Revisão de Complexidade de Tempo

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Complexidade de tempo

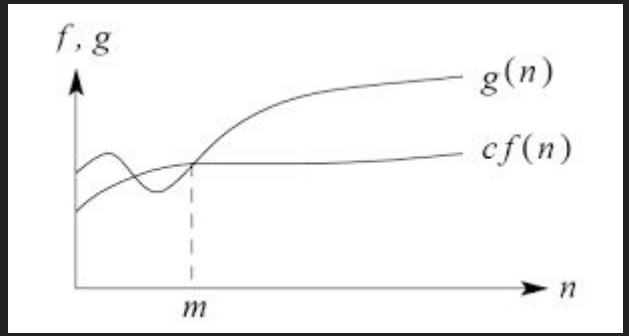
- Podemos medir a complexidade de tempo de execução de algoritmos para descobrir se é possível resolver tal problema computacionalmente com determinada solução
- Para isso, geralmente temos 3 possibilidades
  - Pior caso
  - Caso médio - é necessário uso de probabilidade
  - Melhor caso

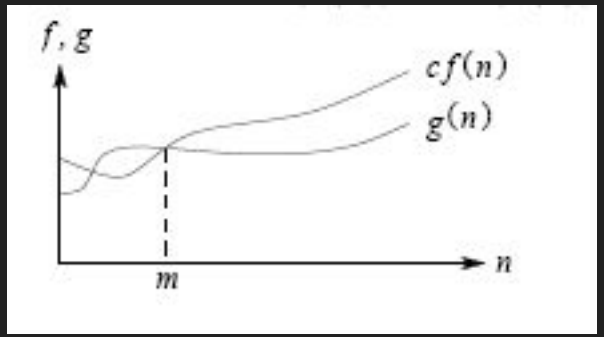
Complexidade de tempo

- Não importa qual possibilidade, consideramos o número de entradas *n*
  - O tempo para resolver o problema aumenta com o tamanho de *n*
- Para *n* pequeno, qualquer algoritmo custa pouco
- Por isso, analisamos os casos de *n* grande
  - O limite do comportamento de custo quando *n* cresce
- O comportamento assintótico da função de custo *f(n)*

Complexidade de tempo

- Dizemos que uma função *f(n)* domina assintoticamente outra função *g(n)* se existem duas constantes positivas *c* e *m* tais que, para *n* >= *m* temos
  - $|g(n)| <= c*|f(n)|$
- Ou seja, a partir de um número de entrada *m* a função *c*f(n)* é sempre maior que *g(n)* não importa o valor do número de entradas
- Notação: *g(n) = O(f(n))*





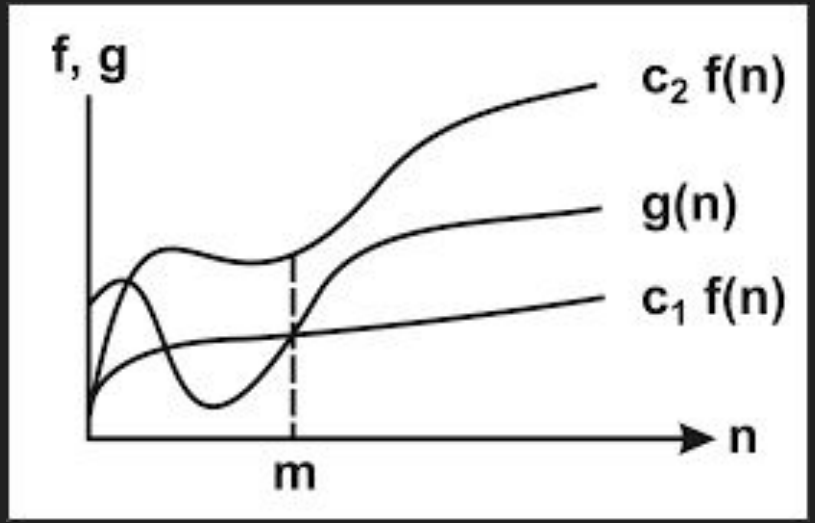
Fonte: [1]

Complexidade de tempo

- O caso contrário, de *g(n)* >= *c*f(n)* para todo n >= m recebe a notação *g(n)* é $\Omega(f(n))$
- Ou seja, a partir de um número de entrada *m* a função *c*f(n)* é sempre MENOR que *g(n)* não importa o valor do número de entradas
- Encontramos um limite inferior!

