**Busca binaria**

A lista deve ser ordenada para o funcionamento da Pesquisa Binária, pois o algoritmo divide o intervalo em duas metades e elimina a metade que não pode conter o elemento. Se a lista não estiver ordenada, não é possível garantir que metade possa ser descartada. Veja a lista para [3, 5, 7, 9, 11, 13, 15], e você busca 10, o algoritmo compara com o elemento do meio (9), e elimina a metade esquerda.

**Interpolation search**

em geral, os resultados da Pesquisa por Interpolação provavelmente serão tipicamente mais rápidos que a Pesquisa Binária se os métodos utilizados para identificá-las estiverem de acordo com as situações discutidas acima. Quando os dados são preenchidos em uma forma de seção transversal adequada, a interpolação pode ser usada para fornecer um bom palpite sobre aonde o valor desejado está, o que de forma simples e eficaz reduz as comparações. Parece que na maioria dos casos, a Interpolação seria preferível. Por outro lado, se a distribuição for irregular, a Pesquisa Binária seria a maneira mais segura ou tão eficiente quanto.

**Jump search**

Medida de Tempo:

A função "measureTime()" utiliza a função "clock()" da biblioteca time.h para medir o tempo gasto na execução de cada busca.

Testes com Listas de Tamanhos Diferentes:

O código testa o desempenho do Jump Search e do Binary Search em listas de tamanhos crescentes (100, 1000, 10000, 100000). Para simular o pior caso, a busca é feita pelo último elemento da lista.

**Exponetial search**

Casos em Listas Grandes:

Quando se trata de listas extensas, a Pesquisa Exponencial é particularmente vantajosa se o valor procurado estiver localizado próximo ao início. A etapa exponencial delimita rapidamente um intervalo pequeno onde o valor pode estar, e então a Pesquisa Binária refina a busca para encontrar o elemento exato. No pior cenário, a complexidade de tempo é O(log n), assim como na Pesquisa Binária, já que a fase exponencial reduz significativamente o tamanho da busca de forma logarítmica.

Casos em Listas Pequenas:

Para listas menores, a Pesquisa Exponencial também é bastante eficiente, pois a fase inicial localiza rapidamente o valor ou delimita um intervalo reduzido. Isso permite que uma etapa de Pesquisa Binária seja aplicada a uma porção mínima da lista. Nesses casos, o tempo total da Pesquisa Exponencial pode ser detalhado ao da Pesquisa Binária devido ao pequeno tamanho da estrutura comprovada.

**Shell sort**

A eficiência do Shell Sort é diretamente influenciada pela escolha da sequência de instruções utilizadas durante o processo. A sequência original, conhecida como sequência da Shell, reduz os intervalos pela metade em cada iteração. Apesar de ser simples de implementar, ela apresenta baixa eficiência, especialmente para listas grandes, pois não aproveita a forma ideal ou o espaçamento entre elementos. Isso resulta em mais comparações e trocas do que necessário. A complexidade de tempo costuma variar entre O(n^2) e O(n^(3/2)).

A sequência de Knuth é uma alternativa mais eficiente, utilizando intervalos maiores no início, o que reduz o número de trocas nas primeiras etapas do algoritmo. Esse método permite que uma fase final, com intervalo igual a 1, seja concluída de forma mais rápida, resultando em uma complexidade esperada de aproximadamente O(n^(3/2)).

Outra opção comum é a sequência de Hibbard, que se baseia em potências de dois. Ela apresenta desempenho prático muito bom, sendo mais eficiente que a sequência de Shell e frequentemente tão eficiente quanto a de Knuth. Sua complexidade de tempo também está em torno de O(n^(3/2)).

**Merge sort**

O conceito de "Dividir para Conquistar" fica bem claro no Merge Sort. Ele funciona dividindo o problema em partes menores, como separar um array grande em subarrays menores. Depois, cada subarray é resolvido individualmente, ordenando-os de forma recursiva. Por fim, as partes ordenadas são combinadas em uma única solução completa, que é a matriz totalmente ordenada. Esse processo continua até que as partes fiquem tão pequenas que sejam fáceis de resolver, como listas com apenas um elemento. Se quisermos usar o Merge Sort para ordenar strings em ordem alfabética, é só fazer algumas adaptações. Em vez de comparar números com operadores como `<=`, usamos a função `strcmp()` da biblioteca `string.h`, que é ótimo para comparar strings. Assim, o algoritmo consegue lidar com textos e organizar as palavras na ordem correta, do jeito que precisamos.

**Selection sort**

O desempenho do Selection Sort varia bastante dependendo do tamanho da lista. Em listas pequenas, como aquelas com até 10 elementos, ele é uma escolha aceitável. Sua simplicidade de implementação e facilidade de compreensão fazem com que seja uma boa opção nesse contexto. O tempo de execução é curto o suficiente para que sua ineficiência não seja um problema significativo.

