Análise de Algoritmos – Tópico 12

Prof. Dr. Juliano Henrique Foleis

QuickSort - Introdução

Inventado em 1959 pelo cientista britânico C.A.R Hoare, o `QUICKSORT` é um dos algoritmos mais famosos da Ciência da Computação e é amplamente utilizado até os dias de hoje. Embora o pior caso do `QUICKSORT` seja $\Theta(n^2)$ para um vetor com n elementos, é o algoritmo mais prático para ser usado por executa em tempo $\Theta(n \lg n)$ no caso médio. Além disto, os fatores constantes escondidos pela notação assintótica em $\Theta(n \lg n)$ são normalmente menores que os encontrados nos algoritmos `HEAPSORT` e `MERGESORT`. Além disto, ao contrário de alguns algoritmos de ordenação, a ordenação acontece *in loco* e funciona bem até em ambientes com memória virtual, indispensáveis hoje em dia.

O algoritmo

```
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)
```

Baseado em divisão e conquista, o algoritmo para ordenar $A[p, \ldots, r]$ pode ser descrito por:

- **DIVISÃO** – Particionar o vetor $A[p, \ldots, r]$ em dois subvetores potencialmente vazios $A[p, \ldots, q-1]$ e $A[q+1, \ldots, r]$ de forma que os elementos $A[p, \ldots, q-1]$ sejam menores ou iguais a $A[q]$, que, por sua vez, é menor ou igual a todos os elementos em $A[q+1, \ldots, r]$.
- **CONQUISTA** – Ordenar $A[p, \ldots, q-1]$ e $A[q+1, \ldots, r]$ recursivamente, usando `QUICKSORT`.
- **COMBINAÇÃO** – Como os subvetores $A[p, \ldots, q-1]$ e $A[q+1, \ldots, r]$ já estão ordenados, trabalho adicional não é necessário para combiná-los: o subvetor $A[p, \ldots, r]$ está ordenado.

Particionamento

O procedimento `PARTITION` é responsável pela fase de divisão do `QUICKSORT` e funciona reorganizando o subvetor $A[p \ldots r]$ *in loco*. O algoritmo `PARTITION` apresentado a seguir é foi desenvolvido por Lomuto, e não foi o `PARTITION` original desenvolvido por Hoare.

```
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.          troca A[i] e A[j]
7.      END IF
8.  END FOR
9.  troca A[i+1] e A[r]
10. RETURN i+1
```

`PARTITION` sempre escolhe $x = A[r]$ como **pivô**, o elemento utilizado como divisor do subvetor. Enquanto executa, o procedimento particiona o vetor em quatro regiões. No início de cada iteração FOR das linhas 3–8, as regiões satisfazem certas propriedades. Podemos utilizar estas propriedades como invariantes de laço para provar que `PARTITION` está correto.

Correção do PARTITION

Invariante de Laço – No início de cada iteração do laço "FOR" das linhas 3–8, para qualquer índice k,

- Se $p \leq k \leq i$, então $A[k] \leq k$
- Se $i+1 \leq k \leq j-1$, então $A[k] > x$
- Se $k = r$, então $A[k] = x$

A Figura 1 abaixo mostra as quatro regiões que o vetor está particionado durante a execução de partition, bem como a relação entre as regiões, o enunciado da invariante de laço e as variáveis de controle do algoritmo de particionamento.

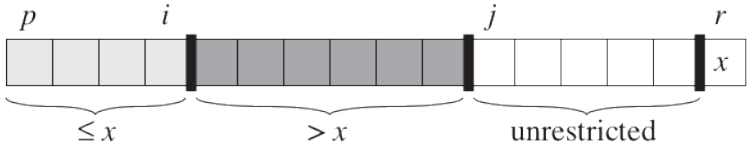

Figura 1: Representação gráfica da invariante de laço do algoritmo `PARTITION` (CRLS, 2012)

Lembrando que os elementos em j e $r-1$ não fazem parte dos casos e os valores destas relações não tem relação particular com o valor. Assim, precisamos mostrar que esta invariante de laço é verdadeira imediatamente antes da primeira iteração, que cada iteração do laço mantém a invariante válida para a próxima iteração e que a invariante provê uma propriedade útil para mostrar que o algoritmo está correto quando o laço termina.

