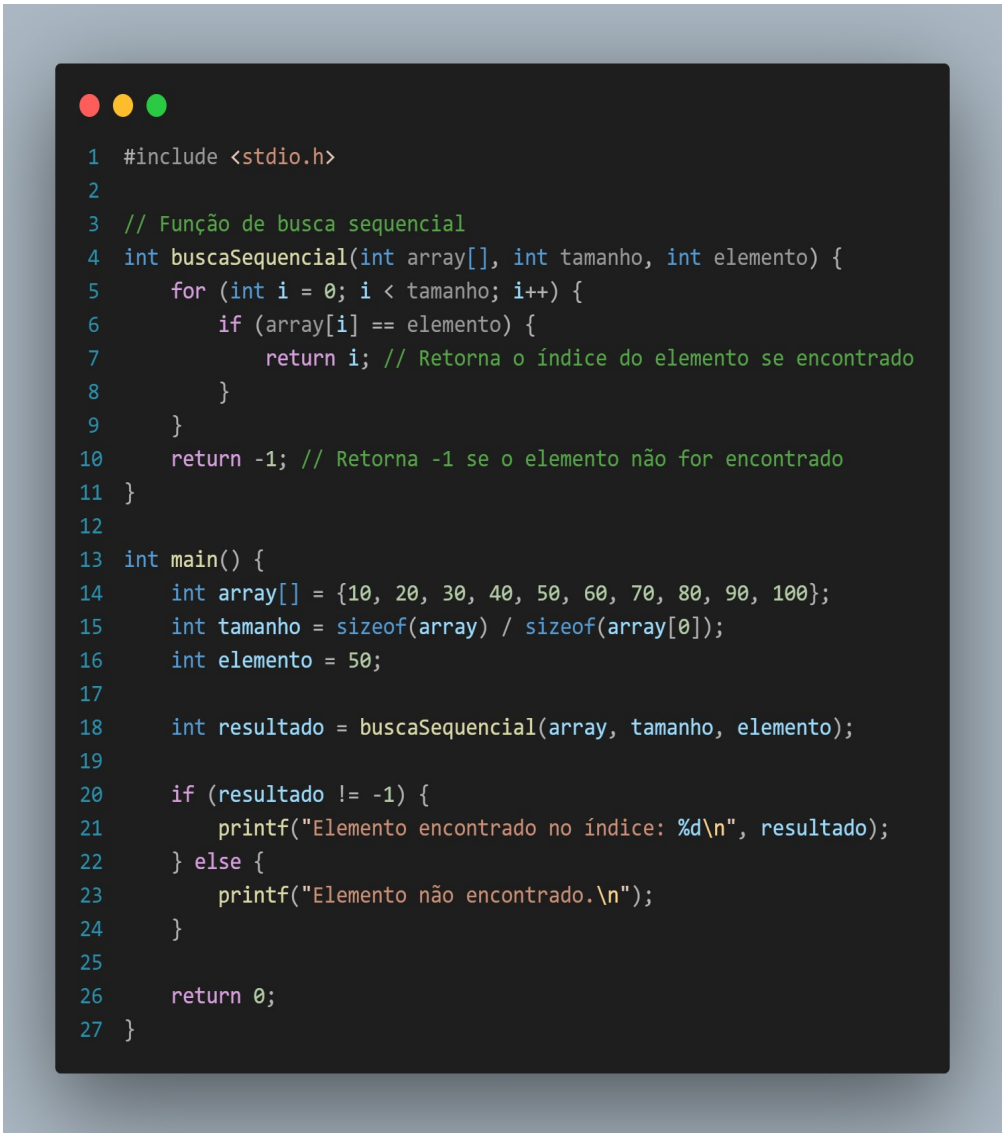
Modo de Funcionamento: A função da busca recebe um array a quantidade n de elementos desse array e o elemento que deseja encontrar. Então vai comparar o elemento com cada posição da lista e caso seja igual irá retornar ele, caso atinja ao final da lista o algoritmo vai retornar que não encontrou o elemento.

