Análise teórica e experimental de algoritmos

Flavio Matias D. Carvalho flavio.carvalho@eic.cefet-rj.br

Adalberto M. de Andrade adalberto.andrade@eic.cefet-rj.br

**Abstract.** Do we need abstract in our project?.

**Resumo.** O trabalho vai ter que ter resumo? resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo resumo também pode ser alterada de forma que haja uma melhor participação na sociedade pelos usu das .

1. Introdução

Este trabalho visa consolidar os conceitos apresentados na disciplina de Análise e Projeto de Algoritmos, ao mesmo tempo em que é exercitado o método científico na análise de algoritmos por meio da comparação teórica e experimental de algoritmos.  
  
O objetivo do trabalho é permitir que se comece a tomar contato com o método científico na área de análise de algoritmos, exercitando, ao mesmo tempo, abordagens teóricas e experimentais para a caracterização dos parâmetros de eficiência de um ou mais algoritmos. Dessa forma, optou-se por ambientes de programação e desenvolvimento de maior familiaridade e problemas computacionais e algoritmos que despertem seu interesse.  
Este trabalho busca responder aos seguintes questionamentos e alcançar os seguintes resultados:  
- Qual o tempo de execução "teórico" de cada algoritmo no pior caso, no caso médio e no melhor caso?  
- Quais as entradas que evidenciam/realizam o melhor caso e o pior caso?  
- O comportamento experimental evidencia as análise teóricas?  
- Se implementarmos os algoritmos em diferentes ambientes (implementações distintas, linguagens distintas, sistemas operacionais distintos, computadores distintos), é possível evidenciar o comportamento assimptótico semelhante?  
- De que forma se comportam diferentes algoritmos para o mesmo problema podem ser comparados?

Escolheu-se para esta finalidade o problema de ordenação, que pode ser resolvido de diversas maneiras por diferentes algoritmos de ordenação. Um algoritmo de ordenação coloca os elementos de uma dada sequência em uma certa ordem. Usou-se os tipos bubble sort, insertion sort, merge sort e quick sort. Os algoritmos foram implementados em diferentes ambientes (computadores distintos) de forma a ser possível evidenciar o comportamento assimptótico semelhante destes.

2. Literatura e trabalhos relacionados

O.

3. Algoritmos

Ordenar consiste em posicionar elementos de uma sequência de informações, ou dados, em uma ordem predefinida. Assim, dada uma seqüencia de n dados < a1, a2, a3, …, an> , o problema de ordenação é uma permutação dessa seqüencia < a’1, a’2, a’3, …, a’n>.

3.1 Bubble Sort

O algoritmo conhecido como bubble sort também é conhecido como ordenação por flutuação ou "por bolha". É um algoritmo de ordenação bem simples que percorre a lista diversas vezes, a cada passagem fazendo o maior elemento da sequência “flutuar” para o topo, como bolhas em um tanque de água que vão para seu próprio nível, e disso vem o nome do algoritmo.

Este algoritmo percorre a lista de itens ordenáveis do início ao fim, verificando a ordem dos elementos dois a dois, e trocando-os de lugar se necessário. Percorre-se a lista até que nenhum elemento tenha sido trocado de lugar na passagem anterior.

No melhor caso, o algoritmo executa *n* operações relevantes, onde n representa o número de elementos da lista. No pior caso, são feitas *n2* operações ou seja, a complexidade desse algoritmo é de Ordem quadrática.

3.2 Insertion Sort

O algoritmo de ordenação Insertion Sort, ou ordenação por inserção, é um algoritmo que, dado uma estrutura, como uma lista, constrói uma matriz final com um elemento de cada vez, uma inserção por vez.  
  
Para se entender o funcionamento da ordenação por Insertion Sort, pode ser feito um paralelo com a forma de como algumas pessoas organizam um baralho num jogo de cartas. Imagine que você está jogando cartas. Você está com as cartas na mão e elas estão ordenadas. Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas obedeçam a ordenação.  
  
