Complexidade de Algoritmos

Siang Wun Song - Universidade de São Paulo - IME/USP

MAC 5710 - Estruturas de Dados - 2008

Objetivo de estudar complexidade de algoritmos

Por que analisar a complexidade dos algoritmos?

- A preocupação com a complexidade de algoritmos é fundamental para projetar algoritmos eficientes.
- Podemos desenvolver um algoritmo e depois analisar a sua complexidade para verificar a sua eficiência.
- Mas o melhor ainda é ter a preocupação de projetar algoritmos eficientes desde a sua concepção.

Para um dado problema considere dois algoritmos que o resolvem.

- Seja n um parâmetro que caracteriza o tamanho da entrada do algoritmo. Por exemplo, ordenar n números ou multiplcar duas matrizes quadradas $n \times n$ (cada uma com n^2 elementos).
- Como podemos comparar os dois algoritmos para escolher o melhor?

Complexidade de tempo ou de espaço

- Precisamos definir alguma medida que expresse a eficiência. Costuma-se medir um algoritmo em termos de tempo de execução ou o espaço (ou memória) usado.
  - Para o tempo, podemos considerar o tempo absoluto (em minutos, segundos, etc.). Medir o tempo absoluto não é interessante por depender da máquina.
  - Em Análise de Algoritmos conta-se o número de operações consideradas relevantes realizadas pelo algoritmo e expressa-se esse número como uma função de n. Essas operações podem ser comparações, operações aritméticas, movimento de dados, etc.

Pior caso, melhor caso, caso médio

- O número de operações realizadas por um determinado algoritmo pode depender da particular instância da entrada. Em geral interessa-nos o pior caso, i.e., o maior número de operações usadas para qualquer entrada de tamanho n.
- Análises também podem ser feitas para o melhor caso e o caso médio. Neste último, supõe-se conhecida uma certa distribuição da entrada.
- Exemplo: Busca seqüencial de um dado elemento em um vetor armazenando n elementos de forma aleatória. Discuta o pior caso, melhor caso e o caso médio. No caso médio suponha a distribuição uniforme e que o dado buscado está dentro do vetor. Como muda o caso médio se o dado em geral não está presente?

Como exemplo, considere o número de operações de cada um dos dois algoritmos que resolvem o mesmo problema, como função de n.

- Algoritmo 1: $f_1(n) = 2n^2 + 5n$ operações
- Algoritmo 2: $f_2(n) = 500n + 4000$ operações

Dependendo do valor de n, o Algoritmo 1 pode requerer mais ou menos operações que o Algoritmo 2.

(Compare as duas funções para $n = 10$ e $n = 100$.)

Comportamento assintótico

- Algoritmo 1: $f_1(n) = 2n^2 + 5n$ operações
- Algoritmo 2: $f_2(n) = 500n + 4000$ operações

Um caso de particular interesse é quando n tem valor muito grande ($n \to \infty$), denominado *comportamento assintótico*.

Os termos inferiores e as constantes multiplicativas contribuem pouco na comparação e podem ser descartados.

O importante é observar que $f_1(n)$ cresce com n^2 ao passo que $f_2(n)$ cresce com n. Um crescimento quadrático é considerado pior que um crescimento linear. Assim, vamos preferir o Algoritmo 2 ao Algoritmo 1.

Notação O

Dada uma função $g(n)$, denotamos por $O(g(n))$ o conjunto das funções

$\{ f(n) : \exists$ constantes c e n_0 tais que $0 \leq f(n) \leq cg(n)$ para $n \geq n_0.\}$

Isto é, para valores de n suficientemente grandes, $f(n)$ é igual ou menor que $g(n)$.

Como abuso de notação, vamos escrever $f(n) = O(g(n))$ ao invés de $f(n) \in O(g(n))$.

- Algoritmo 1: $f_1(n) = 2n^2 + 5n = O(n^2)$
- Algoritmo 2: $f_2(n) = 500n + 4000 = O(n)$

Um polinômio de grau d é de ordem $O(n^d)$. Como uma constante pode ser considerada como um polinômio de grau 0, então dizemos que uma constante é $O(n^0)$ ou seja $O(1)$.

Exercícios sobre notação O

Repetimos a definição:
Dada uma função $g(n)$, denotamos por $O(g(n))$ o conjunto das funções

$\{ f(n) : \exists$ constantes c e n_0 tais que $0 \leq f(n) \leq cg(n)$ para $n \geq n_0 .\}$

Isto é, para valores de n suficientemente grandes, $f(n)$ é igual ou menor que $g(n)$.

Como abuso de notação, vamos escrever $f(n) = O(g(n))$ ao invés de $f(n) \in O(g(n))$.

