Algoritmos

Pedro Hokama

Fontes

- [*clrs*] Algoritmos: Teoria e Prática (Terceira Edição) Thomas H. Cormen, Charles Eric Leiserson, Ronald Rivest, Ronald L. Rivest e Clifford Stein.
- [*timr*] Algorithms Illuminated Series, Tim Roughgarden

Apresentação Baseada:

- 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.

Aleatorização do QuickSort

- A aleatorização de algoritmos é uma ferramenta importante da bagagem de qualquer profissional da computação.
- Veremos como aplicar essa técnica no QuickSort e as vantagens que ela pode trazer.
- A ideia é escolher é escolher cada pivô aleatoriamente com a **mesma probabilidade**.

Análise - QuickSort

Teorema

Para qualquer entrada de comprimento n, o tempo médio de execução do QuickSort (com pivôs aleatórios) é $O(n \log n)$.

- Primeiramente considere como entrada um arranjo A de comprimento n.
- Espaço amostral: $\Omega =$ todos as possíveis sequências de escolhas de pivôs no QuickSort.
- Variável aleatória: para qualquer $\sigma \in \Omega$, $C(\sigma) =$ número de comparações entre dois elementos do arranjo feito pelo algoritmo QuickSort dado as escolhas σ. Note que o tempo de execução total do QuickSort é dominado por esse número.
- O objetivo então é mostrar que $E[C] = O(n \log n)$

- Notação:
  - z_i = i-ésimo menor elemento
  - ou seja, z_i não é o elemento que está originalmente na i-ésima posição do vetor, mas sim o elemento que vai ocupar a i-ésima posição no vetor quando ordenado.

$$(\underbrace{6}_{z_2}, \underbrace{10}_{z_4}, \underbrace{8}_{z_3}, \underbrace{2}_{z_1})$$

  - para uma escolha σ, e $i < j$, seja $X_{ij}(\sigma)$ = número de vezes que z_i e z_j são comparados.
- Fixado dois elementos da entrada, quantas vezes eles podem ser comparados?
  a. 1
  b. 0 ou 1
  c. 0, 1 ou 2
  d. algo entre 0 e $n-1$

Algoritmo 1: QuickSort

Entrada: Um arranjo A, índices l e r
Saída: Um arranjo com os mesmos elementos de A porém ordenados

```
1 se r − l ≤ 0 então devolva A;
2 p = Partição(A, l, r);
3 QuickSort(A, l, p-1);
4 QuickSort(A, p+1, r);
5 devolva A;
```

Algoritmo 2: Partição

Entrada: Um arranjo A, índices l e r
Saída: O mesmo arranjo mas particionado

```
1 Coloca o pivot em A[l] ;
2 pivot = A[l] ;
3 i = l + 1;
4 para j = l + 1 até r faça
5     se A[j] < pivot então
6         Troca A[j] e A[i];
7         i = i + 1;
8 Troca A[l] e A[i − 1];
9 devolva i − 1;
```


- Podemos então facilmente escrever C em função de X

$$C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}(\sigma), \qquad \forall \sigma \in \Omega$$

- Pela linearidade da esperança (lembrando que ela se aplica mesmo se as variáveis não forem independentes)

$$E[C] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}].$$

- Como:

$$\begin{aligned} E[X_{ij}] &= 0 \cdot Pr[X_{ij} = 0] + 1 \cdot Pr[X_{ij} = 1] \\ &= \cancel{0 \cdot Pr[X_{ij} = 0]} + 1 \cdot Pr[X_{ij} = 1] \\ &= Pr[X_{ij} = 1] = Pr[z_i \text{ e } z_j \text{ serem comparados}] \end{aligned}$$

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ e } z_j \text{ serem comparados}]$$

Lema

Para todo $i < j$, $Pr[z_i \text{ e } z_j \text{ serem comparados}] = \frac{2}{(j-i+1)}$

- Fixe z_i, z_j com $i < j$.
- Considere o conjunto $\{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$.
- Desde que nenhum desses números seja escolhido como pivô, todos serão passados juntos para a mesma chamada recursiva.
- Considere então o primeiro entres $\{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$ que é escolhido como pivô:
  - se z_i ou z_j for escolhido primeiro, eles serão escolhidos.
  - se escolher um dos outros então z_i e z_j nunca serão comparados.
- como a escolha é aleatória qualquer um dos elementos do conjunto tem a mesma chance de ser escolhido primeiro (do conjunto) e portando a chance deles serem comparados é

$$\frac{2}{(j-i+1)}$$

□

Então

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ e } z_j \text{ serem comparados}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}$$

$$E[C] = 2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{(j-i+1)}$$

.

Agora note que a soma interna

$$\sum_{j=i+1}^{n} \frac{1}{(j-i+1)} = \sum_{j=2}^{n} \frac{1}{j} \leq \ln n$$

$$E[C] \leq 2 \cdot n \cdot \ln n$$

e portanto:

O tempo de execução esperado do QuickSort é $O(n \log n)$ □

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.
- 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.

$$(5, 3, 8, 9, 10, 7, 1, 2)$$

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 3: Partição

Entrada: Um arranjo A, índices l e r
Saída: O mesmo arranjo mas particionado

```
1 Coloca o pivot em A[l] ;
2 pivot = A[l] ;
3 i = l + 1;
4 para j = l + 1 até r faça
5     se A[j] < pivot então
6         Troca A[j] e A[i];
7         i = i + 1;
8 Troca A[l] e A[i − 1];
9 devolva i − 1;
```

Algoritmo 4: SelecaoR

Entrada: Um arranjo A, índices l e r, inteiro i
Saída: A i-ésima estatística de ordem

```
1 se l == r então devolva A[i];
2 p = Partição(A, l, r);
3 j = p − l + 1 //p é o j-ésimo menor valor do array atual;
4 se j = i então devolva A[p];
5 se j > i então devolva SelecaoR(A, l, p − 1, i);
6 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

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

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

- 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".

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

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

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

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

$$
= 2cn \cdot \overbrace{\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \left(\frac{3}{4}\right)^3 + \ldots\right)}^{\text{P.G. de razão } 3/4} = 2cn \cdot \left(\frac{1}{1 - 3/4}\right) = 2cn \cdot 4 = 8cn
$$

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

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.