Para listas médias, com tamanhos entre 100 e 1000 elementos, o Selection Sort ainda pode ser usado, mas começa a mostrar suas limitações. Comparado a algoritmos mais avançados, como Merge Sort ou Quick Sort, que têm complexidade O(n log n), o Selection Sort, com sua complexidade O(n²), exige muito mais comparações e trocas. Isso se torna menos eficiente à medida que o número de elementos aumenta.

Quando se trata de listas grandes, com mais de 1000 elementos, o Selection Sort se torna altamente ineficiente. Como o número de comparações cresce de forma quadrática, o tempo necessário para ordenar listas extensas pode ser inviável. Nesses casos, algoritmos como Quick Sort ou Merge Sort são muito mais protetores, oferecendo tempos de execução significativamente melhores.

**Bucket sort**

No Bucket Sort, os elementos são organizados em baldes (arrays temporários) com base em seu valor. O array bucketsarmazena esses baldes, e a distribuição dos números é feita utilizando uma função que calcula o índice do balde. Para números no intervalo (0, 1), o índice é obtido multiplicando o número pelo total de baldes. Por exemplo, se tivermos 10 baldes e o número de 0,78, ele será alocado no balde de índice 7, pois 0,78 × 10 = 7,8 (truncado para 7).

Se quisermos adaptar o Bucket Sort para números inteiros positivos que pertencem a intervalos maiores, precisamos ajustar uma fórmula que calcule o índice do balde. Em vez de trabalhar com números entre 0 e 1, devemos considerar a faixa de valores inteiros. Para isso, podemos determinar o número de baldes com base no intervalo total dos números.

Por exemplo, suponha que os números variam entre 0 e 999. Se quisermos 10 baldes, cada balde representará um intervalo de 100 números (0–99, 100–199, e assim por diante). Nesse caso, o índice do balde será calculado dividindo o número pelo tamanho do intervalo do balde. Para um número como 345, ele será colocado no balde de índice 3, pois 345 dividido por 100 é 3,45 (truncado para 3).

Esse ajuste permite que o Bucket Sort seja aplicado de forma eficiente a listas contendo números em faixas maiores, mantendo sua organização eficiente.

**Radix sort**

O Radix Sort, no exemplo com base 10, processa os números por dígito, de 0 a 9. Essa abordagem é amplamente usada porque reflete o sistema de numeração decimal, o que facilita a implementação e o entendimento. A cada iteração, o algoritmo ordena os números com base em um dígito específico, usando o Counting Sort como método auxiliar.

Na base 2, o Radix Sort é adaptado para processar os números bit por bit. Nesse caso, o algoritmo considera cada número como uma sequência de 0s e 1s, aplicando o Counting Sort para organizar os bits em cada passo. Aqui, o cálculo do expoente (exp) é ajustado para refletir a posição dos bits: 1, 2, 4, 8, e assim por diante, representando as potências de 2. A cada iteração, o Counting Sort ordena os números com base no bit correspondente, começando do menos significativo (LSB) até o mais significativo (MSB).

A escolha entre base 10 e base 2 depende do contexto. A base 10 é intuitiva e ideal para números decimais, enquanto a base 2 é eficiente para sistemas computacionais, onde os números são representados

**Quick sort**

O desempenho do Quick Sort está diretamente ligado à entrada e à estratégia de escolha do pivô.

Para listas quase ordenadas, o Quick Sort pode enfrentar sérios problemas de desempenho, especialmente quando o pivô é mal escolhido, como o primeiro ou o último elemento do array. Isso pode levar a divisões muito desiguais dos subarrays, resultando em recursões mais profundas e no pior caso de complexidade, O(n²). Esse comportamento ocorre porque o algoritmo não consegue dividir uma lista de forma equilibrada, aumentando significativamente o número de comparações e trocas.

Por outro lado, em listas completamente desordenadas, o Quick Sort tende a funcionar melhor. Se o pivô for escolhido especificamente, como usando a mediana de três valores ou selecionando um elemento aleatório, o algoritmo consegue dividir o array em partes mais equilibradas. Nesse caso, o Quick Sort alcança sua complexidade esperada de O(n log n), o que o torna muito eficiente para grandes volumes de dados.

A escolha do pivô é, portanto, crucial para o desempenho do Quick Sort, sendo uma estratégia bem planejada a chave para evitar cenários de pior caso e garantir eficiência em diferentes tipos de entradas.

**Ternary search**

A Pesquisa Binária divide o array em duas partes e, a cada iteração, compara o elemento com o do meio, decidindo em qual metade continuar a busca.

Já a Pesquisa Ternária divide o array em três partes. Ela faz duas comparações por iteração, o que pode reduzir o intervalo de busca mais rapidamente.

Embora a Pesquisa Ternária faça mais comparações por iteração, ela reduz o intervalo mais rápido. Porém, na maioria dos casos, a Pesquisa Binária é mais eficiente devido ao menor número de comparações. A Pesquisa Ternária pode ser útil em listas muito grandes, mas, geralmente, a Pesquisa Binária é preferida por sua simplicidade e eficiência.