INICIALIZAÇÃO – Antes da primeira iteração do laço, $i = p-1$ e $j = p$. Como não há elementos entre p e i e não há elementos entre $i+1$ e $j-1$, as duas primeiras propriedades são trivialmente verdadeiras. A atribuição da linha 1 satisfaz a terceira propriedade.

MANUTENÇÃO – Consideremos os 2 casos possíveis: o caso onde a condicional da linha 4 é verdadeira e o caso falso. A Figura 2 ilustra o que acontece em ambos os casos. Quando $A[j] > x$, a única operação é incrementar j. Assim que j é incrementado a propriedade 2 é verdadeira para $A[j-1]$ e as outras condições se mantém inalteradas. Em outras palavras, o elemento sendo analisado pela condicional ($A[j]$) é um número maior que o pivô e já está na partição certa, uma vez que os elementos maiores que o pivô, por invariante de laço, estão entre $i+1$ e $j-1$.

No caso $A[j] \leq x$, i é incrementado, troca $A[i]$ e $A[j]$ e então j é incrementado. Devido à troca, temos que $A[i] \leq x$, e a propriedade 1 é satisfeita. Da mesma forma, $A[j-1] > x$, uma vez que o item que foi trocado para a posição $j-1$ (que estava em $A[i]$, o primeiro maior que x após a instrução de incremento) é, por invariante de laço, maior que x, satisfazendo a segunda propriedade. A terceira propriedade continua verdadeira uma vez que x não é atribuído valores durante a execução do laço e que j nunca assume o valor r.

TÉRMINO – No término, $j = r$. Assim, como a partição $A[j \ldots r-1] = A[j \ldots j-1]$ é vazia, cada elemento do vetor $A[p, \ldots, r-1]$ está em suas respectivas partições descritas pela invariante. Desta forma, $A[p \ldots i]$ contém os elementos menores ou iguais a $A[r]$ e $A[i+1 \ldots j-1] = A[i+1 \ldots r-1]$ contém os elementos maiores que $A[r]$.

Após o término do laço, o vetor está quase no formato necessário por `QUICKSORT` para continuar a ordenação. Para obter este formato, as duas últimas linhas de PARTITION trocam o pivô com o primeiro elemento maior que x, que pela invariante de laço está na posição $i+1$ colocando-o na posição correta no vetor (após o último menor que o pivô e antes do"novo" primeiro maior que ele), e então retorna a posição final do pivô. Portanto, as modificações realizadas no vetor e o valor de retorno satisfazem as especificações do passo da divisão de `QUICKSORT`. Desta forma, o algoritmo `PARTITION` está correto.

Na realidade, `PARTITION` satisfaz uma condição ainda mais forte. Após a linha 2 de `QUICKSORT`, $A[q]$ é estritamente menor que todo elemento de $A[q+1, \ldots, r]$, fazendo que o elemento $A[q]$ esteja em seu local final da ordenação. O tempo de execução de `PARTITION`(A, p, r) é $\Theta(n)$ onde $n = r - p + 1$.

Desempenho do `QUICKSORT`

O desempenho do `QUICKSORT` depende do balanceamento obtido por `PARTITION`, que depende da escolha do pivô e da relação dos valores dos elementos. Se o particionamento é balanceado, ou seja, o pivô "divide" o subvetor em 2 partições

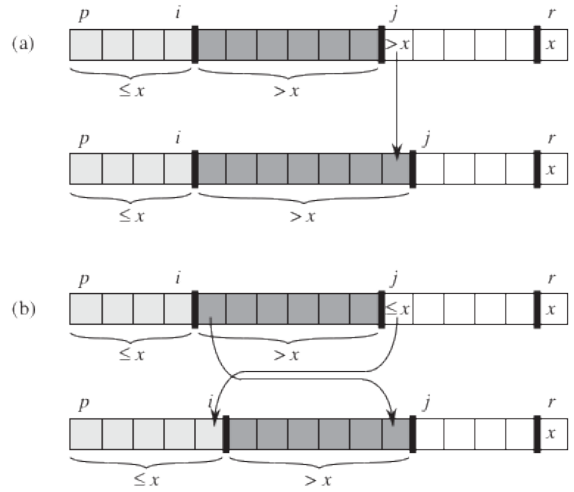

Figura 2: Os dois casos possíveis para uma iteração do `PARTITION`. Em (a) $A[j] > x$. Em (b), $A[j] \leq x$. (CRLS, 2012)

com quantidades praticamente iguais de elementos, o algoritmo possui complexidade assintótica idêntica ao `HEAPSORT` e `MERGESORT`. Caso contrário, a complexidade assintótica é a mesma que do ineficiente `BUBBLESORT`. Primeiramente, analisamos informalmente o desempenho do `QUICKSORT` em relação ao balanceamento do particionamento.

