UNIVERSIDADE FEDERAL DE RORAIMA
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
CURSO BACHAREL EM CIÊNCIA DA COMPUTAÇÃO
ESTRUTURA DE DADOS I

ANDREZA OLIVEIRA GONÇALVES
JONATHAN EMERSON BRAGA DA SILVA

ALGORITMOS DE ORDENAÇÃO

BOA VISTA – RR
2025

ANDREZA OLIVEIRA GONÇALVES
JONATHAN EMERSON BRAGA DA SILVA

ALGORITMOS DE ORDENAÇÃO

Relatório Científico apresentado ao Prof. Filipe Dwan Pereira, com objetivo de obtenção de nota parcial para aprovação na disciplina DCC 302 - Estrutura de Dados I, do Departamento de Ciência da Computação da Universidade Federal de Roraima.

BOA VISTA – RR
2025

1 INTRODUÇÃO

Algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem — em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica. Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade de se acessar seus dados de modo mais eficiente.
Entre os mais importantes, podemos citar bubble sort (ou ordenação por flutuação), insertion sort (ou ordenação por inserção), merge sort (ou ordenação por mistura) e o quick sort. Existem ainda outros tipos, como o selection sort, heap sort e radix sort, além de mecanismos de busca, como busca sequencial e busca binária, os quais exploraremos neste trabalho.

2 SELECTION SORT

Selection Sort (Ordenação por Seleção) trata-se de um algoritmo de ordenação simples e intuitivo. Funciona selecionado repetidamente o menor (ou maior, dependendo da ordem aplicada) elemento da lista não ordenada, trocando-o com o primeiro elemento da lista. Encontra o menor elemento e o coloca na posição correta. Processo esse que se repete até que toda lista esteja de fato ordenada. Complexidade de tempo $O(n^2)$.

Algoritmo

```
// Função do Selection Sort
int selection_sort(int lista[], int n) {
   int i, j, aux, menor_indice;

   for (i = 0; i < n - 1; i++) {
      menor_indice = i;
      // Encontra o menor elemento na parte não ordenada
      for (j = i + 1; j < n; j++) {
            if (lista[j] < lista[menor_indice])
            menor_indice = j;
      }
      // Troca os elementos
      aux = lista[i];
      lista[i] = lista[menor_indice];
      lista[menor_indice] = aux;
   }
}
```

Vantagens:

- Simplicidade: O algoritmo é fácil de entender e implementar.
- Pouco uso de memória: Por ser um algoritmo *in-place*, não precisa de espaço adicional, já que usa o mesmo espaço de entrada.
- Número reduzido de trocas: Realiza no máximo n-1 trocas, o que pode ser vantajoso em cenários onde trocar elementos é custoso.

Desvantagens:

- Ineficiência em listas grandes: Tem complexidade de tempo O(n²), o que o torna ineficiente para listas grandes.
- Não adaptativo: O tempo de execução não melhora significativamente se a lista já estiver parcialmente ordenada.
- Instabilidade: Não é um algoritmo estável, já que ele pode alterar a ordem relativa de elementos iguais.

3 INSERTION SORT

Insertion Sort (Ordenação por Inserção) é um algoritmo de ordenação simples e eficiente para conjuntos de dados pequenos ou parcialmente ordenados. Ele funciona de maneira semelhante à forma como muitas pessoas organizam cartas em um jogo de baralho: pega-se cada elemento e insere-o na posição correta dentro da parte já ordenada do array. Complexidade de tempo O(n²).

Algoritmo:

```
// Função do Insertion Sort
void insertion_sort(int lista[], int n) {
   int i, j, aux;

   for (i = 1; i < n; i++) {
      aux = lista[i];
      j = i - 1;

      // Move os elementos da parte ordenada que são maiores que o auxiliar
      while (j >= 0 && lista[j] > aux) {
         lista[j + 1] = lista[j];
         j--;
      }
      // Insere o elemento na posição correta
      lista[j + 1] = aux;
   }
}
```

Vantagens:

- Eficiência para pequenos conjuntos: Para arrays pequenos ou quase ordenados, o Insertion Sort pode ser mais eficiente do que algoritmos mais complexos, como o próprio Quick Sort e Merge Sort.
- Estabilidade: O Insertion é um algoritmo estável, ou seja, ele mantém a ordem relativa de elementos iguais.
- Ordenação in-place: O algoritmo não requer memória adicional significativa, pois ele ordena os elementos no próprio array.

Desvantagens:

- Ineficiência para grandes conjuntos: Por ter complexidade de tempo O(n²) no pior caso, o que o torna ineficiente para grandes volumes de dados.
- Pouco escalável: À medida que o tamanho do array aumenta, o desempenho do algoritmo diminui significativamente.
- Comparações e trocar excessivas: Em arrays desordenados, o algoritmo realiza muitas comparações e trocas, o que pode ser custoso.

