Nome: Marcos Vinícius da Silva
Matrícula: 16-95704

Relatório Técnico

Quick Sort, Bubble Sort e Heap Sort

Explicação do código

Descrição geral da arquitetura do programa

O programa foi estruturado para demonstrar e comparar três algoritmos de ordenação: Quick Sort, Bubble Sort e Heap Sort. A arquitetura principal consiste em uma função main que orquestra a execução para diferentes tamanhos de vetores. Para cada tamanho, um vetor de números inteiros aleatórios e não repetidos é gerado. Cópias desse vetor são então passadas para as funções de ordenação correspondentes, a fim de garantir que cada algoritmo opere sobre os mesmos dados iniciais e permita uma comparação justa. Estatísticas de desempenho (número de trocas, particionamentos/heapify e tempo de execução) são coletadas para cada algoritmo e exibidas no console.

Explicação das funções criadas para cada algoritmo

gerarVetorAleatorio(int tamanho): Esta função é responsável por criar um vetor de inteiros aleatórios, garantindo que não haja repetições. Ela utiliza um array booleano auxiliar (usado) para rastrear quais números já foram inseridos no vetor, assegurando a unicidade dos elementos.

copiarVetor(int* original, int tamanho): Uma função utilitária para criar uma cópia exata de um vetor. Isso é crucial para que cada algoritmo de ordenação seja testado com a mesma sequência inicial de dados.

bubbleSort(int* arr, int n, Estatisticas* stats): Implementa o algoritmo Bubble Sort. Percorre o array várias vezes, comparando elementos adjacentes e trocando-os se estiverem na ordem errada. Conta o número de trocas e o tempo de execução.

particionar(int* arr, int low, int high, Estatisticas* stats): Uma função auxiliar para o Quick Sort. Escolhe o último elemento como pivô e reorganiza o sub-array de forma que todos os elementos menores que o pivô fiquem antes dele e todos os maiores fiquem depois. Conta o número de trocas e o número de particionamentos.

quickSort(int* arr, int low, int high, Estatisticas* stats): Implementação recursiva do Quick Sort. Divide o array em sub-arrays menores, particiona-os e chama-se recursivamente nos sub-arrays.

iniciarQuickSort(int* arr, int n, Estatisticas* stats): Inicializa as estatísticas e mede o tempo total de execução para o Quick Sort.

heapify(int* arr, int n, int i, Estatisticas* stats): Uma função auxiliar para o Heap Sort. Garante que a sub-árvore com raiz no índice i seja um max-heap. Se o elemento raiz não for o maior, ele é trocado com o maior filho, e heapify é chamado recursivamente para o sub-árvore afetada. Conta o número de trocas e chamadas heapify.

heapSort(int* arr, int n, Estatisticas* stats): Implementa o algoritmo Heap Sort. Primeiro, constrói um max-heap a partir do array de entrada. Em seguida, extrai repetidamente o maior elemento (raiz do heap) e o coloca no final do array, chamando heapify na heap restante.

Justificativa da estratégia de pivô escolhida no quick sort

Para a implementação do Quick Sort, a estratégia de escolha do pivô foi o último elemento do sub-array. Essa escolha é simples de implementar e é bastante comum. Embora outras estratégias (primeiro elemento, elemento central, elemento aleatório ou mediana de três) possam oferecer melhor desempenho em cenários específicos (evitando casos de pior caso para arrays já ordenados ou quase ordenados), a escolha do último elemento é suficiente para demonstrar o funcionamento básico do algoritmo. Para vetores gerados aleatoriamente, a chance de encontrar o pior caso com essa estratégia é mínima.

Método utilizado para gerar vetores sem valores repetidos

O método para gerar vetores sem valores repetidos envolve a criação de um array booleano auxiliar, usado, do mesmo tamanho do vetor a ser gerado. Cada posição nesse array booleano, inicialmente definida como false, corresponde a um número que pode ser inserido no vetor. O programa gera números aleatórios dentro do intervalo [0, tamanho-1]. Antes de adicionar um número ao vetor, ele verifica se a posição correspondente em usado é true. Se for, significa que o número já foi utilizado, e um novo número aleatório é gerado. Se for false, o número é adicionado ao vetor, e a posição correspondente em usado é marcada como true. Isso garante que cada número seja único no vetor gerado.

