Complexidade de Algoritmos Bubble Sort

As operações de comparações e de troca de posição de elementos são executadas no Pior Caso, o algoritmo realizará n-1 troca para o primeiro passo, depois n-2 trocas para o segundo elemento e assim sucessivamente. Trocas = n-1+n-2+n-3...+2+1 aproximadamente n2 trocas.

No Melhor Caso, nenhuma troca será realizada, pois em ambos os casos o algoritmo faz da ordem n comparações.

Complexidade no tempo: Comportamento do algoritmo no tempo, em função do tamanho da entrada.

Complexidade no espaço: Consumo de memória do algoritmo, em função do tamanho da entrada.

O tempo gasto na execução do algoritmo varia em ordem quadrática em relação ao número de elementos a serem ordenados.

- $T = O(n^2)$ – Notação "Big O"

- Atividades mais custosas:

  • Comparações

  • Troca de Posição (swap)

Melhor caso: Vetor Ordenado.

Pior caso: Vetor invertido.

Complexidade de Algoritmos Selection Sort

A operação entre as chaves é feita no loop k, para cada valor de i são realizadas (i-1) comparações no loop, como i varia de 2 até n, o número total de comparações para ordenar a lista toda é que para qualquer valor de i existe no máximo uma troca, se no caso a lista já estiver ordenada não ocorre troca. Pior caso existe uma troca para cada loop de k (n-1) para cada troca exige três movimentos. O algoritmo de seleção é considerado um dos mais simples, além disso, possui uma característica eficiente quanto à quantidade de movimentações de registros, com um tempo de execução linear no tamanho de entrada. Geralmente é utilizado para arquivos de registros maiores com até 1000 registros.

- $T = O(n^2)$

O fato de o conjunto já estar ordenado não ajuda em nada. O algoritmo não é estável, isto é, os registros com chaves iguais nem sempre irão manter a mesma posição relativa de antes do início da ordenação.

Complexidade de Algoritmos Insertion Sort

Se o valor a ordenar possui n elementos, então o algoritmo realizará n – 1 etapas. Quantas comparações e trocas serão realizadas?

• No melhor caso, vetor ordenado, serão realizadas 1 comparação e 1 troca por etapa, um total de (n-1) comparações e (n-1) trocas.

• No pior caso, vetor em ordem inversa, serão realizadas sucessivamente 1,2,3,... n1 comparações e trocas.

A soma dos termos dessa progressão aritmética será $n^2/2$.

Pode ser demonstrado que para um vetor aleatório, o número aproximado de comparações e trocas é $n^2/4$.

Portanto, a complexidade deste algoritmo é quadrática.

- $T = O(n^2)$.

No anel mais interno, na i-ésima iteração, o valor de Ci é: Melhor caso: Ci(número de comparações) = 1 Pior caso: Ci(número de comparações) = i Caso médio: Ci(número de comparações) = 1/i(1 + 2 + ... + i) = (i+1)/2 Se todas as permutações de n são igualmente prováveis para o caso médio, então, o número de comparações é igual a: Melhor caso: C(n) = (1 + 1 + ... + 1) = n - 1 Pior caso: C(n) = (2 + 3 + ... + n) = n2/2 + n/2 - 1 Caso médio: C(n) = 1/2(3 + 4 + ... + n+1) = n2/4 + 3n/4 – 1 O número de movimentações na i-ésima iteração é igual a Mi(n) = Ci(n) – 1 + 3 = Ci(n) + 2 logo, o número de movimentos é igual a: Melhor caso: M(número de elementos do arquivo) = (3 + 3 + ... + 3) = 3(n – 1) Pior caso: M(número de elementos do arquivo) = (4 + 5 + ... + n+2) = n2/2 + 5n/2 – 3 Caso médio: M(número de elementos do arquivo) = 1/2(5 + 6 + ... + n+3) = n2/4 + 11n/4 – 3 Deste modo podemos concluir que: Melhor caso: O(n) Pior caso: O(n2) Caso médio: O(n2) O menor número de comparações e trocas ocorre exatamente quando os elementos estão originalmente em ordem, e o número máximo de comparações e trocas ocorre quando os itens estão originalmente na ordem oposta. Para arquivos já ordenados o algoritmo descobre a um custo O(n) que cada item já está em seu lugar. Então, este é o método a ser utilizado quando o arquivo está ordenado na sua origem. É também um método bom para adicionar um pequeno conjunto de dados a um arquivo já ordenado, originando um outro arquivo ordenado, pois neste caso o custo pode ser considerado linear

Complexidade dos algoritmos QuickSort

caso médio O(N * log N)

Esse método de ordenação divide-se em vários passos:

- Escolher para pivô o primeiro elemento da tabela (p=x[1])
- Se os elementos de x forem rearranjados de forma a que o pivô (p) sejam colocados na posição j e sejam respeitadas as seguintes condições:
    1) todos os elementos entre as posições 1 e j-1 são menores ou iguais que o pivô (p)

2) todos os elementos entre as posições j+1 e n são maiores que o pivô (p)

Então p permanecerá na posição j no final do ordenamento.

- Se este processo for repetido para as sub-tabelas x[1] a x[j-1] e x[j+1] a x[n] e para todas as sub-tabelas criadas nas iterações seguintes obteremos no final uma tabela ordenada.

Portanto a parte mais difícil deste método é o procedimento **parte** que divide a tabela em 2 sub-tabelas dependendo do pivô.

Complexidade dos algoritmos Merge Sort

Primeiramente vamos definir o que é melhor, médio e pior caso para o MergeSort. Melhor Caso – nunca é necessário trocar após comparações.

Médio Caso – há necessidade de haver troca após comparações.

Pior Caso – sempre é necessário trocar após comparações.

Para o MergeSort não tem tanta importância se o vetor está no melhor, médio ou pior caso, porque para qualquer que seja o caso ele sempre terá a complexidade de ordem n*logn, como pode ser verificado na tabela abaixo:

Melhor caso	O(n log2 n)
Médio caso	O(n log2 n)
Pior caso	O(n log2 n)

Isso é pelo motivo de que o MergeSort independentemente em que situação se encontra o vetor, ele sempre irá dividir e intercalar.

Na prática, é difícil (senão impossível) prever com rigor o tempo de execução de um algoritmo ou programa.

O tempo vai depender de varias constantes, como por exemplo, o tempo de processamento de cada computador, do algoritmo implementado. Desta maneira, nós não vamos apresentar aqui como é o calculo da análise de complexidade do MergeSort.

conclusão

Encontrei dificuldade em fazer a implementação do gráfico interativo, e para implementar o algoritmo quickSort que estava apresentando problemas e travamentos com vetores ordenados crescente e decrescente com cem mil posições, que só pode ser resolvido diminuindo os vetores A e B pra mil posições.

Os resultados obtidos podem ser vistos no gráfico de barras implementado no código.