**12 comparação de algoritmos de busca**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tamanho da lista** | **Binary search (0(log n)** | **Interpolation search (0(log log n) melhor caso** | **Jump search (0(n))** | **Exponential search (0(log n))** |
| 10 | 3 | 3 | 3 | 3 |
| 100 | 7 | 7 | 10 | 7 |
| 1.000 | 10 | 10 | 32 | 10 |
| 10.000 | 14 | 14 | 100 | 14 |
| 100.000 | 17 | 17 | 316 | 17 |
| 1.000,000 | 20 | 20 | 1.000 | 20.000 |
| 10.000,000 | 23 | 23 | 3.162 | 23 |
|  |  |  |  |  |

**13 comparação de algoritmos de ordenação**

Tamanho da lista 100

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Tempo de execução em segundos** | **Comparações** |
| Shell sort | 0,000087 | 387 |
| Merge sort | 0,000169 | 547 |
| Selection sort | 0,000343 | 4950 |
| Quick sort | 0,000110 | 698 |
| Bucket sort | 0,000100 | 125 |
| Radix sort | 0,000108 | 300 |

Tamanho da lista 1000

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Tempo de execução em segundos** | **Comparações** |
| Shell sort | 0,001707 | 6861 |
| Merge sort | 0,003167 | 8790 |
| Selection sort | 0,047683 | 499500 |
| Quick sort | 0,002877 | 13200 |
| Bucket sort | 0,001523 | 1250 |
| Radix sort | 0,001388 | 3000 |

Tamanho da lista 5000

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Tempo de execução em segundos** | **Comparações** |
| Shell sort | 0,018567 | 55430 |
| Merge sort | 0,016428 | 55220 |
| Selection sort | 1.042259 | 12498200 |
| Quick sort | 0,011200 | 71920 |
| Bucket sort | 0,007508 | 5001 |
| Radix sort | 0,009136 | 20000 |

Tamanho da lista 10000

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **Tempo de execução em segundos** | **Comparações** |
| Shell sort | 0,031253 | 125801 |
| Merge sort | 0,033358 | 120520 |
| Selection sort | 4.028235 | 49995000 |
| Quick sort | 0,025882 | 168490 |
| Bucket sort | 0,019800 | 10000 |
| Radix sort | 0,022925 | 4000 |

**14 Análise de complexidade**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algoritmo** | **Complexidade de Tempo** | **Complexidade de Espaço** | **Observações** |
| Binary Search | O(logn) | O(1) | Funciona apenas em arrays ordenados. Divide a busca pela metade a cada iteração. |
| Interpolation Search | O(loglogn) no melhor caso, O(n) no pior caso | O(1) | Útil para dados uniformemente distribuídos, mas pode ser lento em dados não uniformes. |
| Jump Search | O(√n)​ | O(1) | Divide o array em blocos, fazendo saltos de tamanho √n​. Após o salto, realiza busca linear no bloco. |
| Exponential Search | O(logn) | O(1) | Rápido para encontrar o intervalo, mas requer que o array esteja ordenado. Utiliza Binary Search no intervalo encontrado. |
| Shell Sort | O(n3/2) no caso médio, O(n2) no pior caso | O(1) | Usa gaps para comparações, melhorando a eficiência para listas parcialmente ordenadas. |
| Merge Sort | O(nlogn) em todos os casos | O(n) | Divide e conquista. Efetivamente lida com listas grandes, mas requer espaço adicional para armazenar subarrays. |
| Selection Sort | O(n2) em todos os casos | O(1) | Simples de implementar, mas ineficiente para listas grandes devido ao grande número de comparações. |
| Quick Sort | O(nlogn) no melhor/médio caso, O(n2) no pior caso | O(logn) | Divide e conquista. O caso pior pode ser evitado com pivôs escolhidos aleatoriamente ou pelo método de mediana de três. |
| Bucket Sort | O(n+k) | O(n+k) | Útil para dados uniformemente distribuídos em um intervalo pequeno. Requer k buckets adicionais, onde k é o número de buckets. |
| Radix Sort | O(nk), onde k é o número de dígitos | O(n+k) | Especializado para inteiros. Rápido para listas com inteiros de tamanho fixo, mas não é comparativo. |

**18 Ordenação Estável e instável**

|  |  |  |
| --- | --- | --- |
| **Algoritmo** | **categoria** | **Explicação do por que** |
| Merge Sort | Estável | Durante a fusão, elementos iguais são copiados na ordem em que aparecem nos subarrays originais. |
| Bucket Sort | Estável | Desde que cada bucket seja ordenado usando um algoritmo estável, os elementos permanecem na ordem relativa em que foram distribuídos nos buckets. |
| Radix Sort | Estável | Ordena dígito por dígito (ou unidade por unidade) preservando a ordem relativa em cada etapa. |