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.

Notação Ω e Θ

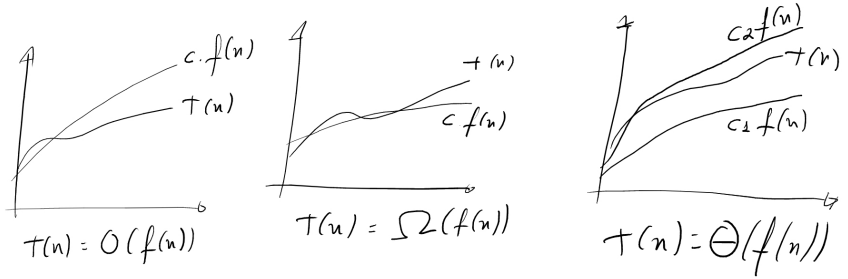
Analogia

- Intuitivamente, você pode comparar a notação O com uma relação de $\leq$. Já que dizer que $T(n)$ ser $O(f(n))$, quer dizer que: (existe constante c e n_0 tal que para todo n maior que n_0)

$$T(n) \leq cf(n)$$

- Nessa comparação a notação Ω seria como uma relação de $\geq$
- E a notação Θ seria como uma relação de $=$

Notação Ω e Θ



Notação Ω

A notação Ω que define um limitante inferior. Formalmente:

Definição

$T(n) = \Omega(f(n))$ se e somente se exitem constantes c, $n_0 > 0$ tais que

$$T(n) \geq c.f(n)$$

para todo $n \geq n_0$.

Notação Θ

A notação Θ indica que uma função $T(n)$ é limitada inferiormente e superiormente por uma função $f(n)$. Formalmente:

Definição

$T(n) = \Theta(f(n))$ se e somente se exitem constantes c_1, c_2 e $n_0 > 0$ tais que

$$c_1.f(n) \leq T(n) \leq c_2.f(n)$$

para todo $n \geq n_0$.

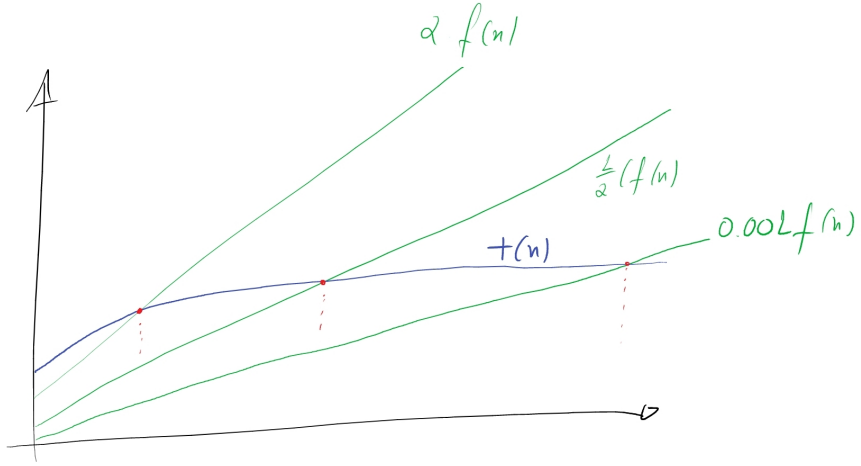
Definição

$T(n) = \Theta(f(n))$ se e somente se $T(n) = O(f(n))$ e também se $T(n) = \Omega(f(n))$.

o e ω

- A notação o (Little-Oh, ózinho, ó pequeno) é semelhante a notação O porém não assintoticamente justo.
- Informalmente pode-se pensar que se O está para uma relação do tipo $\leq$, enquanto o o é uma relação do tipo $<$.
- A notação ω (Little-omega, omeguinha, omega pequeno) é semelhante a notação Ω porém não assintoticamente justo.
- Informalmente pode-se pensar que Ω está para uma relação do tipo $\geq$, enquanto a ω é uma relação do tipo $>$.

o e ω



Notação o

Definição

$T(n) = o(f(n))$ se e somente se para toda constante $c > 0$, existe um $n_0 > 0$ tal que

$$T(n) < c.f(n)$$

para todo $n \geq n_0$.