A cada nova carta adicionada a sua mão de cartas, a nova carta pode ser menor que algumas das cartas que você já tem na mão ou maior, e assim, você começa a comparar a nova carta com todas as cartas na sua mão até encontrar sua posição correta. Você insere a nova carta na posição correta, e, novamente, sua mão é composta de cartas totalmente ordenadas. Então, você recebe outra carta e repete o mesmo procedimento. Então outra carta, e outra, e assim por diante, até você não receber mais cartas.  
  
Esta é a ideia por trás da ordenação por inserção. Percorra as posições do array, começando com o índice 1 (um). Cada nova posição é como a nova carta que você recebeu, e você precisa inseri-la no lugar correto no subarray ordenado à esquerda daquela posição.

4. Análise teórica dos algoritmos

4.1. Insertion Sort

4.1.1.Características

🡺 É eficiente na ordenação de um número pequeno de elementos.

🡺 Funciona de forma análoga da maneira que ordenamos cartas de baralho na mão.

🡺 Para entender o porquê deste algoritmo funciona de forma correta temos que provar três coisas:

🡺 Inicialização: Invariante é verdadeira antes da primeira iteração.

🡺Manutenção: Se é verdadeiro no começo da iteração se manterá verdadeiro também na seguinte.

🡺Término: Quando o ciclo for concluído o array estará ordenado de forma correta.

Estes três passos podem serem comparados as propriedades do método da indução matemática como abaixo apresentado:

Inicialização: Base de indução

Manutenção: Passo indutivo

Término: A indução é concluída quando o ciclo termina.

4.1.2. Desempenho

🡺 O melhor caso é quando os elementos já estão ordenados.

🡺 E o pior caso é quando os elementos do vetor estão na ordem inversa.

4.1.3. Pseudocódigo exemplo:

INSERTION-SORT (A)

1 for j = 2 to A.length

2 key = A[j]

3 i = j – 1

4 while i > 0 and A[i] > key

5 A[i + 1] = A[i]

6 i = i - 1

7 A[i + 1] = key

4.2. Quicksort

4.2.1. Característica:

🡺 Algoritmo recursivo

🡺 Estratégia de divisão e conquista

🡺 Reduz o trabalho necessário para resolver o problema

4.2.2. Ideia:

🡺 Dividir o problema em dois subproblemas menores

🡺 Cada um é tratado de forma independente

🡺 E combinar os resultados para produzir a solução final

🡺 A divisão é feita baseada em pivô (que é um elemento do vetor)

🡺 A reorganização do vetor resulta em elementos a esquerda do pivô menores do que ele.

🡺 E análogo para o lado direito do vetor

🡺 Utiliza duas funções: Uma de ordenação e outra de particionamento

4.2.3. Eficiência

🡺 Complexidade é de O(n log n) para o melhor caso e para o caso médio

🡺 No pior caso tempo de execução pode chegar a O(n²)

🡺A eficiência é semelhante ao Mergesort

🡺O tempo de execução vai depender não apenas da quantidade de elementos, mas do quanto eles estão desbalanceados ou não no vetor.

4.2.4. Pseudocódigo exemplo:

QUICKSORT (A, p, r)

1 if p < r

2 q = PARTITION (A,p,r)

3 QUICKSORT (A, p, q – 1)

4 QUICKSORT (A, q + 1, r)

Para ordenar um vetor inteiro A, a chamada inicial é QUICKSORT (A,1, A.length)

PARTITION procedure

PARTITION (A, p,r)

1 x = A[r]

2 i = p - 1

3 for j = p to r - 1

4 if A[j] ≤ x

5 i = i + 1

6 exchange A[i] with A[j]

7 exchange A[i + 1] with A[r]

8 return i + 1

5. Análise experimental dos algoritmos

Em.

6. Resultados experimentais

Em.

Referências

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Algoritmos (3rd ed.). MIT Press and McGraw-Hill.