Análise e Técnicas de Algoritmos

Jorge Figueiredo

Análise de Algoritmos

Agenda

- Motivação para análise de algoritmos
- Análise assintótica
- Alguns exemplos simples

Introdução

- Dois aspectos importantes:
  - Um problema pode, geralmente, ser resolvido por diferentes algoritmos.
  - A existência de um algoritmo não implica, necessariamente, que este problema possa ser resolvido na prática.
- A análise de algoritmos pode ser definida como o estudo da estimativa de tempo de execução de algoritmos.
- O tempo de execução é determinado pelos seguintes aspectos:
  - Tempo para executar uma instrução ou passo.
  - A natureza do algoritmo.
  - O tamanho do conjunto de dados que constitui o problema.

Introdução

- É necessário ter uma forma de criar medidas de comparação entre algoritmos que resolvem um mesmo problema. Desta forma, é possível determinar:
  - A viabilidade de um algoritmo.
  - Qual é o melhor algoritmo para a solução de um problema.
- O interessante é ter uma comparação relativa entre algoritmos.
  - Assumir que a execução de qualquer passo de um algoritmo leva uma um tempo fixo e igual.
  - O tempo de execução de um computador particular não é interessante.

Introdução

- Qual a quantidade de recursos utilizados para resolver um problema?
  - Tempo
  - Espaço
- Expressar como uma função do tamanho do problema.
  - Como os requisitos crescem com o aumento do problema?
- Tamanho do problema:
  - Número de elementos a ser tratado
  - Tamanho dos elementos

Eficiência de Algoritmo

- Considerar eficiência de tempo:
  - Número de operações expresso em termos do tamanho da entrada.
  - Se dobramos o tamanho da entrada, qual o tempo de resposta?
- Por que eficiência é importante?
  - Velocidade de computação aumentou (hardware)
  - Crescimento de aplicações com o aumento do poder computacional
  - Maior demanda por aumento na velocidade de computação.

Eficiência de Algoritmo

- Quando a velocidade de computação aumenta, podemos tratar mais dados?
- Suponha que:
  - Um algoritmo toma n^2 comparações para ordenar *n* números.
  - Necessitamos de *1* segundo para ordenar ***5*** números (25 comparações)
  - Velocidade de computação aumenta de um fator de ***100***
  - Usando 1 segundo, podemos executar ***100x25*** comparações, i.e., ordenar ***50*** números

Com ***100*** vezes de ganho em velocidade, ordenamos apenas ***10*** vezes mais números!

Eficiência de Algoritmo

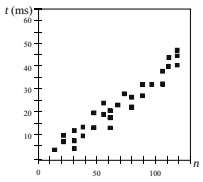
N	T(n) = n	T(n) = nlgn	T(n) = n^2	T(n) = n^3	Tn = 2^n
5	0.005 μs	0.01 μs	0.03 μs	0.13 μs	0.03 μs
10	0.01 μs	0.03 μs	0.1 μs	1 μs	1 μs
20	0.02 μs	0.09 μs	0.4 μs	8 μs	1 ms
50	0.05 μs	0.28 μs	2.5 μs	125 μs	13 dias
100	0.1 μs	0.66 μs	10 μs	1 ms	4 x 10^{13} anos

Como Medir Eficiência de Algoritmo?

- Medindo eficiência:
  - Estudo experimental e/ou Benchmarking.
  - Análise assintótica.

Abordagem Experimental

- Abordagem experimental:
  - Escrever um programa que implementa o algoritmo
  - Executar o programa com diferentes cenários
  - Usar um método como System.currentTimeMillis() para obter medidas acuradas do tempo de execução real.



Abordagem Experimental

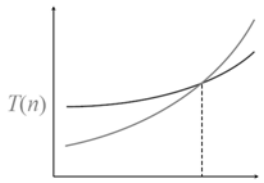
- Limitações dos estudos experimentais:
  - Necessidade de se implementar e testar o algoritmo.
  - Experimentos podem ser feitos apenas em um número limitado de cenários. Pode, portanto, não indicar tempo de execução em cenários que não foram considerados no experimento.
  - Para comparar dois algoritmos: garantir os mesmos hardware e ambiente de software.

