Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

1 O algoritmo *quicksort*

O algoritmo Quicksort, assim como *merge sort*, utiliza o paradigma de *divisão e conquista* para ordenar um vetor $A[1..n]$. As etapas para ordenar um subvetor $A[p..r]$ $(1 \leq p \leq r \leq n)$ são as seguintes:

1. **Dividir**: Esta etapa consiste em particionar o vetor $A[p..r]$ em dois subvetores (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ tais que cada elemento do vetor $A[p..q-1]$ é menor ou igual a $A[q]$, que por sua vez também é menor ou igual do que cada elemento do vetor $A[q+1..r]$. Esta etapa também computa o índice q do particionamento.

2. **Conquistar**: Esta etapa consiste em recursivamente ordenar os subvetores $A[p..q-1]$ e $A[q+1..r]$.

Assim, a ideia do algoritmo pode ser apresentada a partir do pseudocódigo a seguir:

Algorithm 1: Quicksort(A, p, r)

```
1  if p < r then
2  |  q = Partition(A, p, r);
3  |  Quicksort(A, p, q − 1);
4  |  Quicksort(A, q + 1, r);
5  end
```

A ordenação do vetor $A[1..n]$ é, então, obtida pela chamada Quicksort$(A, 1, n)$. O particionamento do vetor consiste na principal etapa do algoritmo Quicksort:

Algorithm 2: Partition(A, p, r)

```
1  x = A[r];
2  i = p − 1;
3  for j = p to r − 1 do
4  |  if A[j] ≤ x then
5  |  |  i = i + 1;
6  |  |  exchange A[i] com A[j];
7  |  end
8  end
9  exchange A[i + 1] with A[r];
10 return i + 1;
```

Exercício 1.1. *Prove a seguinte invariante de laço, e conclua que o algoritmo* Partition *é correto:*

> *Antes de cada iteração do laço* ***for*** *(linhas 3-8), para todo k, temos:*
>
> 1. *Se $p \leq k \leq i$, então $A[k] \leq x$;*
> 2. *Se $i + 1 \leq k \leq j - 1$, então $A[k] > x$;*
> 3. *Se $k = r$, então $A[k] = x$.*

*flaviomoura@unb.br

O número de comparações (linha 4) feitas por Partition em um vetor com n elementos é $\Theta(n)$. Qual seria, então, o tempo de execução de Quicksort?

O tempo de execução $T(n)$ de Quicksort, no pior caso, para entradas de tamanho n ocorre quando o particionamento gera um vetor vazio, e outro com $n-1$ elementos. Neste caso dizemos que o particionamento é *desbalanceado*. Se assumirmos que este desbalanceamento ocorre em cada chamada recursiva, podemos modelar o processo pela seguinte equação de recorrência:

$$T_w(n) = T_w(n-1) + T_w(0) + \Theta(n)$$

Como não há trabalho a fazer em um vetor vazio, temos que $T(0) = \Theta(1)$, e portanto a recorrência acima pode ser reescrita como:

$$T_w(n) = T_w(n-1) + \Theta(n) \tag{1}$$

Como exercício, mostre que $T_w(n) = \Theta(n^2)$, e observe que esta situação ocorre, por exemplo, quando o vetor dado como argumento já está ordenado.

Por outro lado, quando o particionamento é balanceado temos o melhor caso para Quicksort. Agora, o particionamento divide o problema original em dois subproblemas sendo um de tamanho $\lfloor \frac{n}{2} \rfloor$, e outro de tamanho $\lceil \frac{n}{2} - 1 \rceil$, o que nos dá a recorrência:

$$T_b(n) = T_b(\lfloor \frac{n}{2} \rfloor) + T_b(\lceil \frac{n}{2} - 1 \rceil) + \Theta(n)$$

Se ignorarmos as funções de aproximação e a subtração por 1, temos a recorrência:

$$T_b(n) = 2.T_b(\frac{n}{2}) + \Theta(n)$$

que, pelo Teorema Mestre, tem solução $T_b(n) = \Theta(n.\lg n)$.

Assim, o tempo de execução de Quicksort depende se o particionamento está balanceado ou não: Em caso afirmativo, Quicksort é assintoticamente tão rápido quanto *mergesort*, mas se o particionamento não estiver balanceado, Quicksort tem o mesmo comportamento assintótico de *Insertion sort* no pior caso.