4 BUBBLE SORT

O Bubble Sort compara elementos adjacentes e os troca se estiverem fora de ordem, repetindo o processo até que a lista esteja ordenada. Em cada passagem, são considerados os elementos restantes e são comparados com seus adjacentes, realizando-se a troca caso o elemento maior esteja antes de um elemento menor. Assim, é obtido o maior elemento dentre os restantes em sua posição correta. Complexidade de tempo O(n²).

Algoritmo:

```
// Função do Bubble Sort
void bubble_sort(int lista[], int n) {
    int i, j, aux;

    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - 1 - i; j++) {
            if (lista[j] > lista[j + 1]) {
                // Troca os elementos se estiverem fora de ordem
                aux = lista[j];
                lista[j] = lista[j + 1];
                lista[j + 1] = aux;
            }
        }
    }
}
```

Vantagens

- Simples de entender e implementar.
- Estável (não altera a ordem de elementos iguais).

Desvantagens

- A classificação por bolhas tem uma complexidade de tempo de O(n²), o que a torna muito lenta para grandes conjuntos de dados.
- O bubble sort tem quase nenhuma ou limitada aplicação no mundo real.

5 HEAP SORT

Heap Sort é um algoritmo de ordenação baseado na estrutura de dados Heap. Ele transforma o array em uma árvore Heap (máximo ou mínimo) e extrai os elementos em ordem.

Algoritmo:

```
void heapify(int arr[], int n, int i) {
    int maior = i;
    int esq = 2 * i + 1;
    int dir = 2 * i + 2;

    if (esq < n && arr[esq] > arr[maior])
        maior = esq;

    if (dir < n && arr[dir] > arr[maior])
        maior = dir;

    if (maior != i) {
        int temp = arr[i];
        arr[i] = arr[maior];
        arr[maior] = temp;
        heapify(arr, n, maior);
    }
}
// Função principal do Heap Sort
void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    for (int i = n - 1; i > 0; i--) {
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}
```

Vantagens

- Complexidade O(n log n) no pior caso.
- É um algoritmo In-place (não requer memória extra significativa).
- Boa escolha para grandes conjuntos.

Desvantagens

- Não é estável, pois pode alterar a ordem de elementos iguais.
- Mais difícil de implementar.

6 RADIX SORT

Radix sort é um algoritmo de ordenação estável que realiza a ordenação dos números com base em seus dígitos individualmente. Essa ordenação pode ocorrer de forma LSD (Least Significant Digit - Dígito Menos Significativo) ou MSD (Most Significant Digit - Dígito Mais Significativo). Pode ser implementado utilizando a ideia de Buckets, como “caixas” temporárias para armazenar os números durante a ordenação. Ordenação baseada em dígitos com complexidade de tempo O(nk), onde n é o número de elementos e k é o número de dígitos do maior número.

Algoritmo:

```
void radix(int *vet, int tam, int maior){
   Int exp = 1;
      while(abs(maior / exp) > 0) {
         int count[10] = [0]; // Contador
         int  aux[tam];

   for (int i - 0; i < tam; i++) { // Etapa 1
      // Pega o dígito da vez
      int digito = vet[i] / exp;
      digito = digito % 10;

      count[digito]++;
   }

   for (int i = tam - 1; i >= 0; i++) { // Etapa 2
      count[i] = count[i] + count[i - 1];
   }

   for (int i = tam - 1; i >= 0; i–) { // Etapa 3
      int digito = vet[i] /exp;
      digito = digito % 10;

      count[digito]--;
      aux[count[digito]] = vet[1];
   }

   for (int i = 0; i < tam; i++) { //Etapa 4
```

```
        vet[i] = aux[i];
    }
    exp = exp*10;
    }
}
```

Vantagens:

- Complexidade Linear: Pode ter um desempenho muito eficiente, com tempo de execução O(nk), onde n é o número de elementos a serem ordenados. K é o número de dígitos do maior número.
- Bom para grandes volumes de dados: Por não fazer comparação entre elementos, sua performance é estável em grandes conjuntos de dados.

Desvantagens:

- Uso elevado de memória: Precisa de listas auxiliares (geralmente de base 10, para os dígitos de 0 a 9) e pode consumir bastante espaço, especialmente se houver muitos elementos.
- Depende do tamanho do maior número: O tempo de execução depende da quantidade do maior número k, o que pode ser problemático para números muitos grandes.

7 QUICK SORT

Conhecido pela sua eficiência e simplicidade, usando a estratégia de “Divisão e Conquista” para ordenar elementos, escolhendo um pivô e particionando o array em partes menores. Até hoje sendo um dos algoritmos de ordenação mais utilizados devido ao seu desempenho superior na maioria dos casos. Complexidade de tempo O(n log n).

