Faculdade Professor Miguel Ângelo da Silva Santos - FeMASS

DISCIPLINA: Estrutura de dados II
CURSO(S): Sistema de Informação
PROFESSOR (A): Irineu de Azevedo Lima Neto
ALUNO (A): Ariel de Oliveira L Pichone
DATA: 27/09/2021 NOTA:_____

Atividade composta de teoria e prática: Algoritmos de ordenação e complexidade:

Atividade composta de teoria e prática. Veja:
> Teoria:
Considere:
- Ordenação simples (elementares): Bubble Sort, Insert Sort e Selection Sort.
- Ordenação avançada (eficientes): Quick Sort, Shell Sort, Merge Sort, Radix Sort.

Documente estudo sobre funcionamento e análise de complexidade (Big O) para os algoritmos de ordenação acima. Cada algoritmo de ordenação em análise deve ser idêntico ao aplicado na atividade prática (descrita a seguir). A ideia é corroborar maiores tempos na execução da prática aos algoritmos de maior complexidade.

Ordenação simples (elementares):

Bubble Sort
Melhor caso: Ω(n)
Caso médio: θ(n^2)
Pior caso: O(n^2)

Bubble Sort é o algoritmo de classificação mais simples que funciona trocando repetidamente os elementos adjacentes se eles estiverem na ordem errada.
Exemplo:
Primeira passagem:
(**5 1** 4 2 8) -> (**1 5** 4 2 8), aqui, o algoritmo compara os dois primeiros elementos e troca desde 5> 1.
(1 **5 4** 2 8) -> (1 **4 5** 2 8), Trocar desde 5> 4
(1 4 **5 2** 8) -> (1 4 **2 5** 8), Trocar desde 5> 2
(1 4 2 **5 8**) -> (1 4 2 **5 8**), Agora, como esses elementos já estão em ordem (8> 5), o algoritmo não os troca.
Segunda passagem:
(**1 4** 2 5 8) -> (**1 4** 2 5 8)
(1 **4 2** 5 8) -> (1 **2 4** 5 8), Trocar desde 4> 2

(1 2 **4 5** 8) -> (1 2 **4 5** 8)
(1 2 4 **5 8**) -> (1 2 4 **5 8**)
Agora, o array já está ordenado, mas nosso algoritmo não sabe se está completo. O algoritmo precisa de uma passagem inteira sem nenhuma troca para saber que está ordenado.
Terceira passagem:
(**1 2** 4 5 8) -> (**1 2** 4 5 8)
(1 **2 4** 5 8) -> (1 **2 4** 5 8)
(1 2 **4 5** 8) -> (1 2 **4 5** 8)
(1 2 4 **5 8**) -> (1 2 4 **5 8**)

Pior caso e complexidade de tempo médio: O (n * n). O pior caso ocorre quando o array está ordenado de forma reversa.
Melhor caso de complexidade de tempo: O (n). O melhor caso ocorre quando o array já está ordenado.
Casos de limite: Bubble sort leva um tempo mínimo (ordem de n) quando os elementos já estão ordenados.
Em computação gráfica, é popular por sua capacidade de detectar um erro muito pequeno (como a troca de apenas dois elementos) em arrays quase classificados e corrigi-lo apenas com complexidade linear (2n). Por exemplo, é usado em um algoritmo de preenchimento de polígono, onde as linhas delimitadoras são classificadas por sua coordenada x em uma linha de varredura específica (uma linha paralela ao eixo x) e com o incremento de y, sua ordem muda (dois elementos são trocados) apenas em interseções de duas linhas

---

Insert Sort
Melhor caso: Ω(n)
Caso médio: θ(n^2)
Pior caso: O(n^2)

Insert Sort é o algoritmo de classificação simples onde o array é virtualmente dividido em uma parte ordenada e outra não ordenada. Os valores da parte não ordenada são selecionados e colocados na posição correta na parte ordenada.
Exemplo:
12 , 11, 13, 5, 6
Vamos fazer um loop de i = 1 (segundo elemento do array) para 4 (último elemento do array)
i = 1. Como 11 é menor que 12, mova 12 e insira 11 antes de 12
11, **12** , 13, 5, 6
i = 2. 13 permanecerá em sua posição, pois todos os elementos em A [0..I-1] são menores do que 13
11, **12**, **13** , 5, 6
i = 3. 5 se moverá para o início e todos os outros elementos de 11 a 13 se moverão uma posição à frente de sua posição atual.
5, **11**, **12**, **13** , 6
i = 4. 6 se moverão para a posição após 5 e os elementos de 11 a 13 se moverão uma posição à frente de sua posição atual.
5, **6**, **11**, **12**, **13**

Complexidade de tempo: O (n ^ 2)
Casos de limite : Insert Sort leva o tempo máximo para ordenar se os elementos estiverem ordenados na ordem reversa. E leva um tempo mínimo (ordem de n) quando os elementos já estão ordenados.
Usos: Insert Sort é usado quando o número de elementos é pequeno. Também pode ser útil quando o array de entrada está quase todo ordenado, com somente alguns elementos fora de ordem em um grande array completo.

