O Problema da Seleção e Limitante para o Problema de Ordenação

Pedro Henrique Del Bianco Hokama
3 de Setembro de 2019

Referências:

- Notas de aulas fortemente baseadas no curso: Stanford Algorithms by Tim Roughgarden
  - `https://www.youtube.com/playlist?list=PLXFMmlk03Dt7Q0xr1PIAriY5623cKiH7V`
  - Vídeos: 8.1 até 8.6
- CLRS: Cap 8 e 9

1 Problema da Seleção

No problema da seleção, recebemos um arranjo A com n números, e um inteiro $i \in \{1, 2, \ldots, n\}$, e precisamos encontrar a i-ésima estatística de ordem, ou seja, o i-ésimo menor número de A.

Uma motivação para o estudo desse problema é, por exemplo, encontrar a mediana que muitas vezes é representante melhor de um conjunto de dados do que a média, já que essa é mais suscetível a dados fora da curva. Para encontrar a mediana podemos usar $i = (n+1)/2$ se n é impar e $i = n/2$ para n par.

Exercício 1.1: Como resolver esse problema em $O(n \log n)$?

A ideia agora é desenvolver um algoritmo com tempo de execução esperado $O(n)$. Usando o algoritmo de partição do QuickSort.

- Suponha que procuramos o 5^o menor elemento.
- Suponha que escolhemos um pivô qualquer
- Aplicamos a partição e descobrimos que ele é o 3^o menor elemento
- Podemos então procurar o 2^o menor elemento da segunda parte do arranjo.

Algoritmo 1: SelecaoR

Entrada: Um arranjo A, comprimento n, inteiro i
Saída: A i-ésima estatística de ordem
início
  se $n = 1$ **então** devolva $A[i]$;
  $p =$ Escolhe pivô de A uniformemente aleatório;
  Particionar A em torno de p;
  Seja j o novo índice de p;
  se $j = i$ **então** devolva p;
  se $j > i$ **então** devolva SelecaoR(Primeira parte de A, $j-1$, i);
  se $j < i$ **então** devolva SelecaoR(Segunda parte de A, $n-j$, $i-j$);
fim

Exercício 1.2: Provar que SelecaoR está correto? Dica: Usar indução.

Exercício 1.3: Qual o tempo de execução de pior caso do SelecaoR?

Intuitivamente se conseguirmos dividir o arranjo da melhor maneira, ou seja, exatamente na metade, chegaríamos a uma recorrência como $T(n) \leq T(n/2) + O(n)$, que pelo teorema mestre é $O(n)$. Assim como no QuickSort um bom pivô era suficiente para chegar perto do desempenho da mediana, será que nesse algoritmo essa estratégia vai funcionar?

Teorema 1.1: Para qualquer entrada de comprimento n, o tempo de execução esperado de SelecaoR é $O(n)$.

- O algoritmo de partição usa $\leq cn$ operações para alguma constante $c > 0$
- Diremos que SelecaoR está na Fase j se o tamanho atual do arranjo está entre $\left(\frac{3}{4}\right)^{j+1} n$ e $\left(\frac{3}{4}\right)^{j} n$
- Seja X_j uma variável aleatória que indica o número de chamadas recursivas durante a Fase j

Com essa notação podemos calcular o tempo de execução total do SelecaoR é

$$\leq \sum_{j=0}^{\#\text{ Fases}} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n$$

Note que selecionar um pivô que reduza em 3/4 o tamanho do arranjo, ou seja, que faça o algoritmo mudar de Fase é de 50%. Então quantas vezes teremos que fazer a chamada recursiva até obter selecionar pivô bom?

Podemos raciocinar um evento de jogar moedas. Seja N (uma V.A.) o número de lançamentos de moedas até obter uma cara. Note então que $E[X_j] \leq E[N]$

$$\begin{aligned}
E[N] &= 1 + \frac{1}{2}E[N] \\
E[N] &= 1 + \frac{1}{2}(1 + \frac{1}{2}E[N]) \\
E[N] &= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots \\
E[N] &= 2
\end{aligned}$$

Voltamos então para o tempo de execução esperado do SelecaoR

$$
\begin{aligned}
&\leq E\left[\sum_{j=0}^{\#\text{ Fases}} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right] \\
&= E\left[cn \sum_{j=0}^{\#\text{ Fases}} X_j \left(\frac{3}{4}\right)^j\right] \\
&= cn \sum_{j=0}^{\#\text{ Fases}} \left(\frac{3}{4}\right)^j E\left[X_j\right] \\
&\leq cn \sum_{j=0}^{\#\text{ Fases}} \left(\frac{3}{4}\right)^j E[N] \\
&= cn \sum_{j=0}^{\#\text{ Fases}} \left(\frac{3}{4}\right)^j 2 \\
&= 2cn \sum_{j=0}^{\#\text{ Fases}} \left(\frac{3}{4}\right)^j \\
&\leq 2cn \frac{1}{1-\frac{3}{4}} \\
&= 2cn4 \\
&= 8cn
\end{aligned}
$$

E portanto o tempo de execução esperado de SelecaoR para qualquer entrada de tamanho n é $O(n)$.

2 Um limitante inferior para o Problema de Ordenação

Algoritmos de Ordenação como o InsertionSort, BubbleSort, SelectionSort, MergeSort, QuickSort e HeapSort baseiam seu funcionamento em comparação entre seus elementos. É o método usado quando não podemos assumir nenhuma propriedade sobre os elementos da entrada. São algoritmos de propósitos gerais pois funcionam para qualquer tipo de entrada.

> **Teorema 2.1:** Todo algoritmo de ordenação baseado em comparações tem um tempo de execução de pior caso $\Omega(n \log n)$.

- Agora suponha uma entrada qualquer de comprimento n e um algoritmo baseado em comparações que ordena corretamente esse arranjo.
- Seja K o número de comparações feitas pelo algoritmo
- Para cada comparação o algoritmo tem um resultado digamos 0 ou 1
- Portanto o número total de resultados possíveis é 2^K
- O número possível de permutações do arranjo de entrada é $n!$

Se $2^K < n!$ pelo principio da casa de pombos duas entradas idênticas irão obter os mesmos resultados, o que é um absurdo já que o algoritmo ordena corretamente.

Então:

$$
\begin{aligned}
2^K &\geq n! \\
&= n(n-1)(n-2)\dots(n/2)\dots 1 \\
&\geq \underbrace{n(n-1)(n-2)\dots(n/2)}_{n/2 \text{ termos}} \\
&\geq \underbrace{(n/2)(n/2)(n/2)\dots(n/2)}_{n/2 \text{ termos}} \\
&= (n/2)^{(n/2)}
\end{aligned}
$$

Ou seja:

$$
\begin{aligned}
2^K &\geq (n/2)^{(n/2)} \\
\log 2^K &\geq \log(n/2)^{(n/2)} \\
K \log 2 &\geq (n/2)\log(n/2) \\
K &\geq (n/2)\log(n/2)
\end{aligned}
$$

Portanto K é $\Omega(n \log n)$.