A Busca Binária é um algoritmo eficiente para encontrar um elemento em um array ordenado. Diferente da busca linear, que examina cada elemento sequencialmente, a busca binária reduz pela metade o espaço de busca a cada iteração, tornando-a muito mais eficiente para grandes conjuntos de dados.

O funcionamento da busca binária baseia-se na divisão do array em duas metades e na comparação do elemento do meio com o valor desejado. Se o valor buscado for menor que o elemento do meio, a busca continua na metade inferior do array, se for maior, continua na

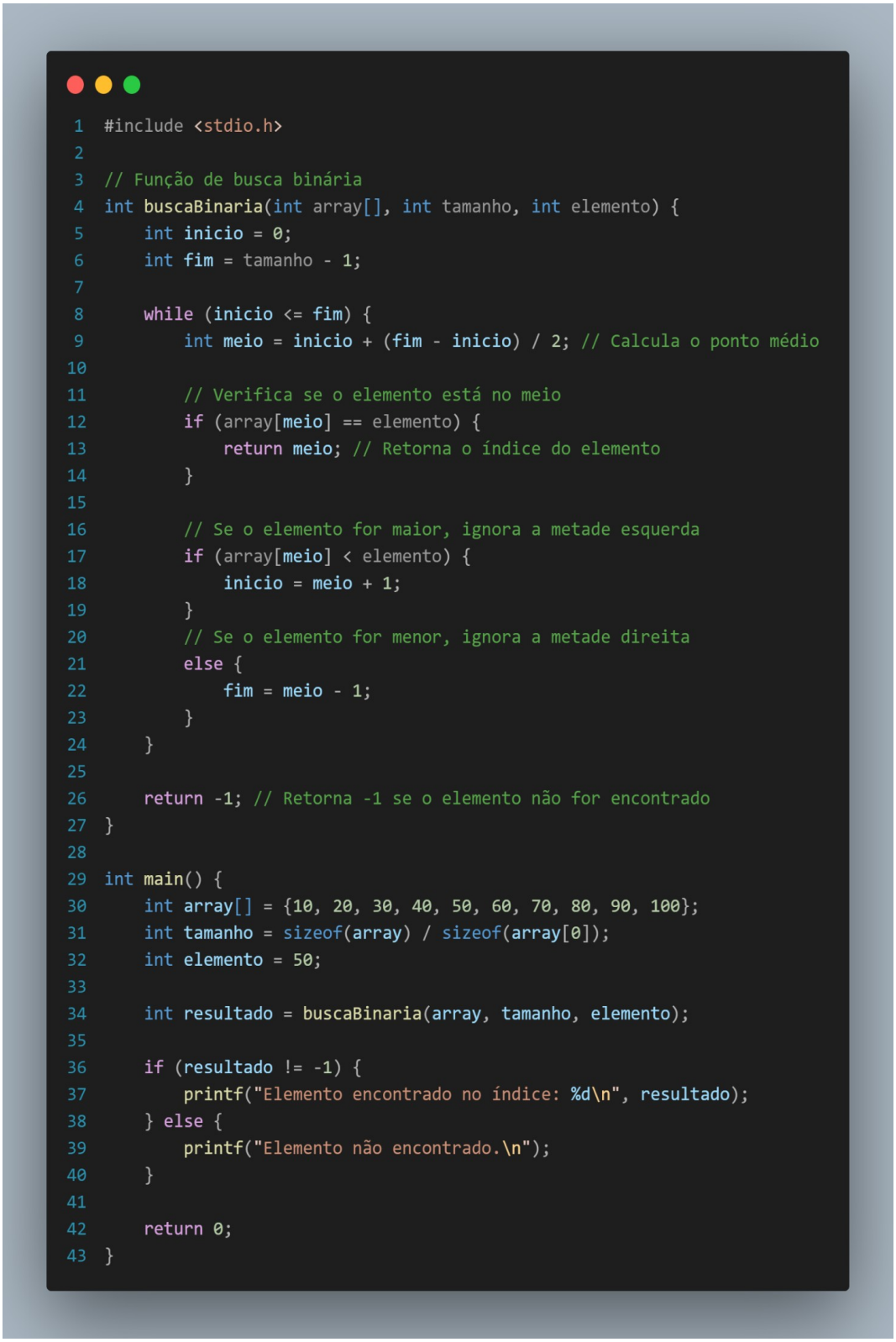
metade superior. Esse processo se repete até que o elemento seja encontrado ou até que o subarray tenha tamanho zero, indicando que o elemento não está presente.