Algoritmo

```
//Função para trocar dois elementos no array
void troca(int *a, int *b){
    int aux = *a;
    *a = *b;
    *b = aux;
}

//Função de particionamento (Lomuto Partition)
int particionar(int lista[], int inicio int fim){
    int pivo = lista[fim];  //Escolhe o último elemento como pivô
    int i  = inicio - 1;                // Índice dos elementos menores que o pivô

    for (int j = inicio; j < fim; j++) {
        if (lista[j] < pivo) {
        i++;
```

```
        troca(&lista[i], &lista[j];
        }
    }

    troca(&lista[i + 1], &lista[fim];          //coloca o pivô na posição correta
    return i + 1;
}

//Função principal do Quick Sort (recursiva)
void quick_sort(int lista[], int inicio, int fim) {
    if (inicio < fim) {
        int pivo = particionar(lista, inicio, fim);

        // Chamadas recursivas para ordenar as duas metades
        quick_sort(lista, inicio, pivo - 1);
        quick_sort(lista, pivo + 1, fim);
    }
}
```

Vantagens:

- Geralmente mais rápido que Merge Sort e Heap Sort.
- Ordenação in-place (usa pouca memória adicional).
- Desempenho eficiente em conjuntos de dados grandes.

Desvantagens:

- Pode ter desempenho ruim no pior caso ($O(n^2)$).
- Não é estável, pois pode alterar a ordem de elementos iguais.

8 MERGE SORT

Merge Sort é um algoritmo de ordenação baseado na estratégia "Dividir para Conquistar". Ele divide recursivamente o array em subarrays menores até que cada um tenha apenas um elemento e, em seguida, os mescla de forma ordenada. Divide o conjunto ao meio, ordena cada parte e as une, com complexidade de tempo O(n log n).

Algoritmo:

```
// Função principal do Merge Sort (recursiva)
void merge_sort(int lista[], int inicio, int fim) {
    if (inicio < fim) {
        int meio = inicio + (fim - inicio) / 2;

        merge_sort(lista, inicio, meio);
        merge_sort(lista, meio + 1, fim);
```

```
        mesclar(lista, inicio, meio, fim);
    }
}
```

9 BUSCA SEQUENCIAL E BINÁRIA

A **busca sequencial** é o algoritmo mais simples de pesquisa em um conjunto de dados. Ele percorre os elementos de uma estrutura (vetor, lista, etc.) um por um até encontrar o valor desejado ou chegar ao fim da estrutura.

Algoritmo:

```
int busca_sequencial(int lista[], int n, int chave) {
    for (int i = 0; i < n; i++) {
        if (lista[i] == chave)
            return i; // Retorna a posição do elemento
    }
    return -1; // Elemento não encontrado
}
```

Vantagens

- Simplicidade de implementação.
- Funciona em qualquer estrutura de dados linear (listas encadeadas, vetores, etc.).
- Não requer que os elementos estejam ordenados.

Desvantagens

- Ineficiente para grandes conjuntos de dados (complexidade O(n)).
- Pode ser muito lento se o elemento procurado estiver no final ou não existir.

A **busca binária** é um método eficiente para encontrar um elemento em um vetor ordenado. O algoritmo divide repetidamente o espaço de busca pela metade até encontrar o elemento desejado ou concluir que ele não está presente.

Algoritmo:

```
int busca_binaria(int lista[], int esq, int dir, int chave) {
    while (esq <= dir) {
        int meio = esq + (dir - esq) / 2;
        if (lista[meio] == chave) return meio;
        if (lista[meio] < chave) esq = meio + 1;
        else dir = meio - 1;
    }
    return -1; // Elemento não encontrado
}
```

Vantagens

- Muito mais rápido do que a busca sequencial em conjuntos grandes (O(log n)).
- Ideal para estruturas ordenadas.

Desvantagens

- Requer que os dados estejam ordenados.
- Não funciona bem para listas encadeadas, pois não permite acesso direto ao meio do conjunto.

10 REFERÊNCIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*.
Sedgewick, R. (2002). *Algorithms in C, Parts 1-5*.
Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in C*.
SEDGEWICK, Robert; WAYNE, Kevin. Algorithms. 4. ed. Boston: Addison-Wesley, 2011.
LEVITIN, Anany. Introduction to the Design and Analysis of Algorithms. 3. ed. Boston: Pearson, 2011.
HEBERT, Bruno L. de Castro. Algoritmos de Ordenação. Universidade Federal de Goiás, 2020. Disponível em: https://ww2.inf.ufg.br/~hebert/disc/aed1/AED1_04_ordenacao1.pdf. Acesso em: 7 fev. 2025.
GEEKS FOR GEEKS. Selection Sort Algorithm. 2024. Disponível em: https://www.geeksforgeeks.org/selection-sort-algorithm-2/. Acesso em: 7 fev. 2025.