**UNASP – CENTRO UNIVERSITÁRIO ADVENTISTA DE**

**SÃO PAULO**

ENGENHARIA DE COMPUTAÇÃO

São Paulo, dezembro de 2024.

Aluno: Isaac Brito Bressan

RA: 197490

Disciplina: Estrutura de Dados

**Lista de Atividade - Reforço de Estudo**

**Link do Repositório no Github:**

<https://github.com/isaacbre/lista-de-algoritmos-de-busca-e-ordena-o>

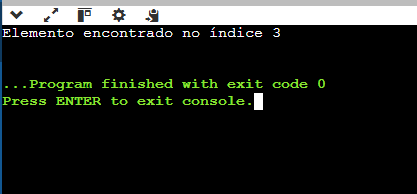
### **1. Binary Search**

### **O que é:** A Pesquisa Binária é um método eficiente para localizar elementos em listas ordenadas. Ele começa verificando o elemento central da lista e comparando-o com o valor alvo. Se o alvo for menor, a busca continua na metade esquerda; se maior, na metade direita.

### **Necessidade de ordenação:** Uma lista deve ser organizada para que a divisão e busca sejam efetivadas. Caso contrário, não é possível confiar na comparação com o elemento central, inviabilizando o método.

### **Exemplo:** Para localizar o número 12 na lista ordenada [3, 6, 8, 12, 15, 18, 22], comparamos 12 ao elemento central (8). Como 12 é maior, analisamos apenas a parte direita. No próximo passo, localizado 12**.**

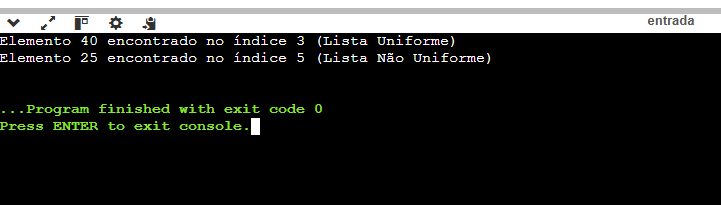
Saída do código



**2. Interpolation Search**

### **Eficiência**: O Interpolation Search pode ser mais eficiente que o Binary Search quando os dados são uniformemente distribuídos, já que ele pode "pular" diretamente para o valor desejado. Contudo, se os dados não são uniformemente distribuídos, pode ser menos eficiente que o Binary Search.

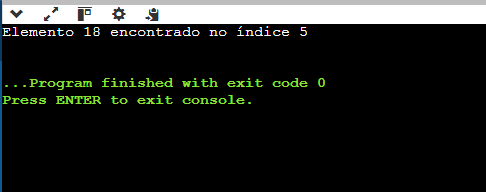
Saída do código



### **3. Jump Search**

**Comparação com o Binary Search**: O Jump Search pode ser mais eficiente que o Binary Search em listas de tamanho médio, especialmente se o número de saltos for reduzido, mas o Binary Search tende a ser mais eficiente em listas menores ou ordenadas de maneira mais uniforme. Portanto, o **Binary Search** é sempre mais eficiente do que o **Jump Search** à medida que o tamanho da lista cresce significativamente. O **Jump Search** tem uma vantagem apenas em listas menores e de tamanho médio, mas sua complexidade de O(√n)O(√n)O(√n) não escala bem para listas grandes.

Saída do código

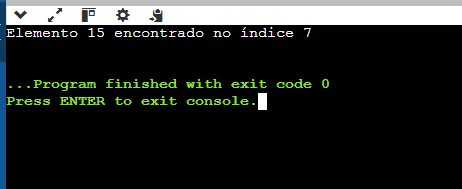


### **4. Exponential Search**

**Combinação com Jump Search e Binary Search**: O Exponential Search é eficiente em listas grandes, pois encontra rapidamente o intervalo onde o alvo pode estar, e depois utiliza o Binary Search para localizá-lo.

**Desempenho em lista**: Em listas pequenas, o Exponential Search pode ser menos eficiente que o Binary Search, mas em listas muito grandes, sua abordagem exponencial inicial pode reduzir o número de comparações.

