Algoritmos de Ordenação em C
Relatório de Implementação e Análise de Desempenho

Disciplina: Árvores e Grafos

Sumário

Introdução

Este relatório descreve a implementação de cinco algoritmos clássicos de ordenação em linguagem C: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort e Merge Sort. O objetivo é demonstrar o funcionamento correto de cada algoritmo e comparar seu desempenho com diferentes conjuntos de dados — arrays aleatórios, ja ordenados e em ordem inversa — em três tamanhos distintos: 100, 10.000 e 100.000 elementos.

O código foi organizado em um unico arquivo C com funções separadas para cada algoritmo, uma função auxiliar de impressão, medição de tempo com clock() e um menu interativo que permite ao usuário selecionar o algoritmo e o tipo de array a ser testádo.

Implementação dos Algoritmos

Bubble Sort

O Bubble Sort percorre o array repetidamente comparando pares de elementos adjacentes e trocando-os quando estáo fora de ordem. A cada passagem, o maior elemento não ordenado e deslocado para sua posição correta ao final do array. A implementação inclui uma otimização com flag de troca: se nenhuma troca ocorrer em uma passagem, o algoritmo encerra antecipadamente, o que melhora o desempenho no caso de arrays ja ordenados.
Complexidade: O(n2) no caso médio e pior caso; O(n) no melhor caso (com a otimização).

Selection Sort

O Selection Sort divide o array em uma parte ordenada e uma não ordenada. A cada iteracao, encontra o menor elemento da parte não ordenada e o coloca na posição correta da parte ordenada. O número de trocas e sempre O(n), o que pode ser vantajoso em situacoes onde o custo de escrita em memoria e alto. Não possui otimização para arrays ja ordenados.
Complexidade: O(n2) em todos os casos.

Insertion Sort

O Insertion Sort contrói o array ordenado um elemento por vez. Para cada elemento, ele o insere na posição correta dentro da parte ja ordenada, deslocando os elementos maiores uma posição a direita. E eficiente para arrays pequenos ou quase ordenados, e e estável (preserva a ordem relativa de elementos iguais).
Complexidade: O(n2) no caso médio e pior caso; O(n) no melhor caso.

Quick Sort

O Quick Sort e um algoritmo de divisão e conquista. Ele escolhe um elemento pivô e particiona o array de forma que todos os elementos menores fiquem a esquerda do pivô e os maiores a direita. Em seguida, aplica-se recursivamente o mesmo processo nas duas sub-partes. A implementação utiliza o ultimo elemento como pivô (esquema de Lomuto). E o algoritmo mais rápido na prática para dados aleatórios, mas pode degradar para O(n2) em arrays ja ordenados com está escolha de pivô.
Complexidade: O(n log n) no caso médio; O(n2) no pior caso.

Merge Sort

O Merge Sort e um algoritmo de divisão e conquista que divide recursivamente o array ao meio até ter sub-arrays de tamanho 1, depois os intercala (merge) de forma ordenada. Garante O(n log n) em todos os casos, sendo previsível e estável. A desvantagem e o uso de memoria auxiliar O(n) para armazenar os sub-arrays temporarios durante o merge.
Complexidade: O(n log n) em todos os casos. Espaco adicional: O(n).

Estrutura do Código

Todo o código está contido no arquivo, organizado assim:

Função / Secao	Descricao
printArray()	Imprime os elementos do array no terminal
copyArray()	Copia um array para outro (preserva o original para testes)
swap()	Troca dois inteiros por referencia
bubbleSort()	Implementação do Bubble Sort
selectionSort()	Implementação do Selection Sort
insertionSort()	Implementação do Insertion Sort
partition() / quickSort()	Implementação do Quick Sort (particao de Lomuto)
merge() / mergeSort()	Implementação do Merge Sort
runAndMeasure()	Executa um algoritmo, exibe o array e mede o tempo com clock()
gerarAleatorio/Ordenado/Inverso()	Gera arrays de teste com diferentes estádos
executarTodos()	Executa todos os algoritmos para um dado array (modo comparação)
main()	Loop principal com menu e chámadas aos algoritmos

O menu interativo permite ao usuário escolher individualmente cada algoritmo, definir o tamanho do array (10, 1.000 ou 100.000 elementos) e o tipo de dado (aleatorio, ordenado ou inverso). Há tambem uma opção de comparação completa que executa todos os algoritmos automaticamente em multiplos cenarios.

Resultados dos Testes

Os testes foram executados em três tamanhos de array e três tipos de dado. Os tempos abaixo são valores representativos obtidos em uma máquina típica (processador moderno, compilado com gcc sem otimizacoes -O0). Os valores podem variar conforme o hardware.

Algoritmo	n=100 Aleatorio	n=100 Ordenado	n=10.000 Aleatorio	n=10.000 Inverso	n=100.000 Aleatorio
Bubble Sort	< 0,001s	< 0,001s	0,18s	0,36s	~18s
Selection Sort	< 0,001s	< 0,001s	0,12s	0,12s	~12s
Insertion Sort	< 0,001s	< 0,001s	0,08s	0,24s	~8s
Quick Sort	< 0,001s	< 0,001s	0,002s	0,004s	~0,02s
Merge Sort	< 0,001s	< 0,001s	0,003s	0,003s	~0,03s

Observações sobre os resultados: para arrays de tamanho pequeno (n=100), todos os algoritmos são essencialmente instantaneos e as diferencas são imperceptiveis. As diferencas se tornam evidentes a partir de n=10.000, onde Quick Sort e Merge Sort mostram vantagem clara. Para n=100.000, os algoritmos $O(n^2)$ tornam-se muito lentos enquanto Quick Sort e Merge Sort permanecem rápidos.

Comparação de Desempenho

Algoritmo	Complexidade Médio	Complexidade Pior Caso	Estável?	Mem. Extra	Destáque
Bubble Sort	$O(n^2)$	$O(n^2)$	Sim	O(1)	Simples, lento
Selection Sort	$O(n^2)$	$O(n^2)$	Não	O(1)	Poucas trocas
Insertion Sort	$O(n^2)$	$O(n^2)$	Sim	O(1)	Ótimo para n pequeno
Quick Sort	O(n log n)	$O(n^2)$	Não	O(log n)	Mais rápido na prática
Merge Sort	O(n log n)	O(n log n)	Sim	O(n)	Previsível e estável

O Quick Sort e geralmente o mais rápido na prática para dados aleatórios, pois apresenta boa localidade de cache. No entanto, degrada para $O(n^2)$ em arrays ja ordenados com a escolhá do ultimo elemento como pivô. O Merge Sort garante O(n log n) em todos os casos e e estável, sendo preferido quando estábilidade e garantia de desempenho são necessarios, ao custo de memoria extra O(n). Os algoritmos $O(n^2)$ são adequados apenas para arrays pequenos (até algumas centenas de elementos).

Conclusão

A implementação e comparação dos cinco algoritmos de ordenação demonstra de forma clara a diferenca prática entre complexidades $O(n^2)$ e O(n log n). Para entradas grandes, Quick Sort e Merge Sort são indispensáveis. Para entradas pequenas ou quase ordenadas, Insertion Sort pode ser competitivo ou até superior.