Algoritmos

Pedro Hokama

Fontes

- [*clrs*] Algoritmos: Teoria e Prática (Terceira Edição) Thomas H. Cormen, Charles Eric Leiserson, Ronald Rivest e Clifford Stein.
- [*timr*] Algorithms Illuminated Series, Tim Roughgarden
- Desmistificando Algoritmos, Thomas H. Cormen.
- Algoritmos, Sanjoy Dasgupta, Christos Papadimitriou e Umesh Vazirani
- Stanford Algorithms
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt7Q0xr1PIAriY5623cKiH7V`
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt5EMI2s2WQBsLsZl7A5HEK6`
- Conjunto de Slides dos Professores Cid C. de Souza, Cândida N. da Silva, Orlando Lee, Pedro J. de Rezende
- Conjunto de Slides do Professores Cid C. de Souza para a disciplina MO420

Qualquer erro é de minha responsabilidade.

Problema da Seleção

Problema da Seleção

Dado um arranjo A com n números, e um inteiro $i \in \{1, 2, \ldots, n\}$, encontrar a i-ésima estatística de ordem, ou seja, o i-ésimo menor número de A.

- Comumente desejamos encontrar a mediana de um conjunto.
- A mediana normalmente é menos afetada por corrupção em um conjunto de dados.
- Como resolver esse problema em $O(n \log n)$?
  - Ordenar A e devolver $A[i]$.
  - Isso é uma chamada "redução", reduzir uma instância de um Problema A para uma instância do Problema B, usar um algoritmo para resolver a instância do Problema B, e usar o resultado para responder o problema A. Esse conceito será visto com detalhes no futuro.

Problema da Seleção

- Será que podemos fazer melhor? 🤔
- Certamente não vamos encontrar nada melhor que linear, pois pelo menos precisamos ler todos os valores uma vez. Mas será que conseguimos algo mais próximo de $O(n)$ do que $O(n \log n)$?

Problema da Seleção

- Queremos encontrar um algoritmo de tempo esperado linear 🤞
    - Ideia: usar o algoritmo Partição do QuickSort.

$$(\textcolor{red}{5}, 3, 8, 9, 10, 7, 1, 2)$$

partição

$$(\underbrace{2, 3, 1}_{\leq 5}, 5, \underbrace{10, 7, 8, 9}_{\geq 5})$$

- Suponha que procuramos o 6º menor elemento.
- Suponha que escolhemos um pivô qualquer
- Aplicamos a partição e descobrimos que ele é o 4º menor elemento
- Podemos então procurar o 2º menor elemento da segunda parte do arranjo.

Algoritmo 1: Partição

Entrada: Um arranjo A, índices l e r

Saída: O mesmo arranjo mas particionado

Coloca o pivot em A[l] ;
$pivot = A[l]$;
$i = l + 1$;
para $j = l + 1$ *até* r **faça**
  se $A[j] < pivot$ **então**
    Troca $A[j]$ e $A[i]$;
    $i = i + 1$;
Troca $A[l]$ e $A[i-1]$;
devolva $i - 1$;

Algoritmo 2: SelecaoR

Entrada: Um arranjo A, índices l e r, inteiro i

Saída: A i-ésima estatística de ordem

se $l == r$ **então** devolva $A[i]$;
$p =$ Partição(A, l, r);
$j = p - l + 1$ //p é o j-ésimo menor valor do array atual;
se $j = i$ **então** devolva A[p];
se $j > i$ **então** devolva SelecaoR(A, l, $p-1$, i);
se $j < i$ **então** devolva SelecaoR(A, $p+1$, r, $i-j$);

SeleçãoR - Análise

- Corretude = Análogo ao QuickSort.
- Complexidade:
    - Antes vamos pensar:
    - Qual o tempo de execução de pior caso? Ou seja, se na partição sempre dividirmos da forma mais desbalanceada possível. $O(n^2)$.
    - Qual o tempo de execução se na partição sempre dividirmos da forma mais balanceada possível.

$$T(n) \leq T(n/2) + O(n)$$

    - Teorema mestre: como $a < b^d$ caímos no caso 2. E portanto $T(n) = O(n)$.
- Assim como no QuickSort um pivô aleatório era suficiente para chegar perto do desempenho da mediana, será que nesse algoritmo essa estratégia vai funcionar?