Selection Sort
Melhor caso: Ω(n)
Caso médio: θ(n^2)
Pior caso: O(n^2)

O algoritmo Selection Sort ordena um array encontrando repetidamente o elemento mínimo (considerando a ordem crescente) da parte não classificada e colocando-o no início. O algoritmo mantém dois sub-arrays em um determinado array.

1) O array que já está ordenado.
2) Sub-array restante não ordenada.

Em cada iteração do Selection Sort , o elemento mínimo (considerando a ordem crescente) do subarray não ordenado é selecionado e movido para o subarray ordenado.
O exemplo a seguir explica as etapas acima:
arr [] = 64 25 12 22 11

// Encontre o elemento mínimo em arr [0 ... 4]
// e coloque-o no início
11 25 12 22 64

// Encontre o elemento mínimo em arr [1 ... 4]
// e coloque-o no início de chegada [1 ... 4]
11 **12** 25 22 64

// Encontre o elemento mínimo em arr [2 ... 4]
// e coloque-o no início de [2 ... 4]
11 12 **22** 25 64

// Encontre o elemento mínimo em arr [3 ... 4]
// e coloque-o no início de chegada [3 ... 4]
11 12 22 **25** 64
Complexidade de tempo: O (n 2), pois há dois loops aninhados.
A coisa boa sobre o Selection Sort é que nunca faz mais do que O (n) trocas e pode ser útil quando a gravação na memória é uma operação de custo.

Ordenação avançada (eficientes):

Quick Sort
Melhor caso: Ω(n log(n))
Caso médio: θ(n log(n))
Pior caso: O(n^2)
Assim como o Merge Sort , o Quick Sort é um algoritmo de divisão e conquista. Ele seleciona um elemento como pivô e particiona o array fornecido ao redor do pivô selecionado. Existem muitas versões diferentes do Quick Sort que selecionam o pivô de maneiras diferentes.

1. Sempre escolhe o primeiro elemento como pivô.
2. Sempre escolhe o último elemento como pivô (implementado)
3. Escolhe um elemento aleatório como pivô.
4. Escolhe a mediana como pivô.

O principal processo em Quick Sort é a partição (). O destino das partições é, dado um array e um elemento x do array como pivô, colocar x em sua posição correta no array ordenado e colocar todos os elementos menores (menores que x) antes de x, e colocar todos os elementos maiores (maiores que x) depois x. Tudo isso deve ser feito em tempo linear.

Tempo gasto pelo QuickSort, em geral, pode ser escrita da seguinte forma.
T (n) = T (k) + T (nk-1) + \ theta(n)
Os primeiros dois termos são para duas chamadas recursivas, o último termo é para o processo de partição. k é o número de elementos menores que o pivô.
O tempo gasto pelo QuickSort depende do array de entrada e da estratégia de partição.

Pior caso: o pior caso ocorre quando o processo de partição sempre seleciona o maior ou o menor elemento como pivô. Se considerarmos a estratégia de partição acima, onde o último elemento é sempre escolhido como pivô, o pior caso ocorreria quando o array já estivesse classificada em ordem crescente ou decrescente. A seguir está a recorrência para o pior caso.
T (n) = T (0) + T (n-1) + \ theta(n)
que é equivalente a
T (n) = T (n-1) + \ theta(n)
A solução da recorrência acima é \ theta(n 2).

Melhor caso: o melhor caso ocorre quando o processo de partição sempre escolhe o elemento do meio como pivô. A seguir está a recorrência para o melhor caso.
T (n) = 2T (n / 2) + \ theta(n)
A solução da recorrência acima é \ theta (nLogn).

Shell Sort
Melhor caso: O(n log n)
Caso médio: O(n) total, O(1) auxiliar
Pior caso: O(n2)

ShellSort é principalmente uma variação do Insert Sort . No Insert Sort, movemos os elementos apenas uma posição à frente. Quando um elemento precisa ser movido muito à frente, muitos movimentos estão envolvidos. A ideia do shellSort é permitir a troca de itens

distantes. Em shellSort, criamos o array h-ordenado para um grande valor de h. Continuamos reduzindo o valor de h até que se torne 1. Diz-se que um array está ordenado em h se todas as sublistas de cada h'ésimo elemento forem ordenadas.

Complexidade de tempo: A complexidade de tempo da implementação do shellsort é O(n2). O algoritmo implementado é reduzido pela metade em cada iteração. Existem muitas outras maneiras de reduzir a lacuna que levam a uma maior complexidade de tempo.