Análise Assintótica

- Metodologia para analisar tempo de execução de algoritmos.
- Ao contrário da abordagem experimental:
  - Usa uma descrição de alto nível dos algoritmos em vez de testar uma de suas implementações.
  - Leva em consideração todas as possíveis entradas.
  - Permite a avaliação de eficiência de algoritmos de uma forma que é independente do hardware e ambiente de software utilizado.

Notação Assintótica

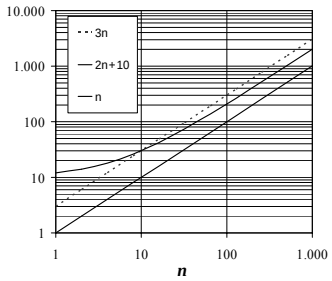
•**Objetivo**: simplificar a análise descartando informações desnecessárias:
- –"arredondamento" 1,000,001≈1,000,000
- –Dizer que $3n^2 \approx n^2$



Big-Oh

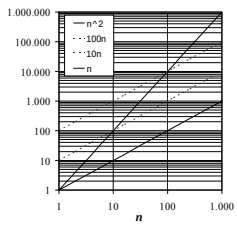
- Sejam duas funções $f(n)$ e $g(n)$.
  - $f(n)$ é $O(g(n))$ se existem constantes positivas c e n_0 tais que $f(n) \leq cg(n)$ para $n \geq n_0$
- Exemplo: $2n + 10$ é $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Escolher $c = 3$ e $n_0 = 10$

Big-Oh



Big-Oh

•Exemplo: n^2 não é $O(n)$
- $n^2 \leq cn$
- $n \leq c$
- – A desigualdade não pode ser satisfeita pois c deve ser uma constante.



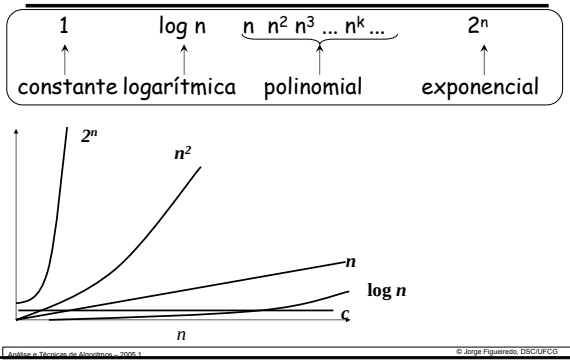
Big-Omega

- Sejam duas funções $f(n)$ e $g(n)$.
  - $f(n)$ é $\Omega(g(n))$ se $g(n)$ é $O(f(n))$
  - Existe uma constante real $c > 0$ e $n_0 \geq 1$ tal que $f(n) \geq c\, g(n)$ para $n \geq n_0$

Big-Theta

- Sejam duas funções $f(n)$ e $g(n)$
  - $f(n)$ é $\Theta(g(n))$ se $f(n)$ é $O(g(n))$ e $f(n)$ é $\Omega(g(n))$
  - existem constantes reais $a > 0$ e $b > 0$, e uma constante inteira $n_0 \geq 1$ tal que $a\, g(n) \geq f(n) \geq b\, g(n)$ para $n \geq n_0$

Taxas de Crescimento Mais Comum



Análise Assintótica

- A análise assintótica é baseada nessas definições e estabelece uma ordem relativa entre funções.
- A notação Big-Oh é usada para expressar o número de operações primitivas executadas como função do tamanho da entrada.
  - um algoritmo que executa em tempo ***O***(n) é melhor do que um que executa em tempo ***O***(n^2)
  - de forma semelhante, ***O***(log n) é melhor do que ***O***(n)
- Cuidado! Preste atenção a constantes muito altas. Um algoritmo que executa em tempo *1,000,000.n* é ***O***(n), mas menos eficiente do que um que executa em tempo $2n^2$, que é ***O***(n^2).

