MAC 5711 - Análise de Algoritmos

Departamento de Ciência da Computação

Primeiro semestre de 2004

Aulas de 18/3 e 23/3 - Quicksort

O assunto destas aulas foi o algoritmo quicksort. Consideramos inicialmente a seguinte implementação deste algoritmo:

```
0.  QuickSort(A, p, r)
1.      se p < r
2.          q := Particiona(A, p, r);
3.          QuickSort(A, p, q);
4.          QuickSort(A, q + 1, r);
```

com a função Particiona implementada da seguinte maneira.

```
0.  Particiona(A, p, r)
1.      x := A[p];
2.      i := p − 1;
3.      j := r + 1;
4.      enquanto i < j faça
5.          repita i := i + 1 até que A[i] ≥ x;
6.          repita j := j − 1 até que A[j] ≤ x;
7.          se (i < j)
8.              troca(A[i], A[j]);
9.      devolva j;
```

Os dois seguintes invariantes valem durante a execução da função Particiona.

(P1) Entre as linhas 4 e 5, existe u tq $i < u \leq r$ e $A[u] \geq x$ e existe v tq $p \leq v < j$ e $A[v] \leq x$;

(P2) Entre as linhas 6 e 7, tem-se que $i \leq r$ e $j \geq p$.

Esses dois invariantes permitem primeiramente concluir que o Particiona pára após um número finito de iterações. De fato, os dois comandos repita, das linhas 5 e 6, sempre param, pelo invariante (P1). Além disso, a cada iteração do enquanto da linha 4, o valor de $j - i$, que é sempre inteiro e inicialmente é $r - p + 2$, diminui até ficar menor ou igual a zero. Na verdade, essa análise permite concluir que o Particiona consome tempo $\Theta(n)$, onde $n := r-p+1$.

Pode-se concluir também a partir dos invariantes que, imeditamente antes da linha 9, $p \leq j < i \leq r$, ou seja, o valor de q na linha 2 do quicksort é tal que $p \leq q < r$. Isso garante, por sua vez, que as duas chamadas recursivas do quicksort tratam de vetores com menos elementos que o vetor original, ou seja, a recursão do quicksort termina.

Temos portanto a seguinte recorrência para o tempo consumido pelo quicksort:

$$T(n) \quad = \quad T(k) + T(n-k) + \Theta(n), \tag{1}$$

onde $n = r - p + 1$ é o número de elementos no trecho que será ordenado e $k = q - p + 1$ é o número de elementos no lado esquerdo da partição. Como já foi observado, $1 \leq k \leq n - 1$. Evidentemente, a solução desta recorrência depende do valor de k.

Análise do pior caso do Quicksort

1. O pior caso do quicksort é $\Omega(n^2)$.

   Para mostrar isso, precisamos mostrar uma instância (uma seqüência para cada n) em que o algoritmo consome tempo $\Theta(n^2)$.

   Se a seqüência de entrada estiver ordenada em ordem crescente, a função Particiona sempre deixa apenas um elemento do lado esquerdo, ou seja, $k = 1$ na recorrência, que se reduz neste caso a

   $$T(n) \;=\; T(1) + T(n-1) + \Theta(n) \;=\; c + T(n-1) + \Theta(n) \;=\; T(n-1) + \Theta(n).$$

   Exercício: A solução desta recorrência é $T(n) = \Theta(n^2)$.

   Ou seja, o tempo cosumido pelo quicksort quando a entrada é uma seqüência ordenada é $\Theta(n^2)$ (onde n é o tamanho da seqüência).

   Isso implica que, no pior caso, o quicksort é $\Omega(n^2)$. Em outras palavras, esse caso mostra uma delimitação inferior para o tempo de pior caso do quicksort. Porém ainda não descarta a possibilidade de existir alguma instância onde o quicksort se comporta pior ainda, por exemplo, consumindo tempo $\Theta(n^3)$.

2. O pior caso do quicksort é $O(n^2)$.

   Para mostrar isso, precisamos mostrar que, **em qualquer intância**, o quicksort consome tempo $O(n^2)$.

   Isso é feito provando-se que a solução da seguinte recorrência, derivada de (1), é $O(n^2)$.

   $$T(n) \;=\; \max_{1 \le k \le n-1} \{T(k) + T(n-k)\} + n. \tag{2}$$

   Mais exatamente, mostraremos, por indução em n, que a solução de (2) satisfaz $T(n) \le n^2$, para todo $n \ge 1$. (Observe que já substituímos o $\Theta(n)$ por n, e que consideraremos $T(1) = 1$.)