Merge Sort
Melhor caso: Ω(n log(n))
Caso médio: θ(n log(n))
Pior caso: O(n log(n))
Assim como QuickSort , Merge Sort é um algoritmo de divisão e conquista . Ele divide o array de entrada em duas metades, chama a si mesmo para as duas metades e, em seguida, mescla as duas metades ordenadas. A função merge() é usada para mesclar duas metades. O merge(arr, l, m, r) é um processo-chave que assume que arr [l..m] e arr[m + 1..r] são ordenados e mescla os dois sub-arrays ordenados em um.

Complexidade de tempo: ordenação de arrays em máquinas diferentes. Merge Sort é um algoritmo recursivo e a complexidade do tempo pode ser expressa como a seguinte relação de recorrência.
T (n) = 2T (n / 2) + θ (n)
A recorrência acima pode ser resolvida usando o método da árvore de recorrência ou o método mestre. Ele se enquadra no caso II do Método Mestre e a solução da recorrência é θ (nLogn).
A complexidade de tempo da classificação por mesclagem é θ (nLogn) em todos os 3 casos (pior, média e melhor), pois a classificação por mesclagem sempre divide o array em duas metades e leva tempo linear para mesclar as duas metades.

Radix Sort
Melhor caso: Ω(nk)
Caso médio: θ(nk)
Pior caso: O(nk)
A ideia do Radix Sort é fazer a ordenação dígito a dígito, começando do dígito menos significativo ao dígito mais significativo. O Radix Sort usa a ordenação por contagem como uma sub-rotina para ordenar.
Exemplo:
Lista original não ordenada:
170, 45, 75, 90, 802, 24, 2, 66

Ordenando pelo dígito menos significativo (1 casa) dá:
[* Observe que mantemos 802 antes de 2, porque 802 apareceu primeiro antes de 2 na lista original , e da mesma forma para os pares 170 e 90 e 45 e 75.]

17<u>0</u> , 9<u>0</u>, 80<u>2</u> , <u>2</u>, 2<u>4</u> , 4<u>5</u> , 7<u>5</u>, 6<u>6</u>

Ordenando pelo próximo dígito (casa 10s) dá:
[* Observe que 802 vem novamente antes de 2, assim como 802 vem antes de 2 na lista anterior.]

802, 2, 24, 45, 66, 170, 75, 90

Classificando pelo dígito mais significativo (casa da 100) dá:
2, 24, 45, 66, 75, 90, 170, 802

Tempo de execução do Radix Sort
Se houver d dígitos inteiros de entrada. O Radix Sort leva tempo O(d * (n + b)) onde b é a base para representar números, por exemplo, para o sistema decimal, b é 10. Qual é o valor de d? Se k for o valor máximo possível, então d seria O(log b (k)).
Portanto, a complexidade de tempo geral é O ((n + b) * log b (k)). O que parece mais do que a complexidade de tempo de algoritmos de classificação baseados em comparação para um grande k. Vamos primeiro limitar k. Seja k <= n c onde c é uma constante. Nesse caso, a complexidade torna-se O (nLog b (n)). Mas ainda não supera os algoritmos de classificação baseados em comparação.
E se tornarmos o valor de b maior? Qual deve ser o valor de b para tornar linear a complexidade do tempo? Se definirmos b como n, obtemos a complexidade do tempo como O (n). Em outras palavras, podemos classificar um array de inteiros com um intervalo de 1 a n^c se os números forem representados na base n (ou cada dígito leva log 2 (n) bits).

> Prática:
Desenvolver um (1) programa (OO em Ling. C++) que permita ao usuário escolher o tamanho (quantidade) de elementos de um vetor (dinâmico, indexado). Logo, o programa deve preencher o vetor recém-alocado com valores inteiros aleatórios (gerando um vetor desordenado).
Em seguida, o programa usa este vetor aleatório para aplicar TODOS os algoritmos de ordenação acima (preserve o vetor original, crie cópia se necessário). Logo, compute o tempo de execução de cada algoritmo de ordenação para o mesmo vetor de origem (basta computar a diferença entre tempos inicial e final de cada função de algoritmo, pesquise pela biblioteca C time C++, <time>, <ctime>, p. ex.).
Obs.1: ao aumentar o tamanho (quantidade) grande de dados aleatórios, há tendência de obter maiores tempos de execução dos algoritmos. Manipule unidade de tempo adequadamente entre mili ou microsegundos para fins de apresentação de resultados.
Obs.2: Leitura recomendada: Cap. 9 (Ordenação) - livro-base (Adam Drozdek).

>> Enviar aquivo ZIP com material de resposta (pasta de prj. prático + PDF documentação).