SeleçãoR - Complexidade

Teorema

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

- Para facilitar a análise, vamos dividir a execução do SeleçãoR em **Fases**.
- Diremos que SelecaoR está na Fase j se o tamanho atual do arranjo está entre $\left(\frac{3}{4}\right)^{j+1} n$ e $\left(\frac{3}{4}\right)^{j} n$
- Por exemplo, se $n = 1000$,

Fase	Tamanho
0	(750, 1000]
1	(562, 750]
2	(421, 562]
3	(316, 421]
...	...

- O algoritmo Partição realiza $\leq cn$ operações para alguma constante $c > 0$.
- Seja X_j uma variável aleatória do número de chamadas recursivas durante a Fase j

$$\text{Tempo de execução do SeleçãoR} \leq \sum_{\text{Fases } j} X_j \cdot \underbrace{c \cdot \overbrace{\left(\frac{3}{4}\right)^j n}^{\text{tamanho máximo do vetor durante Fase j}}}_{\text{operações por subproblema na Fase j}}$$

$$\boxed{\text{Tempo de execução do SeleçãoR} \leq \sum_{\text{Fases } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n}$$

- Se SeleçãoR escolher um pivô que faz uma divisão de $25 - 75\%$ ou melhor,

$< p$	p	$> p$

- então a Fase atual termina!
- Já que por definição o subproblema (mesmo que for a maior porção) é no máximo 75%
- Lembre que metade dos elementos escolhidos vão dar uma divisão nessa proporção. E portanto a probabilidade de migrar para a próxima fase é de 50%
- Pense então que essa é a mesma probabilidade de jogar uma moeda e obter "cara".
- $E[X_j] \leq$ número esperado de vezes que você precisa jogar uma moeda para obter "cara".

- $E[X_j] \leq E[N]$, N = número esperado de vezes que você precisa jogar uma moeda para obter "cara".

$$\begin{aligned}
E[N] &= 1 + \frac{1}{2}E[N] \\
E[N] - \frac{1}{2}E[N] &= 1 + \frac{1}{2}E[N] - \frac{1}{2}E[N] \\
\frac{1}{2}E[N] &= 1 \\
\frac{1}{2}E[N] \cdot 2 &= 1 \cdot 2 \\
E[N] &= 2
\end{aligned}$$

$$\boxed{\text{Tempo de execução do SeleçãoR} \leq \sum_{\text{Fases } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n}$$

$$E\left[\sum_{\text{Fases } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right] = cn \cdot E\left[\sum_{\text{Fases } j} X_j \cdot \left(\frac{3}{4}\right)^j\right]$$

$$(L.E.) = cn \cdot \sum_{\text{Fases } j} E[X_j] \left(\frac{3}{4}\right)^j \leq 2cn \cdot \sum_{\text{Fases } j} \left(\frac{3}{4}\right)^j$$

$$= 2cn \cdot \overbrace{\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \left(\frac{3}{4}\right)^3 + \dots\right)}^{\text{P.G. de razão 3/4}}$$

$$= 2cn \cdot \left(\frac{1}{1 - 3/4}\right) = 2cn \cdot 4 = 8cn$$

$$E[\text{Tempo de execução do SeleçãoR}] = O(n) \quad \square$$

Um limitante inferior para Algoritmos 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.
- Ou quando não podemos acessar diretamente os elementos mas apenas compará-los através de uma função.
- São algoritmos de propósitos gerais pois funcionam para qualquer tipo de entrada.
- Existem algoritmos de ordenação que se baseiam em outro tipo de interação com os dados. BucketSort, CountingSort, RadixSort.
- Mas esses algoritmos partem de alguma premissa sobre os dados. Ex: São inteiros com N dígitos, são valores de ponto flutuante uniformemente distribuídos entre 0 e 1;

Teorema

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

- 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 dos pombos duas entradas idênticas irão obter os mesmos resultados, o que é um absurdo já que o algoritmo ordena corretamente.
Portanto:

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

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

Portanto K é $\Omega(n \log n)$. □

- Dessa forma nenhum algoritmo baseado em comparações pode ser melhor que $O(n \log n)$.
- Podemos afirmar que o problema da seleção é mais fácil que o problema da ordenação.