Algoritmo Busca Sequencial:



```c
#include <stdio.h>

// Função de busca sequencial
int buscaSequencial(int array[], int tamanho, int elemento) {
    for (int i = 0; i < tamanho; i++) {
        if (array[i] == elemento) {
            return i; // Retorna o índice do elemento se encontrado
        }
    }
    return -1; // Retorna -1 se o elemento não for encontrado
}

int main() {
    int array[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int tamanho = sizeof(array) / sizeof(array[0]);
    int elemento = 50;

    int resultado = buscaSequencial(array, tamanho, elemento);

    if (resultado != -1) {
        printf("Elemento encontrado no índice: %d\n", resultado);
    } else {
        printf("Elemento não encontrado.\n");
    }

    return 0;
}
```

Algoritmo Busca Binária:



```c
#include <stdio.h>

// Função de busca binária
int buscaBinaria(int array[], int tamanho, int elemento) {
    int inicio = 0;
    int fim = tamanho - 1;

    while (inicio <= fim) {
        int meio = inicio + (fim - inicio) / 2; // Calcula o ponto médio

        // Verifica se o elemento está no meio
        if (array[meio] == elemento) {
            return meio; // Retorna o índice do elemento
        }

        // Se o elemento for maior, ignora a metade esquerda
        if (array[meio] < elemento) {
            inicio = meio + 1;
        }
        // Se o elemento for menor, ignora a metade direita
        else {
            fim = meio - 1;
        }
    }

    return -1; // Retorna -1 se o elemento não for encontrado
}

int main() {
    int array[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int tamanho = sizeof(array) / sizeof(array[0]);
    int elemento = 50;

    int resultado = buscaBinaria(array, tamanho, elemento);

    if (resultado != -1) {
        printf("Elemento encontrado no índice: %d\n", resultado);
    } else {
        printf("Elemento não encontrado.\n");
    }

    return 0;
}
```

Vantagens Busca Sequencial:

- Não necessita de ordenação prévia para funcionar

- Simples para implementar
- Eficiente para listas pequenas

Desvantagens Busca Sequencial:

- É ineficiente para conjuntos de arrays muito grandes
- No pior caso percorre todos os elementos da lista

Vantagens Busca Binária:

- Eficiência: A busca binária tem complexidade O(log n), o que a torna muito mais rápida do que a busca linear O(n) para grandes conjuntos de dados.
- Redução de Comparações: O algoritmo reduz o número de comparações em cada iteração, tornando a busca mais eficiente. Facilidade de Implementação: O algoritmo é simples de implementar e entender

Desvantagens Busca Binária:

- Requer Ordenação Prévia: Para que a busca binária funcione corretamente, o array precisa estar ordenado, o que pode adicionar um custo extra se a ordenação ainda não tiver sido feita.
- Dificuldade de Uso com Listas Dinâmicas: Em estruturas dinâmicas como listas encadeadas, a busca binária perde eficiência, pois o acesso a elementos no meio da lista é mais lento.

Referências

https://www.dio.me/articles/insertion-sort-estrutura-de-dados

https://www.tutorialspoint.com/data_structures_algorithms/selection_sort_algorithm.htm

https://mundobitabitblog.wordpress.com/2017/07/03/101/

https://elemarjr.com/clube-de-estudos/artigos/o-que-e-e-como-funciona-o-bubblesort/

https://luizcalazans.medium.com/entendendo-o-heapsort-95ec851dcdbf

https://www.w3schools.com/dsa/dsa_algo_radixsort.php
https://programae.org.br/termos/glossario/o-que-e-quicksort-e-para-que-serve/
https://www.w3schools.com/dsa/dsa_algo_mergesort.php