Como foram contabilizadas as estatísticas de desempenho?

Número de trocas: Uma variável trocas dentro da estrutura Estatisticas é incrementada toda vez que dois elementos no array são fisicamente trocados de posição. Isso é feito diretamente nas funções de ordenação (Bubble Sort, Quick Sort e Heap Sort).

Número de particionamentos (Quick Sort): A variável particionamentos_heapify (reutilizada para ambas as métricas) é incrementada na função particionar do Quick Sort cada vez que um particionamento é concluído.

Número de chamadas para "heapify" (Heap Sort): A variável particionamentos_heapify é incrementada na função heapify do Heap Sort a cada vez que a função é invocada.

Tempo de execução: O tempo de execução é medido utilizando a biblioteca chrono do C++. Antes de chamar a função de ordenação, std::chrono::high_resolution_clock::now() é usado para registrar o inicio do tempo. Após a conclusão da ordenação, o fim do tempo é registrado. A diferença entre fim e inicio é então calculada e convertida para milissegundos (std::chrono::duration<double, std::milli>) para fornecer a métrica de tempo de execução.

Análise comparativa

Tabela com as métricas coletadas para 10, 100, 1.000 e 10.000 elementos

Tamanho	Algoritmo	Trocas	Particionamentos/Heapify	Tempo (ms)
10	Bubble Sort	21	N/A	8
	Quick Sort	29	6	8
	Heap Sort	29	34	14
100	Bubble Sort	2401	N/A	251
	Quick Sort	440	63	78
	Heap Sort	575	625	116
1.000	Bubble Sort	249883	N/A	1.6764
	Quick Sort	5234	666	851
	Heap Sort	9077	9577	1.543
10.000	Bubble Sort	23713243	N/A	164.136
	Quick Sort	82146	6698	1.1163
	Heap Sort	124447	129447	2.1397

Eficiência, escalabilidade e características observadas

- Bubble Sort:

Eficiência: O Bubble Sort demonstra ser extremamente ineficiente à medida que o tamanho do vetor aumenta. Para 10 elementos, o tempo de execução é negligenciável (0,0008 ms), mas para 10.000 elementos, o tempo salta para 164,136 ms, um aumento massivo. O número de trocas é exponencialmente maior em comparação com os outros algoritmos, chegando a mais de 23 milhões para 10.000 elementos.

Escalabilidade: Muito pobre, confirmando sua complexidade de tempo $O(n^2)$.

Características Observadas: As trocas ocorrem aos milhões para entradas maiores, indicando um alto custo operacional.

- Quick Sort:

Eficiência: O Quick Sort demonstra ser o mais eficiente entre os três algoritmos para os tamanhos de vetor testados, especialmente para 1.000 e 10.000 elementos. Para 10.000 elementos, ele completa a ordenação em apenas 1,1163 ms, com um número de trocas significativamente menor (82.146) em comparação com o Bubble Sort. O número de particionamentos cresce de forma logarítmica, mantendo a eficiência.

Escalabilidade: Excelente, consistente com sua complexidade média de O(nlogn).

Características Observadas: Menor tempo de execução e menor número de trocas para vetores maiores em comparação com o Heap Sort, indicando uma constante de tempo menor na prática.

- Heap Sort:

Eficiência: O Heap Sort apresenta boa eficiência, com tempo de execução de 2,1397 ms e 124.447 trocas para 10.000 elementos, sendo significativamente superior ao Bubble Sort. No entanto, é consistentemente mais lento que o Quick Sort para os tamanhos testados e registra um número maior de chamadas heapify em comparação com os particionamentos do Quick Sort, e também um número de trocas maior que o Quick Sort.

Escalabilidade: Muito boa, mantendo sua complexidade de O(nlogn).

Características Observadas: As numerosas chamadas à função heapify são a característica dominante, impactando o tempo de execução em comparação com o Quick Sort.

