Lista de atividades

**Engenharia da computação – 4 sem.**

**Nome: Gustavo Salviano**

**RA:196620**

1. **Binary Search**

**Explique por que a lista deve estar ordenada para que o Binary Search funcione corretamente. Forneça exemplos**:

Se a lista não estiver ordenada, as divisões em metades não são úteis, pois não é possível garantir que o elemento buscado está na metade esquerda ou direita. Isso ocorre porque os elementos não obedecem a uma ordem fixa.

**Exemplo: Lista ordenada**

Lista: [1,3,5,7,9,11][1, 3, 5, 7, 9, 11][1,3,5,7,9,11]

Queremos encontrar o número 777:

1. O meio é 555. Como 7>57 > 57>5, busca-se na metade direita: [7,9,11][7, 9, 11][7,9,11].
2. O novo meio é 999. Como 7<97 < 97<9, busca-se na metade esquerda: [7][7][7]
3. O meio é 777, e o elemento é encontrado.
4. **Interpolation Search**

**Identifique casos em que o Interpolation Search é mais eficiente que o Binary Search:**

O interpolation serach pode ser mais eficiente do que o binary search nos seguintes casos:

- Distribuição Uniforme dos Dados

- Busca em Arrays Muito Grandes

- Menor Sobrecarga em Cálculos de Índices

1. **Jump Search**

**Compare o tempo de execução do Jump Search com o Binary Search em listas de diferentes tamanhos:**

A comparação entre o tempo de execução do **Jump Search** e do **Binary Search** depende do tamanho da lista, da forma como os elementos estão organizados e da eficiência dos algoritmos em termos de operações realizadas.

1. **Exponential Search**

**Analise o desempenho do Exponential Search em listas muito grandes e pequenas:**

Para **listas pequenas**, o ganho em eficiência do Exponential Search é marginal, e o Binary Search pode ser mais direto e simples.

Para **listas muito grandes**, o Exponential Search é ideal, especialmente se o elemento está próximo do início, pois ele rapidamente reduz o espaço de busca antes de aplicar a busca binária.

1. **Shell Sort**

**Explique como a escolha da sequência de intervalos afeta a eficiência do algoritmo:**

A escolha da **sequência de intervalos** (ou **gaps**) no **Shell Sort** é um dos fatores mais críticos para determinar a eficiência do algoritmo. O Shell Sort funciona dividindo o array em subgrupos, com base em intervalos decrescentes, e ordenando esses subgrupos antes de realizar uma ordenação final com intervalo 1. A sequência escolhida afeta diretamente o número de comparações e movimentações necessárias para ordenar o array.

**7) Selection Sort**

**Analise o desempenho do Selection Sort em listas pequenas, médias e grandes:**

Listas pequenas: Selection Sort pode ser eficiente e suficiente devido à simplicidade e baixa sobrecarga.

Listas médias e grandes: Algoritmos como Merge Sort, Quick Sort ou Heap Sort são muito mais adequados, pois escalam melhor com o tamanho da lista.

**9) Radix Sort**

**Explique como o algoritmo lida com bases diferentes (ex.: base 10 e base 2):**

O **Radix Sort** é um algoritmo de ordenação não-comparativa que organiza os dados processando os dígitos (ou bits) de cada elemento de forma iterativa, usando uma base numérica especificada. A base determina a granularidade com que os dados são analisados e, consequentemente, o número de passes necessários para ordenar a lista.

**10) Quick Sort**:

**Analise o desempenho do Quick Sort em listas quase ordenadas e completamente desordenadas:**

Listas Quase Ordenadas:

* Risco de pior caso se o pivô for mal escolhido (e.g., sempre o primeiro ou último elemento).
* Solução: Usar randomização ou mediana de três para garantir partições mais equilibradas.

Listas Completamente Desordenadas:

* O desempenho é geralmente independentemente da escolha do pivô.
* Algoritmos que utilizam pivô aleatório ou mediana de três são robustos neste cenário.

**11) Ternary Search**

**Identifique situações em que o Ternary Search seria mais eficiente que o Binary Search:**

O Ternary Search é mais eficiente que o Binary Search quando:

1. O problema envolve a otimização de funções unimodais (encontrar máximos ou mínimos)
2. Trabalhamos com intervalos contínuos ou funções de domínio real.
3. As comparações são custosas e a redução de número de iterações compensa.

**14) Análise de Complexidade**

**Analise a complexidade de tempo e espaço de cada algoritmo de busca e ordenação listados:**

**Buscas**:

A **Binary Search** é a mais eficiente em listas ordenadas de tempo.

**Interpolation Search** pode ser mais eficiente, mas depende da distribuição dos dados.

**Jump Search** e **Exponential Search** são alternativas eficientes, com complexidade de tempo respectivamente.

**Ordenação**:

**Merge Sort** e **Quick Sort** são as opções mais eficientes para listas grandes, ambos com tempo com **Merge Sort** sendo mais estável, mas exigindo mais memória.

**Shell Sort** e **Selection Sort** têm desempenho inferior, mas **Shell Sort** pode ser otimizado dependendo da sequência de intervalos.

**Bucket Sort** e **Radix Sort** são eficientes para dados com distribuições específicas, mas ambos exigem memória extra.

**18) Ordenação Estável e Instável**

**Identifique quais algoritmos de ordenação da lista são estáveis e explique o que isso significa. Demonstre com exemplos.**

Algoritmos Estáveis:

* Merge Sort
* Bubble Sort
* Insertion Sort

Algoritmos Não Estáveis:

* Quick Sort
* Selection Sort

**Importancia**: A estabilidade é importante em situações em que é necessário manter a ordem de elementos que são iguais, especialmente em cenários de ordenação múltipla. Por exemplo, se você ordenar primeiro por **nome** e depois por **idade**, um algoritmo estável garante que, quando ordenado por idade, as pessoas com o mesmo nome mantêm a ordem original.