Saída do código



### **5. Shell Sort**

**Sequências de intervalo**: As sequências de intervalo afetam a eficiência do algoritmo. As mais comuns são:

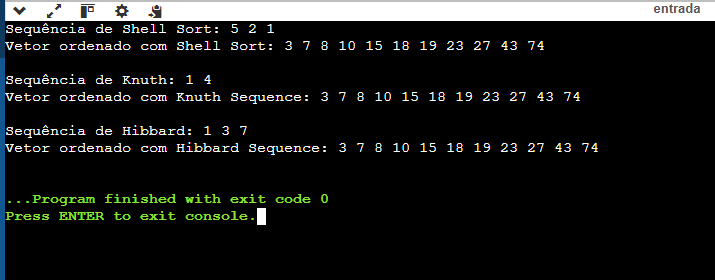
**Shell**: Gaps fixos que diminuem até 1.

**Knuth**: Usa gaps com a fórmula 3^k - 1.

**Hibbard**: Usa gaps como potências de 2 menos 1.

**Eficiência**: A escolha de sequência de intervalo afeta o desempenho; por exemplo, a sequência de Hibbard é mais eficiente que a de Shell.

Saída do código

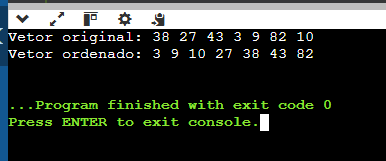


### **6. Merge Sort**

**Conceito de "dividir para conquistar"**: Divide-se a lista em duas metades, resolve-se cada metade (ordenação) e depois mescla as duas metades ordenadas. Isso é feito recursivamente.

**Modificação para strings**: Ao ordenar strings, o algoritmo funciona da mesma maneira, mas compara as strings lexicograficamente (como em um dicionário).

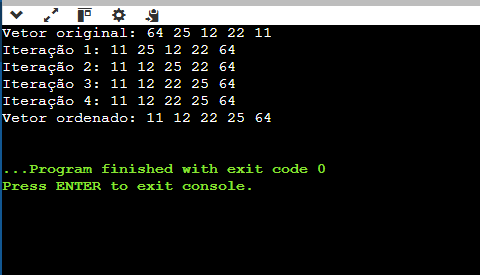
Saída do código



### **7. Selection Sort**

**Desempenho em diferentes tamanhos**: Embora simples, o Selection Sort não é eficiente em listas grandes, já que seu tempo de execução é quadrático (O(n^2)), mas pode ser útil em listas pequenas.

Saída do código

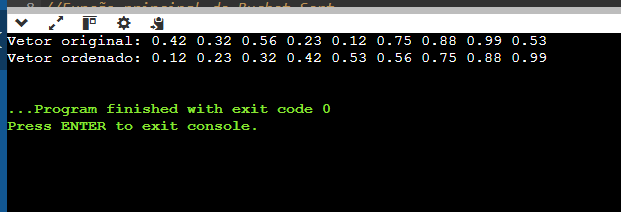


### **8. Bucket Sort**

**Conceito**: O Bucket Sort ordena uma lista de números em ponto flutuante no intervalo [0, 1) dividindo-os em "baldes", cada um contendo uma parte dos dados. Cada balde é ordenada separadamente e depois os baldes são combinados.

**Adaptação para números inteiros**: O Bucket Sort pode ser adaptado para ordenar números inteiros, criando baldes de acordo com intervalos apropriados.

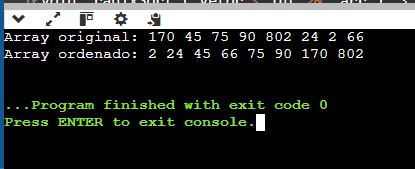
Saída do código



### **9. Radix Sort**

**Exemplo de bases**: Em base 10, o Radix Sort lida com os dígitos decimais de cada número, enquanto na base 2, lida com os dígitos binários. Cada dígito é tratado como uma chave para ordenação.

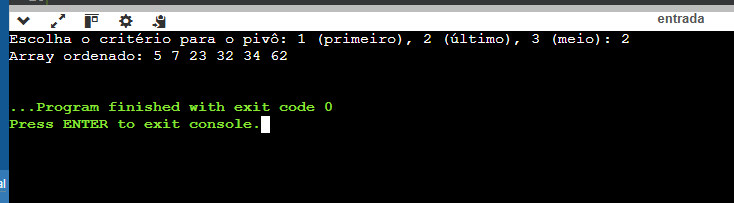
Saída do código



### **10. Quick Sort**

**Desempenho com diferentes escolhas de pivô**: O desempenho do Quick Sort pode ser afetado pela escolha do pivô. A escolha ideal pode ser o pivô mediano ou um pivô aleatório, mas em listas quase ordenadas, o algoritmo pode ter desempenho pior.