Opinião fundamentada

Análise crítica sobre as vantagens e desvantagens dos três algoritmos

- Bubble Sort:

Vantagens: A principal vantagem é sua simplicidade conceitual e de implementação.

Desvantagens: Os dados empíricos confirmam que o Bubble Sort é dramaticamente ineficiente para conjuntos de dados de qualquer tamanho razoável. Seu desempenho $O(n^2)$ é ruim, com o número de trocas e o tempo de execução crescendo exponencialmente. Não é adequado para uso prático em quase nenhum cenário de ordenação real.

- Quick Sort:

Vantagens: A análise dos dados demonstra que o Quick Sort é o algoritmo mais rápido na prática para os tamanhos de entrada testados. Seu tempo de execução é significativamente menor do que o Bubble Sort e, de forma consistente, ligeiramente menor do que o Heap Sort. O número de trocas e particionamentos é bem gerenciável, refletindo sua eficiência média O(nlogn). É um algoritmo "in-place", o que significa que ele requer pouca memória auxiliar.

Desvantagens: Embora não observado nos testes com vetores aleatórios, a principal desvantagem teórica do Quick Sort é seu pior caso de desempenho de $O(n^2)$, que ocorre com escolhas de pivô desfavoráveis (por exemplo, em vetores já ordenados ou inversamente ordenados, dependendo da estratégia do pivô). Isso pode ser mitigado por técnicas como pivôs aleatórios ou mediana de três.

- Heap Sort:

Vantagens: O Heap Sort oferece a garantia de desempenho de O(nlogn) em todos os casos (melhor, médio e pior), o que o torna mais robusto do que o Quick Sort em cenários onde a estabilidade do tempo de execução é crítica. É também um algoritmo "in-place", o que é benéfico para a gestão de memória.

Desvantagens: Embora eficiente, os dados mostram que o Heap Sort é marginalmente mais lento que o Quick Sort na prática para os tamanhos testados. O maior número de chamadas heapify e, por consequência, de trocas, pode indicar uma constante de tempo maior que a do Quick Sort. Além disso, seu padrão de acesso à memória (menos localizado que o Quick Sort) pode resultar em pior desempenho de cache.

Comentários sobre a importância da escolha do pivô no quick sort

A estratégia de escolha do pivô é fundamental para o desempenho do Quick Sort. No nosso caso, usando o último elemento como pivô, os resultados foram excelentes devido à natureza aleatória dos dados de entrada. Com vetores aleatórios, a probabilidade de uma escolha de pivô que leve ao pior caso ($O(n^2)$) é baixa. No entanto, se os dados de entrada fossem sempre ordenados ou inversamente ordenados (dependendo da implementação específica), a escolha de um pivô fixo (como o último elemento) poderia causar a degradação do desempenho. Para aplicações críticas, a estratégia de pivô aleatório ou a mediana de três são preferíveis, pois minimizam a chance do pior caso, garantindo uma performance mais consistente e próxima do O(nlogn) médio.

Considerações finais sobre em quais cenários cada algoritmo seria mais adequado

Bubble Sort: Não é adequado para praticamente nenhuma aplicação prática de ordenação. Sua utilidade é restrita a propósitos educacionais ou para ordenar conjuntos de dados triviais (muito pequenos), onde a clareza e simplicidade do código são mais importantes do que qualquer métrica de desempenho.

Quick Sort: É a escolha ideal para a maioria das aplicações gerais que exigem alta performance de ordenação em grandes conjuntos de dados. Sua velocidade superior na prática (como demonstrado pelos tempos de execução) o torna a opção preferida em sistemas onde a performance média é crucial. É amplamente utilizado em bibliotecas de ordenação de alto desempenho.

Heap Sort: É mais adequado para cenários onde a garantia de desempenho no pior caso é um requisito não negociável. Isso inclui sistemas embarcados, sistemas de tempo real ou ambientes onde a imprevisibilidade do desempenho do Quick Sort (devido ao pior caso) não é aceitável. Embora ligeiramente mais lento que o Quick Sort em média, sua consistência de O(nlogn) em todas as situações e seu atributo in-place o tornam uma alternativa robusta e confiável.