Complexidade de Tempo

- Para determinar o tempo de execução de um determinado algoritmo
  - descobrir a forma geral da curva que caracteriza seu tempo de execução em função do tamanho do problema.
- Para simplificarmos a análise de complexidade de tempo:
  - adotamos a não existência de unidades de tempo particulares.
  - não consideramos também os termos de ordem inferior, isto é, usamos Big-Oh.
- A complexidade de tempo para diferentes algoritmos pode indicar diferentes classes de complexidade. Cada classe é caracterizada por uma família diferente de curva.

Complexidade de Tempo

- Informalmente, para se determinar a ordem de complexidade de uma determinada função f(n):
  1. Separar f(n) em duas partes: termo dominante e termos de ordem inferior.
  2. ignorar os termos de ordem inferior.
  3. ignorar as constantes de proporcionalidade.

Análise de Algoritmos Simples

- Em nosso modelo de análise, consideramos que as instruções são executadas sequencialmente e que o conjunto de instruções simples (adição, comparação, atribuição, etc) tomam exatamente uma unidade de tempo para serem executadas.
- Tipos de análise:

  Pior caso: indica o maior tempo obtido, levando em consideração todas as entradas possíveis.

  Melhor caso: indica o menor tempo obtido, levando em consideração todas as entradas possíveis.

  Média: indica o tempo médio obtido, considerando todas as entradas possíveis.

Análise de Algoritmos Simples

- Como o objetivo é determinar a forma da curva que caracteriza o algoritmo, vamos definir algumas regras que podem ser utilizadas:

  Laços: O tempo de execução de um laço é no máximo o tempo de execução das instruções dentro do laço (incluindo os testes) vezes o número de iterações.

  Aninhamento de Laços: Analisar os mais internos. O tempo total de execução de uma instrução dentro de um grupo de laços aninhados é o tempo de execução da instrução multiplicado pelo produto dos tamanhos de todos os laços.

  Instruções Consecutivas: Apenas efetuar a soma.

  If/Else: o tempo de execução de uma instrução if/else nunca é maior do que o tempo de execução do teste mais o maior dos tempos de execução de S1 e S2. S1 e S2 representam as instruções do *then* e *else*, respectivamente.

Análise de Algoritmos Simples

Chamada de Funções: A análise é feita como no caso de laços aninhados. Para calcular a complexidade de um programa com várias funções, determina-se primeiro a complexidade de cada uma das funções. Desta forma, na análise, cada uma das funções é vista como uma instrução com a complexidade que foi calculada.

Recursão: É a parte mais difícil da análise de complexidade. Em muitos casos, pode-se fazer a linearização através da substituição da chamada recursiva por alguns laços aninhados, por exemplo. Entretanto, existem algoritmos que não possibilitam a linearização. Nestes caso, é necessário usar uma relação de recorrência que tem que ser resolvida.

Algumas Dicas

- Identificar a operação fundamental usada no algoritmo. A análise dessa operação fundamental identifica o tempo de execução. Isso pode evitar a análise linha-por-linha do algoritmo.
- Quando um algoritmo, em uma passada de uma iteração, divide o conjunto de dados de entrada em uma ou mais partes, tomado cada uma dessas partes e processando separada e recursivamente, e depois juntando os resultados, este algoritmo é possivelmente O(n.log n)
- Um algoritmo é O(log n) se ele leva tempo constante para reduzir o tamanho do problema, geralmente pela metade. Por exemplo, a pesquisa binária.
- Se o algoritmo leva tempo constante para reduzir o tamanho do problema em um tamanho constante, ele será O(n).
- Algoritmos combinatoriais são exponenciais. Por exemplo, o problema do caixeiro viajante.

Exercício

Qual a complexidade do algoritmo abaixo?

```
Potencia(x, n)
p ← 1
i ← 0
while i < n do
      p ← p.x
      i ← i + 1
return p
```