O que ocorre se o particionamento é sempre da ordem de 9-1, *i.e.* 90% dos elementos são menores ou iguais ao pivô, e apenas 10% são maiores do que o pivô? (Exercício!) A resposta desta pergunta sugere que a complexidade do caso médio para Quicksort está mais próxima do melhor caso do que do pior caso.

Exercício 1.2. *Mostre que a complexidade de Quicksort, no melhor caso, é* $\Omega(n \lg n)$.

Exercício 1.3. *Mostre que o algoritmo Quicksort é correto.*

2 Cota inferior para algoritmos baseados na comparação de chaves

Os algoritmos de ordenação estudados até aqui são baseados na comparação de chaves, ou seja, a operação básica destes algoritmos é a comparação entre elementos do vetor (ou lista) que se quer ordenar. Neste contexto, vimos que *Insertion Sort* e *Quicksort* têm, no pior caso, complexidade de tempo quadrática, *i.e.* estão em $O(n^2)$. Já *Merge sort* é capaz de, no pior caso, ordenar um vetor com n elementos em tempo $O(n.\lg n)$. A dúvida que surge naturalmente agora é: seria possível construir um algoritmo baseado na comparação de chaves que consiga ordenar um vetor com n elementos, no pior caso, realizando menos do que $c.n.\lg n$ comparações para alguma constante positiva c? Ou seja, seria possível melhorar a cota $O(n.\lg n)$ no pior caso?

Para responder a esta pergunta, assumiremos que os n elementos $x_1, x_2, \ldots, x_n$ a serem ordenados são distintos. Além disto, o processo de ordenação será modelado por *árvores de decisão*, que são árvores (binárias) completas que representam as comparações realizadas entre elementos por um algoritmo de ordenação particular em uma entrada de tamanho dado. Assim, para construirmos a árvore de decisão para um algoritmo A (baseado na comparação de chaves) que recebe como entrada um vetor com n elementos distintos, anotaremos seus nós internos com $i : j$, para $1 \leq i, j \leq n$, e cada folha com a permutação $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. A execução do algoritmo corresponde a um caminho da raiz até uma folha. Cada nó interno corresponde a comparação $a_i \leq a_j$ (a rigor $a_i < a_j$ já que os elementos são

distintos), e a subárvore à esquerda indica as comparações subsequentes. A subárvore à direita indica as comparações subsequentes quando $a_i > a_j$, e uma folha indica que o algoritmo ordenou o vetor de forma que $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$. Como qualquer algoritmo de ordenação correto tem que ser capaz de produzir qualquer uma das permutações de uma entrada, cada uma das $n!$ possíveis permutações do vetor de tamanho n dado com entrada tem que aparecer como uma folha. O caminho mais longo da raiz de uma árvore de decisão até uma folha representa o número de comparações no pior caso do algoritmo em consideração. Ou seja, o número de comparações no pior caso de um algoritmo corresponde a altura de sua árvore de decisão. A cota inferior da altura de todas as possíveis árvores de decisão será então a cota inferior, no pior caso, para qualquer algoritmo de ordenação baseado na comparação de chaves.

Teorema 2.1. *Qualquer algoritmo baseado na comparação de chaves requer* $\Omega(n. \lg n)$ *comparações no pior caso.*

Demonstração. Precisamos determinar a altura de uma árvore de decisão onde cada permutação do vetor de entrada aparece como uma folha. Considere a árvore de decisão de altura h contendo l folhas de um algoritmo para ordenar n elementos distintos. Como cada uma das $n!$ permutações do vetor de entrada aparece como uma folha, temos que $n! \leq l$, e como uma árvore binária de altura h não possui mais do que 2^h folhas, temos que $n! \leq l \leq 2^h$. Portanto $h \geq \lg(n!) = \Omega(n. \lg n)$. □

Exercício 2.2. *Sejam* l *o número de folhas em uma árvore binária, e* h *sua altura. Prove que* $l \leq 2^h$.

Exercício 2.3. *Prove que* $\lg(n!) = \Theta(n. \lg n)$.