   Base: $n = 1$.

   Vale trivialmente já que $T(1) = 1 \le 1^2$.

   Passo: $n \ge 2$.

   Suponha (hipótese de indução) que $T(k) \le k^2$ para todo $1 \le k < n$. Então temos que

   $$\begin{aligned} T(n) &= \max_{1 \le k \le n-1} \{T(k) + T(n-k)\} + n \\ &\le \max_{1 \le k \le n-1} \{k^2 + (n-k)^2\} + n \quad \text{(pela hipótese de indução)} \\ &= \max_{1 \le k \le n-1} \{2k^2 - 2nk + n^2\} + n \\ &= (2 - 2n + n^2) + n \quad \text{(pois o máximo de } 2k^2 - 2nk + n^2 \text{ é atingido qdo } k = 1) \\ &= n^2 - n + 2 \\ &\le n^2 \quad (\text{pois } n \ge 2). \end{aligned}$$

   Isso termina a prova de que $T(n) \le n^2$ para todo $n \ge 1$. Assim, $T(n) = O(n^2)$, ou seja, qualquer que seja a instância, o quicksort consome tempo $O(n^2)$.

Juntando os dois casos acima, concluímos que a complexidade de pior caso do quicksort é $\Theta(n^2)$.

Versões probabilísticas do Quicksort e análise do caso médio

A análise de pior caso do quicksort sugere que este algoritmo poderia ser pior do que o mergesort, cuja complexidade de pior caso é $\Theta(n \log n)$. No entanto, na prática, observa-se que o quicksort é o mais rápido dos algoritmos de ordenação de propósito geral.

Para entender melhor esse fenômeno, precisamos fazer uma análise mais cuidadosa da complexidade do quicksort: uma análise do comportamento *esperado* do quicksort. Análises de comportamento esperado ou, como se diz algumas vezes, de caso médio, são geralmente mais complexas que análises de pior caso. Um dos primeiros problemas em que se esbarra quando se pensa em fazer uma análise de caso médio é o seguinte: qual é a distribuição de probabilidade sobre as instâncias? Qual é a chance de cada uma das instâncias aparecer como entrada do algoritmo? Sem ter uma resposta precisa para essa pergunta, não há como calcular o tempo esperado que o algoritmo levará para concluir seu processamento. Pior ainda... essa informação não é bem definida! Em cada circunstância, a distribuição pode ser diferente!

No caso do quicksort, há uma maneira elegante de evitar-se esse problema: basta considerarmos versões *probabilísticas* do algoritmo. Para tais versões, pode-se fazer uma análise do caso médio sem tomar conhecimento da distribuição de probabilidade das instâncias. (Um algoritmo probabístico é um algoritmo cujo comportamento é determinado não apenas pela instância em questão, mas também pelo resultado de chamadas de funções como a função `rand()`, disponível na linguagem `C`.)

Existem duas variantes probabilísticas simples do quicksort. Uma consiste em, simplesmente, antes de acionar o quicksort tradicional, substituir a seqüência original por uma permutação dela, escolhida dentre todas as possíveis permutações, de acordo com a distribuição uniforme de probabilidade (isto é, cada permutação tem a mesma chance de ser escolhida). Daqui até o final dessas notas, assumiremos que os elementos da seqüência em questão são dois a dois distintos (ou seja, não há valores repetidos na seqüência). Isso facilita a análise sem comprometer as conclusões.

A segunda maneira de se obter um quicksort probabilístico consiste em, antes de se acionar a função Particiona, escolher-se um índice i aleatoriamente entre p e r, de acordo com a distribuição uniforme, e trocar os elementos $A[i]$ e $A[p]$ de posição. Ou seja, substituir a chamada à função Particiona na linha 2 do quicksort pela chamada da seguinte variante probabilística dessa:

0. **Particiona_Probabilístico** (A, p, r)
1. i := random(p, r);
2. troca$(A[i], A[p])$;
3. devolva Particiona(A, p, r);

É essa segunda variante do quicksort que analisaremos.

Primeiramente, note que o pior caso dessa variante probabilística do quicksort (e da outra também) tem complexidade de pior caso $\Theta(n^2)$, exatamente como o quicksort original. A única diferença é que, em qualquer instância, o algoritmo pode consumir tempo $\Theta(n^2)$. Basta ele ser azarado (mas bem azarado!) ☺ nas escolhas aleatórias. A demonstração de que a complexidade de pior caso é $O(n^2)$ é exatamente a mesma.