1. É verdade que $2n^2 + 100n = O(n^2)$? Prove.
2. É verdade que $10 + \frac{4}{n} = O(n^0) = O(1)$? Prove.
3. Escreva a seguinte função em notação O, sem provar: $4n^2 + 10 \log n + 500$.
4. Mesmo para a função. $5n^n + 102^n$
5. Mesmo para a função. $2(n-1)^n + n^{n-1}$

Notação Ω

Dada uma função $g(n)$, denotamos por $\Omega(g(n))$ o conjunto das funções

$\{ f(n) : \exists$ constantes c e n_0 tais que $0 \leq cg(n) \leq f(n)$ para $n \geq n_0.\}$

Isto é, para valores de n suficientemente grandes, $f(n)$ é igual ou maior que $g(n)$.

Novamente, abusando a notação, vamos escrever $f(n) = \Omega(g(n))$.

Notação Θ

Dadas duas funções $f(n)$ e $g(n)$, temos

$f(n) = \Theta(g(n))$

se e somente se

$f(n) = O(g(n))$ e

$f(n) = \Omega(g(n))$.

Importância

Considere 5 algoritmos com as complexidades de tempo. Suponhamos que uma operação leve 1 ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
512	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

(Verifique se resolveria usar uma máquina mais rápida onde uma operação leve 1 ps (pico segundo) ao invés de 1 ms: ao invés de 10^{137} séculos seriam 10^{128} séculos :-)

Podemos muitas vezes melhorar o tempo de execução de um programa otimizando o código (e.g. usar $x + x$ ao invés de $2x$, evitar re-cálculo de expressões já calculadas, etc.).

Entretanto, melhorias muito mais substanciais podem ser obtidas se usarmos um algoritmo diferente, com outra complexidade de tempo, e.g. obter um algoritmo de $O(n \log n)$ ao invés de $O(n^2)$.

Importância

Considere 5 algoritmos com as complexidades de tempo. Suponhamos que uma operação leve 1 ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
512	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

(Verifique se resolveria usar uma máquina mais rápida onde uma operação leve 1 ps (pico segundo) ao invés de 1 ms: ao invés de 10^{137} séculos seriam 10^{128} séculos :-)

Podemos muitas vezes melhorar o tempo de execução de um programa otimizando o código (e.g. usar $x + x$ ao invés de $2x$, evitar re-cálculo de expressões já calculadas, etc.).

Entretanto, melhorias muito mais substanciais podem ser obtidas se usarmos um algoritmo diferente, com outra complexidade de tempo, e.g. obter um algoritmo de $O(n \log n)$ ao invés de $O(n^2)$.

Importância

Considere 5 algoritmos com as complexidades de tempo. Suponhamos que uma operação leve 1 ms.

n	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
512	0.512s	9s	4m 22s	1 dia 13h	10^{137} séculos

(Verifique se resolveria usar uma máquina mais rápida onde uma operação leve 1 ps (pico segundo) ao invés de 1 ms: ao invés de 10^{137} séculos seriam 10^{128} séculos :-)

Podemos muitas vezes melhorar o tempo de execução de um programa otimizando o código (e.g. usar $x + x$ ao invés de $2x$, evitar re-cálculo de expressões já calculadas, etc.).

Entretanto, melhorias muito mais substanciais podem ser obtidas se usarmos um algoritmo diferente, com outra complexidade de tempo, e.g. obter um algoritmo de $O(n \log n)$ ao invés de $O(n^2)$.

Cota superior ou limite superior (*upper bound*)

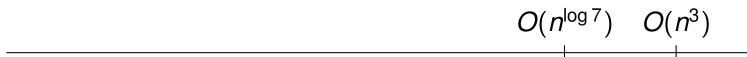
- Seja dado um problema, por exemplo, multiplicação de duas matrizes quadradas $n \times n$.
- Conhecemos um algoritmo para resolver este problema (pelo método trivial) de complexidade $O(n^3)$.
- Sabemos assim que a complexidade deste problema não deve superar $O(n^3)$, uma vez que existe um algoritmo que o resolve com esta complexidade.
- Uma cota superior ou limite superior (*upper bound*) deste problema é $O(n^3)$.

$O(n^3)$

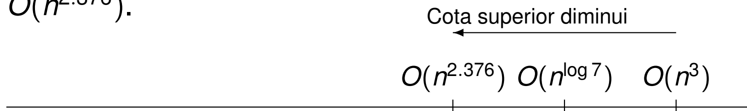
- A cota superior de um problema pode mudar se alguém descobrir um outro algoritmo melhor.

Cota superior (*upper bound*)

- O Algoritmo de Strassen reduziu a complexidade para $O(n^{\log 7})$. Assim a cota superior do problema de multiplicação de matrizes passou a ser $O(n^{\log 7})$.



- Coppersmith e Winograd melhoraram ainda para $O(n^{2.376})$.



- Note que a cota superior de um problema depende do algoritmo. Pode diminuir quando aparece um algoritmo melhor.

Analogia com *record mundial*

A cota superior para resolver um problema é análoga ao *record mundial* de uma modalidade de atletismo. Ele é estabelecido pelo melhor atleta (algoritmo) do momento. Assim como o *record mundial*, a cota superior pode ser melhorada por um algoritmo (atleta) mais veloz.