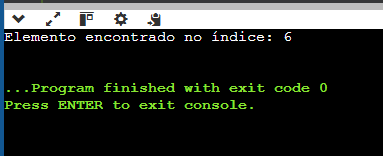
Saída do código



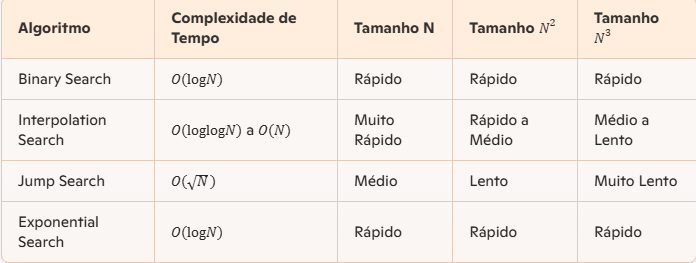
### **11. Ternary Search**

**Comparação com o Binary Search**: O Binary Search é mais simples e geralmente mais eficiente em listas pequenas, enquanto o Ternary Search pode ser útil em listas muito grandes, embora seja mais complexo de implementar.

Saída do código

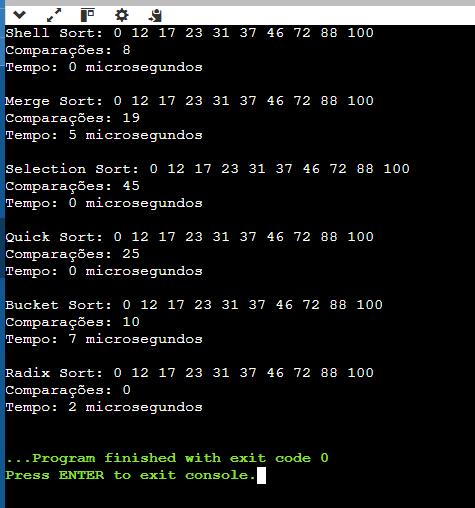


**12. Comparação de Algoritmos de Busca**



### **13. Comparação de Algoritmos de Ordenação**

### Saída do código



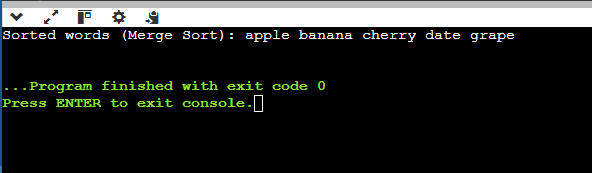
### **14. Análise de Complexidade**

**Busca**: A maioria dos algoritmos de busca (exceto a Pesquisa por Interpolação em casos não uniformemente distribuídos) tem complexidade *O(log n)* e todos têm complexidade de espaço *O(1).*

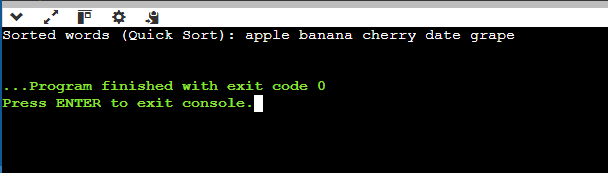
**Ordenação**: Entre os algoritmos de ordenação, **Merge Sort**, **Quick Sort** e **Radix Sort** possuem complexidade *Sobre registro n*, sendo os mais eficientes. **Shell Sort** pode ter desempenho variável, enquanto **Selection Sort** tem desempenho quadrático *Sobre n\*2.* O **Bucket Sort** depende da distribuição dos elementos, podendo ser muito eficiente em alguns casos.