Complexidade de tempo

- Voltando ao caso do *O(f(n))*
- Caso exista uma outra constante c_2 na qual *c*f(n)* <= *g(n)* <= c_2**f(n)* para todo n >= m
- Ou seja, a partir de um valor de entrada *m*, a função *g(n)* é igual a *f(n)* a menos de uma constante
- *f(n)* é um limite assintótico firme
- *g(n)* = **Θ***(f(n))*



Fonte: [1]

Complexidade de tempo

- ➔ f(n) = O(1) -> Algoritmo de Complexidade Constante
- ➔ f(n) = O(logn) -> Algoritmo de Complexidade Logarítmica
- ➔ f(n) = O(n) -> Algoritmo de Complexidade Linear
- ➔ f(n) = O(n*logn) -> Típico de algoritmos que usam divisão e conquista
- ➔ $f(n) = O(n^2)$ -> Algoritmo de Complexidade Quadrática
- ➔ $f(n) = O(n^3)$ -> Algoritmo de Complexidade Cúbica
- ➔ $f(n) = O(2^n)$ -> Algoritmo de Complexidade Exponencial
- ➔ f(n) = O(n!) -> Algoritmo de Complexidade Fatorial

Função de Custo	Tamanho de *n*					
	10	20	30	40	50	60
n	0,00001s	0,00002s	0,00003s	0,00004s	0,00005s	0,00006s
n^2	0,0001s	0,0004s	0,0009s	0,0016s	0,0025s	0,0036s
n^3	0,001s	0,008s	0,027s	0,064s	0,125s	0,316s
n^5	0,1s	3,2s	24,3s	1,7 min	5,2 min	13 min
2^n	0,001s	1s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Adaptado de: [1]

Complexidade de tempo

- Para contar operações em geral
  - Comandos de atribuição, leitura ou escrita são O(1)
    - Avaliar condição também O(1)
  - For
    - Somar o tempo do corpo mais a condição de terminação multiplicado pelo número de iterações

Complexidade de tempo

- ➔ Para contar operações em procedimentos não recursivos
  - Cada procedimento deve ser contado separadamente
  - Iniciar naqueles que não chamam outros
  - Depois os que chamam os já avaliados
  - Até acabar

Complexidade de tempo

- ➔ Para contar operações em procedimentos recursivos
  - ◆ associa uma função de complexidade *f(n)* desconhecida
    - *n* é o tamanho dos argumentos
  - ◆ Encontrar uma equação de recorrência para *f(n)*
    - Maneira de definir uma função por uma expressão envolvendo essa mesma função

Complexidade de tempo

- Equação de recorrência:
  - Seja *T(n)* função de complexidade de um algoritmo
  - Vamos supor que o algoritmo execute 5 linhas de tempo O(1) e uma chamada recursiva *n* vezes
    - Ex: percorrer um vetor recursivamente
  - *T(n)* é especificado em função dos termos anteriores
    - *T(1), T(2), ..., T(n-1)*

Complexidade de tempo

- Equação de recorrência:
  - Seja *T(n)* função de complexidade de um algoritmo
  - Vamos supor que para cada chamada o algoritmo execute 5 linhas de tempo O(1) e uma chamada recursiva *n/2* vezes
  - *T(n)* é especificado em função dos termos anteriores
    - *T(1), T(2), ..., T(n-1)*
  - *T(n) = 5*n + T(n/2), T(1) = 5* (é preciso do valor de *n=1*)

Complexidade de tempo

- Equação de recorrência:
  - *T(n) = 5*n + T(n/2), T(1) = 5* (é preciso do valor de *n=1*)
  - *T(2) = 5*2 + T(2/2) = 15*
  - *T(4) = 5*4 + T(4/2) = 35*
  - Precisamos de k-1 passos para o valor de $T(2^k)$
- Precisamos substituir os termos *T(k)*, *k < n*, até que todos os sejam substituídos por fórmulas apenas com T(1)

Complexidade de tempo

- Equação de recorrência:
  - $T(n) = 5*n + T(n/2)$
  - $T(n/2) = 5*(n/2) + T(n/2/2)$
  - $T(n/2/2) = 5*(n/2/2) + T(n/2/2/2)$

...

- $T(n/2/2.../2) = 5*(n/2/2.../2) + T(n/2.../2)$

Complexidade de tempo

- $T(n) = 5n + 5n*(½) + 5n*(1/2^2) + 5n*(1/2^3) + ... + (n/2/2...2)$
- Soma de série geométria de razão ½ multiplicada por 5*n* e adicionada de T(n/2/2.../2) que é menor ou igual a 1
  - Desprezamos o termo final quando *n* tende a infinito
- $T(n) = 5n*\Sigma_{i=0}^{\infty} (1/2)^i = n(1/(1-½)) = 5n*2 = 10n$
- Logo, o programa é *O(n)*

Referências

- [1] ZIVIANI, N. Projeto de Algoritmos. 3º edição revisada e ampliada. Cengage Learning, 2017.
- https://pt.wikipedia.org/wiki/Bubble_sort
- https://en.wikipedia.org/wiki/Bubble_sort
- https://www.geeksforgeeks.org/bubble-sort/
- https://pt.wikipedia.org/wiki/Insertion_sort
- https://en.wikipedia.org/wiki/Insertion_sort
- https://www.geeksforgeeks.org/insertion-sort/
- https://pt.wikipedia.org/wiki/Merge_sort
- https://en.wikipedia.org/wiki/Merge_sort
- https://www.geeksforgeeks.org/merge-sort/