“Cota superior” da corrida de 100 metros rasos:

Ano	Atleta (Algoritmo)	Tempo
1988	Carl Lewis	9s92
1993	Linford Christie	9s87
1993	Carl Lewis	9s86
1994	Leroy Burrell	9s84
1996	Donovan Bailey	9s84
1999	Maurice Greene	9s79
2002	Tim Montgomery	9s78
2007	Asafa Powell	9s74
2008	Usain Bolt	9s72
2008	Usain Bolt	9s69
2009	Usain Bolt	9s58

Seqüência de Fibonacci

Para projetar um algoritmo eficiente, é fundamental preocupar-se com a sua complexidade. Como exemplo: considere a seqüência de Fibonacci.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

A seqüência pode ser definida recursivamente:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Dado o valor de n, queremos obter o n-ésimo elemento da seqüência.

Vamos apresentar dois algoritmos e analisar sua complexidade.

Algoritmo 1: função $fibo1(n)$

Seja a função $fibo1(n)$ que calcula o n-ésimo elemento da seqüência de Fibonacci.

Input: Valor de n

Output: O n-ésimo elemento da seqüência de Fibonacci

Function $fibo1(n)$

1: **if** $n = 0$ **then**
2:     **return** 0
3: **else**
4:     **if** $n = 1$ **then**
5:         **return** 1
6:     **else**
7:         **return** $fibo1(n-1) + fibo1(n-2)$
8:     **end if**
9: **end if**

Experimente rodar este algoritmo para $n = 100$:-)

A complexidade é $O(2^n)$.

(Mesmo se uma operação levasse um picosegundo, 2^{100} operações levariam 3×10^{13} anos = 30.000.000.000.000 anos.)

Algoritmo 1: função $fibo1(n)$

Seja a função $fibo1(n)$ que calcula o n-ésimo elemento da seqüência de Fibonacci.
Input: Valor de n
Output: O n-ésimo elemento da seqüência de Fibonacci
Function $fibo1(n)$

1: **if** $n = 0$ **then**
2:     **return** 0
3: **else**
4:     **if** $n = 1$ **then**
5:         **return** 1
6:     **else**
7:         **return** $fibo1(n - 1) + fibo1(n - 2)$
8:     **end if**
9: **end if**

Experimente rodar este algoritmo para $n = 100$:-)
A complexidade é $O(2^n)$.
(Mesmo se uma operação levasse um picosegundo, 2^{100} operações levariam 3×10^{13} anos = 30.000.000.000.000 anos.)

Algoritmo 2: função *fibo2*(n)

Function *fibo2*(n)

```
1:  if n = 0 then
2:      return 0
3:  else
4:      if n = 1 then
5:          return 1
6:      else
7:          penultimo ← 0
8:          ultimo ← 1
9:          for i ← 2 until n do
10:             atual ← penultimo + ultimo
11:             penultimo ← ultimo
12:             ultimo ← atual
13:         end for
14:         return atual
15:     end if
16: end if
```

A complexidade agora passou de $O(2^n)$ para $O(n)$.

Voce sabe que dá para fazer em $O(\log n)$?

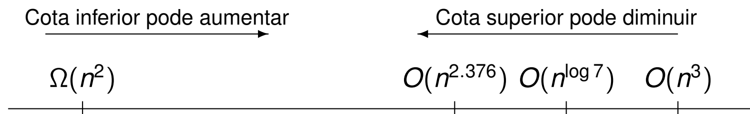
Cota inferior (*lower bound*)

- As vezes é possível demonstrar que, para um dado problema, qualquer que seja o algoritmo a ser usado, o problema requer pelo menos um certo número de operações.
- Essa complexidade é chamada cota inferior (*lower bound*) do problema.
- Note que a cota inferior depende do problema mas não do particular algoritmo.

Cota inferior para multiplicação de matrizes

- Para o problema de multiplicação de matrizes quadradas $n \times n$, apenas para ler os elementos das duas matrizes de entrada ou para produzir os elementos da matriz produto leva tempo $O(n^2)$. Assim uma cota inferior trivial é $\Omega(n^2)$.
- Na analogia anterior, uma cota inferior de uma modalidade de atletismo não dependeria mais do atleta. Seria algum tempo mínimo que a modalidade exige, qualquer que seja o atleta. Uma cota inferior trivial para os 100 metros rasos seria o tempo que a velocidade da luz leva para percorrer 100 metros no vácuo.

Meta: aproximando as duas cotas



- Se um algoritmo tem uma complexidade que é igual à cota inferior do problema, então ele é assintoticamente ótimo.
- O algoritmo de Coppersmith e Winograd é de $O(n^{2.376})$ mas a cota inferior (conhecida até hoje) é de $\Omega(n^2)$. Portanto não podemos dizer que ele é ótimo.
- Pode ser que esta cota superior possa ainda ser melhorada. Pode também ser que a cota inferior de $\Omega(n^2)$ possa ser melhorada (isto é "aumentada"). Para muitos problemas interessantes, pesquisadores dedicam seu tempo tentando encurtar o intervalo ("gap") até encostar as duas cotas.