### **15. Busca e Ordenação em Strings**

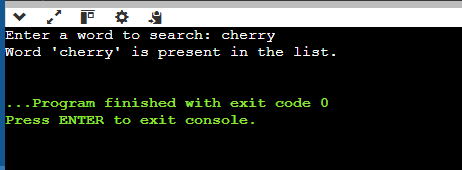
Saída do código Merge Sort



Saída do código Quick Sort



Saída do código com o Binary Search



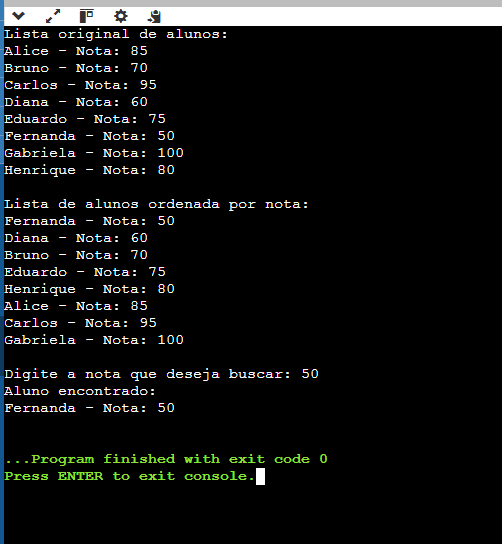
### **16. Aplicação Prática de Busca**

Saída do código

### 

### **17. Busca e Ordenação em Dados Reais**

Saída do código

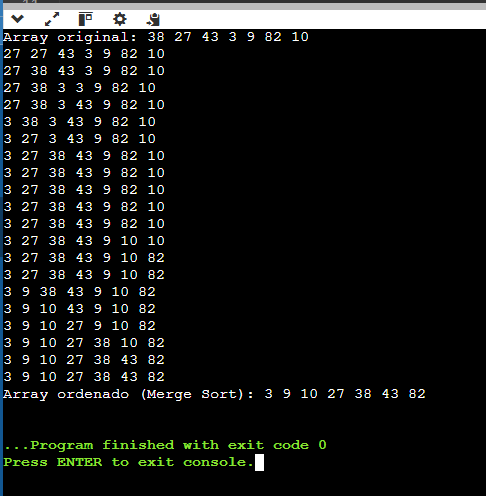


### **18. Ordenação Estável e Instável**

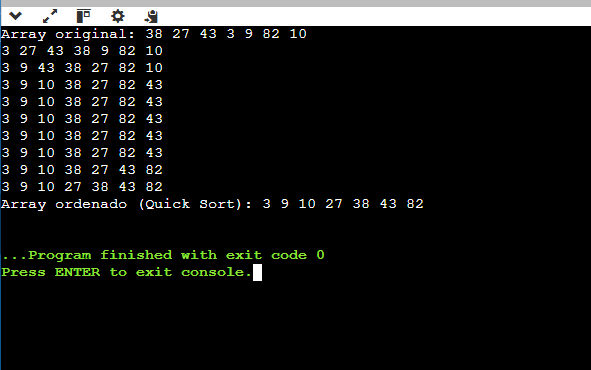
Algoritmos de ordenação podem ser classificados como resultados ou instáveis. Quando dizemos que um algoritmo é estável, significa que ele mantém a ordem dos elementos iguais como estavam na lista original, o que é importante quando precisamos ordenar os dados por múltiplos critérios. Por exemplo, no Merge Sort, se dois elementos têm o mesmo valor, eles permanecem na mesma ordem. Já no Quick Sort ou no Selection Sort, a ordem desses elementos pode ser alterada durante a ordenação. Algoritmos como Merge Sort, Bucket Sort e Radix Sort são resultados, enquanto Shell Sort, Selection Sort e Quick Sort são instáveis. A escolha entre um algoritmo estável ou instável vai depender de quão importante é preservar a ordem dos elementos iguais em seu contexto, como em sistemas de classificação com mais de umas classificações de ordenação.

**19. Análise Visual dos Algoritmos**

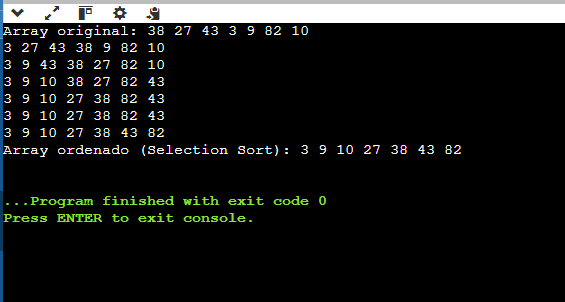
Saída do código Merge Sort



Saída do código Quick Sort



Saída do código Selection Sort



**20. Desafios de Implementação**

Saída do código de um elemento não encontrada na lista