Exercício: Provar que para todo $k \geq 1$, $n^{k-1} = o(n^k)$.

Notação ω

Definição

$T(n) = \omega(f(n))$ se e somente se para toda constante $c > 0$, existe um $n_0 > 0$ tal que

$$T(n) > c.f(n)$$

para todo $n \geq n_0$.

Mais exemplos de Notação Assintótica

Exemplo

Provar que $2^{n+10} = O(2^n)$.

Precisamos exibir c e $n_0 > 0$ tais que

$$2^{n+10} \leq c2^n \text{ para todo } n \geq n_0.$$

Como escolher c e n_0?

$$2^{n+10} \leq c2^n$$
$$2^{10}2^n \leq c2^n$$
$$1024 \cdot 2^n \leq c2^n$$

Quando isso é verdade?
Escolhemos então algum $c \geq 1024$, e $n_0 \geq 1$.

Exemplo

Provar que $2^{n+10} = O(2^n)$.

Precisamos exibir c e $n_0 > 0$ tais que

$$2^{n+10} \leq c2^n \text{ para todo } n \geq n_0.$$

Escolhemos então $c = 1024$, e $n_0 = 1$ e vamos mostrar que:

$$2^{n+10} \leq 1024 \cdot 2^n \text{ para todo } n \geq 1$$
$$2^{10}2^n \leq 1024 \cdot 2^n \text{ para todo } n \geq 1$$
$$1024 \cdot 2^n \leq 1024 \cdot 2^n \text{ para todo } n \geq 1$$

logo $2^{n+10} = O(2^n)$. □

Exemplo

Provar que 2^{10n} não é $O(2^n)$.

Suponha por absurdo (por contradição) que $2^{10n} = O(2^n)$ e portanto $2^{10n} \leq c2^n$ para alguma constante c e $n \geq n_0$. Então

$$2^{10n} \leq c2^n$$
$$2^{9n}2^n \leq c2^n$$
$$2^{9n}\cancel{2^n} \leq c\cancel{2^n}$$
$$2^{9n} \leq c$$

obviamente não existe uma constante que seja maior que 2^{9n} para todos os naturais n, portanto chegamos a uma contradição. Logo 2^{10n} não pode ser $O(2^n)$. □

Exemplo

Para quaisquer dois pares de funções positivas, $f(n)$ e $g(n)$, provar que $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

Para mostrar que a afirmação é verdadeira, podemos escolher $c_1 = 1/2$, $c_2 = 1$ e $n_0 = 1$ e verificamos que:

$$\frac{1}{2}(f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq f(n) + g(n) \quad \forall n \geq n_0$$

Usando limites

$T(n) \in o(f(n))$ se $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$.

$T(n) \in \omega(f(n))$ se $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$.

$T(n) \in O(f(n))$ se $\lim_{n\to\infty} \frac{T(n)}{f(n)} < \infty$.

$T(n) \in \Omega(f(n))$ se $\lim_{n\to\infty} \frac{T(n)}{f(n)} > 0$.

$T(n) \in \Theta(f(n))$ se $0 < \lim_{n\to\infty} \frac{T(n)}{f(n)} < \infty$.

Exemplo

$T(n) = \ln n$ e $f(n) = n^e$.

$$\lim_{n\to\infty} \frac{\ln n}{n^e} = \lim_{n\to\infty} \frac{1/n}{e.n^{e-1}} = 0.$$

portanto $\ln n = o(n^e)$. Obviamente $\ln n = O(n^e)$ também.