Particionamento no Pior Caso

O pior caso do `QUICKSORT` acontece quando `PARTITION` produz dois subproblemas: um com 0 elementos e outro com $n-1$ elementos em todas as chamadas recursivas em `QUICKSORT`. Como o particionamento tem custo $\Theta(n)$ e `QUICKSORT`(0) tem custo constante, a recorrência neste caso é:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

Intuitivamente, ao somarmos o custo de todos os níveis da recursão teríamos n níveis, cada um com um elemento a menos que o nível anterior de custo $\Theta(n)$ teríamos uma série aritmética $(cn + c(n-1) + c(n-2) + \cdots + c1)$, que é $\Theta(n^2)$. Desta forma, se o particionamento for totalmente desbalanceado em todo nível da recursão, o tempo de execução é $\Theta(n^2)$, que não é melhor que o `INSERTIONSORT`, por exemplo. Além disto, o pior caso ocorre quando o vetor de entrada já está ordenado, um caso que o `INSERTIONSORT` executa em tempo $\Theta(n)$.

Particionamento no Melhor Caso

No corte (particionamento) mais bem distribuído possível, dois subproblemas devem ser resolvidos recursivamente, um com $\lceil \frac{n}{2} \rceil$ elementos e outro com $\lceil \frac{n}{2} \rceil - 1$ elementos. Neste caso, o tempo necessário para `QUICKSORT` executar é dado pela recorrência

$$T(n) = 2T\left(\tfrac{n}{2}\right) + \Theta(n)$$

Ignorando o chão e teto e a subtração de 1, sem perda de generalidade. Pelo caso 2 do teorema mestre, a recorrência tem solução $T(n) = \Theta(n \lg n)$. Portanto, se o particionamento for balanceado em todos os níveis, temos um algoritmo mais eficiente, comparável ao `HEAPSORT`, com constantes escondidas potencialmente menores.

Particionamento no Caso Médio

Uma das vantagens práticas do `QUICKSORT` é que a complexidade do caso médio se assemelha mais ao melhor caso do que ao pior caso. Para entender melhor esta relação é necessário entender como o balanço do particionamento reflete na recorrência que descreve o tempo de execução.

Suponha que `PARTITION` sempre produza subproblemas em proporção 9 pra 1 em todos os níveis da recursão. Note que esta aproximação é grosseira, uma vez que em uma execução média todo tipo de corte seria realizado por `PARTITION`, nas mais variadas proporções. No entanto, como veremos adiante, esta aproximação nos permite ter uma

boa noção do tempo de execução no caso médio, sem nos ater aos cálculos mais complicados em uma contabilização mais detalhada da distribuição de probabilidade dos possíveis cortes realizados por `PARTITION`. Portanto, supondo a proporção 9 pra 1, a recorrência de `QUICKSORT` seria

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n)$$

Como exercício, construa a árvore de recursão para esta recorrência. Note que todo nível da árvore tem custo cn até o nível de altura $\log_1 0 = \Theta(\lg n)$. A partir deste nível, todos os demais possuem custo $O(cn)$, ou seja, no máximo cn. A recursão termina na altura $\log_{\frac{10}{9}} n = \Theta(\lg n)$. É possível mostrar, pelo método da substituição que $T(n) = \Theta(n \lg n)$.

Assim, com um balanceamento com corte em proporção 9 pra 1 em todo nível da recursão, que intuitivamente parece relativamente desbalanceado, `QUICKSORT` executa em $O(n \lg n)$, assintoticamente o mesmo que um corte perfeitamente balanceado em todas as chamadas recursivas. De fato, mesmo um corte em proporção 99 pra 1 em todas as chamadas levaria tempo $O(n \lg n)$. Aliás, qualquer corte de proporção constante leva a uma árvore de altura $\Theta(\lg n)$, tal que o custo de cada nível é $O(n)$. Por fim, pode-se concluir que, intuitivamente, desde que o corte seja em proporção constante, o tempo de execução de `QUICKSORT` é $O(n \lg n)$.