Agora vamos mostrar que a complexidade esperada do algoritmo é $O(n \log n)$. Mais especificamente, vamos mostrar que o tempo esperado de consumo do algoritmo para uma dada instância é $O(n \log n)$.

A recorrência do caso médio do quicksort é a seguinte:

$$T(n) \quad = \sum_{1 \le t \le n-1} \Pr[X = t](T(t) + T(n-t)) + \Theta(n), \tag{3}$$

onde X é a variável aleatória cujo valor é o número de elementos do intervalo que ficaram do lado esquerdo da partição.

Analisando o comportamento da função Particiona_Probabilístico, deduzimos que

$$\Pr[X = t] \quad = \quad \begin{cases} 2/n & \text{se } t = p \\ 1/n & \text{se } p < t < r. \end{cases} \tag{4}$$

De fato, a função devolve p se e somente se o pivô x da linha 1 é o menor elemento de $A[p\,..\,r]$ ou o segundo menor elemento de $A[p\,..\,r]$. A função devolve j, com $p < j < r$, se e somente se o pivô x for exatamente o $(j+1)$-ésimo menor elemento de $A[p\,..\,r]$. (Verifique isso.)

Com isso, podemos prosseguir com a análise do caso médio. Substituindo (4) em (3), obtemos que

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{1 \le t \le n-1} (T(t) + T(n-t)) + \frac{1}{n}(T(1) + T(n-1)) + \Theta(n) \\ &= \frac{1}{n} \sum_{1 \le t \le n-1} (T(t) + T(n-t)) + \Theta(n), \end{aligned}$$

pois $\frac{1}{n}(T(1) + T(n-1)) = O(n)$, já que sabemos que $T(n) = O(n^2)$. Observe que cada $T(t)$ aparece duas vezes no somatório (uma vez como a primeira parcela, outra vez como a segunda). Assim, reorganizando o somatório e substituindo $\Theta(n)$ por n, obtemos a recorrência

$$T(n) \quad = \quad \frac{2}{n} \sum_{1 \le t \le n-1} T(t) + n. \tag{5}$$

Vamos mostrar por indução em n que $T(n) \le 4n \log n + 2n$ para todo $n \ge 1$.

Base: $n = 1$.

Trivial, já que $T(1) = 1 \le 2 = 4.1.0 + 2.1$.

Passo: $n \ge 2$.

Suponha (hipótese de indução) que $T(t) \le 4t \log t + 2t$, para todo t tal que $1 \le t < n - 1$. Temos que

$$\begin{aligned} T(n) &= \tfrac{2}{n} \textstyle\sum_{t=1}^{n-1} T(t) + n \qquad \text{(por (5))} \\ &\le \tfrac{2}{n} \textstyle\sum_{t=1}^{n-1} (4t \log t + 2t) + n \qquad \text{(por hipótese de indução)} \\ &\le \tfrac{2.4}{n} \textstyle\sum_{t=1}^{\lceil \frac{n}{2} \rceil - 1} t \log \tfrac{n}{2} + \tfrac{2.4}{n} \textstyle\sum_{t=\lceil \frac{n}{2} \rceil}^{n-1} t \log n + \tfrac{2}{n} \textstyle\sum_{t=1}^{n-1} 2t + n \\ &= \tfrac{8}{n}(\log n - 1) \textstyle\sum_{t=1}^{\lceil \frac{n}{2} \rceil - 1} t + \tfrac{8}{n} \log n \textstyle\sum_{t=\lceil \frac{n}{2} \rceil}^{n-1} i + \tfrac{4}{n} \textstyle\sum_{t=1}^{n-1} t + n \\ &= \tfrac{8}{n} \log n \textstyle\sum_{t=1}^{n-1} t - \tfrac{8}{n} \textstyle\sum_{t=1}^{\lceil \frac{n}{2} \rceil - 1} t + 2(n-1) + n \\ &= 4(n-1) \log n - \tfrac{4}{n}(\lceil \tfrac{n}{2} \rceil (\lceil \tfrac{n}{2} \rceil - 1)) + 2(n-1) + n \\ &\le 4n \log n - 4 \log n - 2(\tfrac{n}{2} - 1) + 2(n-1) + n \\ &= 4n \log n + 2n - 4 \log n \\ &\le 4n \log n + 2n \quad \text{(pois } n \ge 2\text{)}. \end{aligned}$$

Isso completa a prova de que $T(n) \le 4n \log n + 2n$ para todo $n \ge 1$, donde podemos concluir que $T(n) = O(n \log n)$. Em outras palavras, acabamos de demonstrar que a complexidade de caso médio da segunda variante probabilística do quicksort é $O(n \log n)$.