Divisão e Conquista: O paradigma

O paradigma de divisão e conquista tem três etapas:

1. **Dividir** o problema em subproblemas menores.
2. **Conquistar** os subproblemas (normalmente de forma recursiva).
3. **Combinar** a solução dos subproblema para encontrar uma solução para o problema original.

Diferentes algoritmos tem a complexidade diferente em cada uma das fases, por exemplo, no MergeSort a divisão é trivial mas a combinação exige esforço. Já no QuickSort a divisão é bastante elaborada mas a combinação é trivial.

O Problema

- Suponha que você e um amigo escolheram 10 filmes que ambos assistiram.
- Cada um ordenou esses 10 filmes em ordem de preferência.
- Queremos saber a compatibilidade entre essas duas listas, e verificar se essa amizade pode dar certo.
- Um serviço de streaming poderia usar essa comparação para verificar usuários que tem gostos parecidos para fazer recomendações.

Problema do Número de Inversões

Problema do Número de Inversões

Dado um arranjo A contendo n inteiros em uma ordem arbitrária, encontrar o número total de inversões, ou seja, o número de pares (i, j) de índices $1 < i, j < n$ tais que $i < j$ e $A[i] > A[j]$.

Considere o vetor seguinte vetor

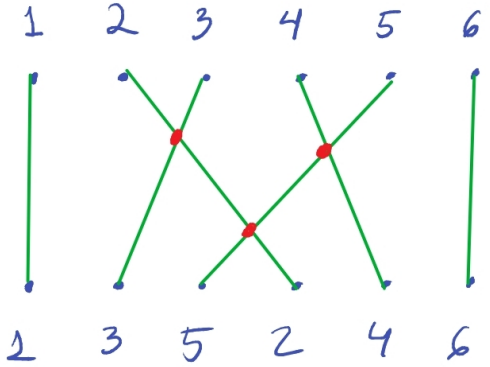
$$A = (1, 3, 5, 2, 4, 6)$$

qual o número de inversões?

- os elementos 3 e 2, então os índices $(2, 4)$ formam uma inversão
- os elementos 5 e 2, então os índices $(3, 4)$ formam uma inversão
- os elementos 5 e 4, então os índices $(3, 5)$ formam uma inversão

Problema do Número de Inversões

$$A = (1, 3, 5, 2, 4, 6)$$



Problema do Número de Inversões

Qual o número máximo de inversões em um vetor de tamanho 6?

a. 6
b. 15
c. 21
d. 36
e. 64

Problema do Número de Inversões

Qual o número máximo de inversões em um vetor de tamanho n?

- O máximo de inversões acontece se todos os pares de índice estiverem invertidos.
- Ou seja, dos n elementos, quaisquer dois que escolhermos estará invertido.

$$\binom{n}{2}$$

- Relembrando:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Então

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n.(n-1).(n-2)!}{2!(n-2)!} = \frac{n^2-n}{2}$$

Uma ideia ingenua

Como seria uma solução força bruta?

Algoritmo 1: ContaInversões

Entrada: Um vetor A de tamanho n
Saída: O número de inversões

```
1 t = 0;
2 para i de 1 até n − 1 faça
3     para j de i + 1 até n faça
4         se A[i] > A[j] então
5             t++ ;
6 devolva t;
```

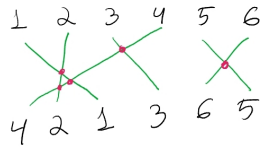
Qual a complexidade desse algoritmo?

a. $\log n$
b. n
c. $n \log n$
d. n^2
e. n^3

Podemos fazer melhor?
Yep!

Uma algoritmo de divisão e conquista

$(4, 2, 1, 3, 6, 5)$



Valor verdadeiro: 5 inversões

Divisão
(4,2,1) (3,6,5)
Conquista
3 inversões 1 inversão
Combinação?
Como contar as inversões entre as metades?