Intuição para o Caso Médio

Para descrever exatamente o comportamento aleatório de `QUICKSORT`, devemos considerar quão frequente encontramos várias entradas diferentes, bem como a relação entre as posições relativas de seus valores. Assim, o comportamento do `QUICKSORT` depende da ordenação prévia relativa dos elementos da entrada, não dos valores em si. Assumimos por enquanto que todas as permutações dos elementos da entrada ocorrem com a mesma probabilidade.

Quando `QUICKSORT` é executado em um vetor qualquer, o particionamento dificilmente ocorre da mesma forma em todos os nívels, como a análise anterior assumiu. Esperamos que alguns cortes sejam bem balanceados e outros não. No caso médio, `PARTITION` produz uma quantidade mista de cortes "bons" e "ruins". Em uma árvore de recursão para o caso médio de `QUICKSORT`, há cortes bons e ruins distribuídos pela árvore. Suponha, por questão de simplicidade de cálculos, que cortes bons e ruins estão em níveis alternados na árvore, como apresentado na Figura 3 a seguir.

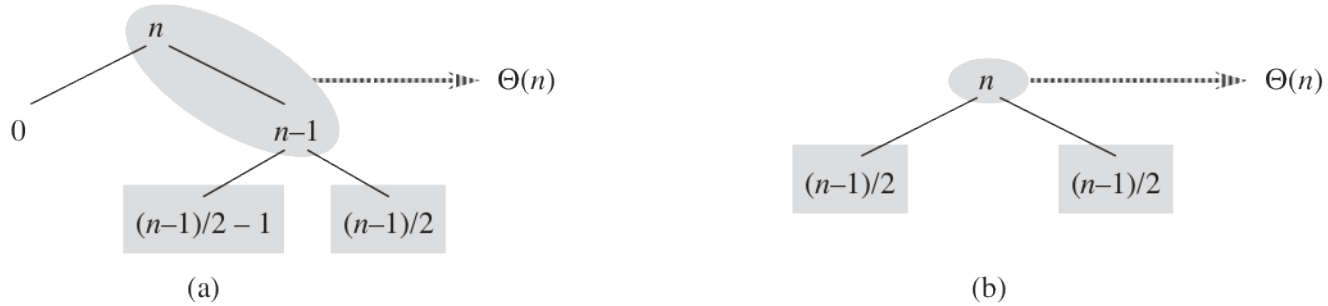

Figura 3: Dois casos da execução do Quicksort misturando um particionamento "bom"e outro "ruim". (CRLS, 2012)

A Figura 3(a) mostra dois cortes sucessivos em uma árvore de recursão. Na raíz, o custo para particionar é n, e os subvetores produzidos tem tamanho $n-1$ e 0: um corte "ruim". No próximo nível, o subvetor com $n-1$ elementos é particionado em subvetores de tamanho $\frac{n-1}{2} - 1$ e $\frac{n-1}{2}$: um corte "bom". Assumimos que o custo é constante para um vetor de tamanho 0. A combinação do corte "ruim" seguido por um corte "bom" produz três subproblemas de tamanhos 0, $\frac{n-1}{2} - 1$ e $\frac{n-1}{2}$ com custo de particionamento total de $\Theta(1) + \Theta(n) + \Theta(n-1) = \Theta(n)$, que não é pior que o particionamento mostrado na Figura 3(b). Neste caso, a Figura 3(b) possui apenas um nível de particionamento com corte "bom" e que produz dois subproblemas com $\frac{n-1}{2}$ problemas e custo de particionamento $\Theta(n)$, mesmo que, nesta situação, apenas um único corte "bom" foi realizado.

Intuitivamente, o custo do corte "ruim" é absorvido no custo $\Theta(n)$ do corte "bom", resultando em um corte bom. Assim, o custo do `QUICKSORT`, quando os níveis alternam entre cortes "bons" e "ruins" é parecido com o custo envolvido quando há apenas cortes "bons": $O(n \lg n)$, mas com uma constante maior, escondida na notação O.

Bibliografia

[CRLS] CORMEN, T. H. et al. Algoritmos: Teoria e Prática. Elsevier, 2012. 3a Ed. Capítulo 7 (*Quicksort*), Seções 7.1 e 7.2.