Uma algoritmo de divisão e conquista

- Ideia: dividir o vetor em 2 metades, contar o número de inversões.
- Mas ainda faltará as inversões entre os elementos da primeira e da segunda metade.
- Podemos então contar essas inversões?
- Para isso vamos então classificar as inversões em três tipos:
  - **Esquerda:** se $i, j \leq n/2$
  - **Direita:** se $i, j > n/2$
  - **Split:** se $i \leq n/2 < j$
- Nova ideia: contar as inversões esquerda, direita e split.

Uma algoritmo de divisão e conquista

Uma ideia (ainda incompleta) seria:

Algoritmo 2: Count

Entrada: Um arranjo A de comprimento n
Saída: O número de inversões

```
1 se n ≤ 1 então devolva 0;
2 senão
3     x = Count(Primeira metade de A, n/2);
4     y = Count(Segunda metade de A, n/2);
5     z = ContaSplit(A,n);
6 devolva x + y + z;
```

- Se conseguirmos contar o número de inversões split em tempo linear $O(n)$ a árvore de recursão fica idêntica ao MergeSort.
- A complexidade total ficaria $O(n \log n)$
- O número de inversões de tipo split é $O(n^2)$. Será que podemos contar um número quadrático de coisas em tempo linear? 🤔
- Yep!

- Considere que Count conta o número de inversões, mas também ordena o vetor.
- E vamos dar uma olhada na função Merge do MergeSort

Algoritmo 3: Merge

Entrada: B e C arranjos ordenados com $m/2$
Saída: Arranjo D de tamanho m com os mesmos elementos de B e C mas ordenados

```
1 i = 1; j = 1;
2 para k de 1 até m faça
3     se B[i] < C[j] então
4         D[k] = B[i];
5         i++;
6     senão
7         D[k] = C[j];
8         j++;
9 devolva D;
```


- Se não houver inversões do tipo split, o que vai acontecer na hora de copiar B e C?
- Nesse caso B seria copiado inteiramente antes de C.
- O que acontece significa, em número de inversões, quando copiamos um elemento do vetor C?
- Isso significa que o elemento $C[j]$ copiado está em inversão do tipo split com todos os valores que ainda não foram copiados de B.
- Isso significa $|B| - i + 1$ elementos.

Modificando o Merge para Contar Inversões Splits

Algoritmo 4: Merge

Entrada: B e C arranjos ordenados com $m/2$
Saída: Arranjo D de tamanho m com os mesmos elementos de B e C mas ordenados

```
1 i = 1; j = 1;
2 para k de 1 até m faça
3     se B[i] < C[j] então
4         D[k] = B[i];
5         i++;
6     senão
7         D[k] = C[j];
8         j++;
9 devolva D;
```

Algoritmo 5: MergeCountSplit

Entrada: B e C arranjos ordenados com $m/2$
Saída: Arranjo D de tamanho m com os elementos de B e C mas ordenados, e o número de inversões Splits

```
1 i = 1; j = 1; t = 0;
2 para k de 1 até m faça
3     se B[i] < C[j] então
4         D[k] = B[i];
5         i++;
6     senão
7         D[k] = C[j];
8         j++; t = t + (m/2 − i + 1);
9 devolva (D, t);
```


Uma algoritmo de divisão e conquista (versão completa)

Algoritmo 6: SortCount

Entrada: Um arranjo A, o comprimento do arranjo n
Saída: Um arranjo com os mesmos elementos de A porém ordenados, O número de inversões

```
1 se n ≤ 1 então devolva (A, 0);
2 senão
3     (B, x) = SortCount(Primeira metade de A, n/2);
4     (C, y) = SortCount(Segunda metade de A, n/2);
5     (D, z) = MergeCountSplit(B, C, n);
6 devolva (D, x + y + z);
```


- Assim como no MergeSort, a árvore de recursão de SortCount tem $\log n$ níveis e cada nível executa $O(n)$ operações, então a complexidade total de SortCount é

$$O(n \log n)$$

- Portanto muito melhor que a versão força bruta!