Análise de Algoritmos — O caso não recursivo

Flávio Velloso Laper

Universidade Fumec

9 de maio de 2013

Conteúdo

Conteúdo

UNIVERSIDADE FUMEC

Por que analisar algoritmos?

- Para determinar sua correção.
- Para determinar sua eficiência na utilização de algum recurso:
    - Tempo.
    - Memória.
    - Potência.
    - Espaço.
- Para comparar algoritmos.

Por que analisar algoritmos?

- Para determinar sua correção.
- Para determinar sua eficiência na utilização de algum recurso:
  - Tempo.
  - Memória.
  - Potência.
  - Espaço.
- Para comparar algoritmos.

Fatores que influenciam o tempo de execução

- Velocidade da máquina que executa o programa.
- Linguagem de implementação do programa.
- Eficiência do compilador.
- Tamanho da entrada.
- Organização da entrada.

Fatores que influenciam o tempo de execução

- Velocidade da máquina que executa o programa.
- Linguagem de implementação do programa.
- Eficiência do compilador.
- Tamanho da entrada: $T(n)$.
- Organização da entrada: pior caso, caso médio.

Conteúdo

Generalização do tempo de execução

Pergunta central

Como o tempo de execução varia à medida que o tamanho da entrada cresce?

- Ênfase: ordens de magnitude e taxas de crescimento.
- Funções típicas:

n	(1)	$\log n$	n	$n \log n$	n^2	n^3	2^n
5	1	3	5	15	25	125	32
10	1	4	10	33	100	10^3	10^3
100	1	7	100	644	10^4	10^6	10^{30}
1000	1	10	1000	10^4	10^6	10^9	10^{300}
10.000	1	13	10.000	10^5	10^8	10^{12}	10^{3000}

UNIVERSIDADE FUMEC

Análise do tempo de execução

Tempo de execução

O tempo de execução $T(n)$ de um algoritmo, para uma entrada de tamanho n, é proporcional ao número de instruções executadas.

- Vamos contar instruções!
- Exemplo: cálculo da média de n números:

$n \leftarrow$ lê entrada do usuário;
$soma \leftarrow 0$;
$i \leftarrow 0$;
enquanto $i < n$ **faça**
    $número \leftarrow$ lê entrada do usuário;
    $soma \leftarrow soma + número$;
    $i \leftarrow i + 1$;
$média \leftarrow soma/n$;

Resultado da análise do tempo de execução

Instrução	Número de execuções
1	1
2	1
3	1
4	$n+1$
5	n
6	n
7	n
8	1

- Total: $T(n) = 4n + 5$.
- Complexidade: $T(n) = 4n + 5 = O(n)$ (linear).

Conteúdo

Notação O-grande

Definição

Definição simplificada

Sejam $f(n)$ e $g(n)$ duas funções. Escreve-se

$$f(n) = O(g(n)) \quad \text{ou} \quad f = O(g)$$

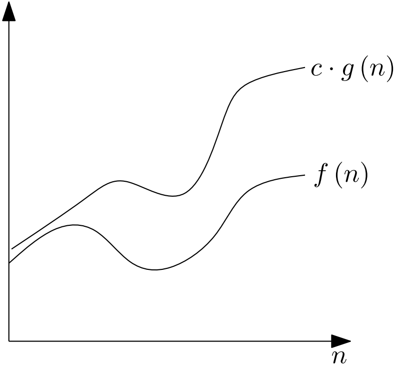
(leia-se "f de n é O-grande de g de n" ou "f é O-grande de g") se existe um inteiro positivo C tal que $f(n) \leq C \cdot g(n)$ para todo inteiro positivo n.

Ideia central: estabelecimento de limite superior.

Notação O-grande

Interpretação gráfica

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



Notação O-grande

Exemplos

Exemplos:

- $5n = O(n)$.
- $4n + 5 = O(n)$.
- $n^2 \neq O(n)$.
- $n^2 + 3n - 1 = O\left(n^2\right)$.
- $n^2 + 3n - 1 = O\left(n^3\right)$.
- $2n^7 - 6n^5 + 10n^2 - 5 = O\left(n^7\right)$.

Notação O-grande

Polinômios

Resultado geral para polinômios

Todo polinômio é O-grande de seu termo de maior grau:

$$a_k n^k + a_{k-1} n^{k-1} + a_{k-2} n^{k-2} + \cdots + a_2 n^2 + a_1 n + a_0 = O\left(n^k\right)$$

Constantes são ignoradas.

Técnica para demonstração:

- Tomar o valor absoluto de cada coeficiente.
- Mudar todos os expoentes para o termo de maior grau ($n^j \leq n^d$ se $j \leq d$).
- Adicionar termos para obter C.

Notação O-grande

Interpretação

- O-grande: aproximação do limite superior do tempo de execução do algoritmo.
- *Pergunta*: sejam dois algoritmos $O(n)$ e $O\left(n^2\right)$. Qual deles é melhor?

Notação O-grande

Interpretação

- O-grande: aproximação do limite superior do tempo de execução do algoritmo.
- *Pergunta*: sejam dois algoritmos $O(n)$ e $O\left(n^2\right)$. Qual deles é melhor?
- O-grande não indica quão bom é um algoritmo.

Notação O-grande

Interpretação

- O-grande: aproximação do limite superior do tempo de execução do algoritmo.
- *Pergunta*: sejam dois algoritmos $O(n)$ e $O\left(n^2\right)$. Qual deles é melhor?
- O-grande não indica quão bom é um algoritmo.
- O-grande fornece um limite superior para o quão ruim ele pode ser.

Notação O-grande

Interpretação

- O-grande: aproximação do limite superior do tempo de execução do algoritmo.
- *Pergunta*: sejam dois algoritmos $O(n)$ e $O(n^2)$. Qual deles é melhor?
- O-grande não indica quão bom é um algoritmo.
- O-grande fornece um limite superior para o quão ruim ele pode ser.
- Outros limites:
  - Limite inferior da taxa de crescimento: Ω-grande.
    - $T(n) = \Omega(g(n))$: o melhor tempo de execução é $g(n)$ (melhor caso).
  - Limite estrito da taxa de crescimento: Θ-grande.
    - Mais preciso: estabelece limites superior e inferior.
    - Um algoritmo $\Theta(n)$ é melhor que um $\Theta(n^2)$.

Notações assintóticas

Definições

Notação O-grande (definição rigorosa)

Sejam $f(n)$ e $g(n)$ duas funções. Escreve-se: $f(n) = O(g(n))$ se existem inteiros positivos C e N tais que $f(n) \leq C \cdot g(n)$ para todo inteiro $n \geq N$.

Notação Ω-grande

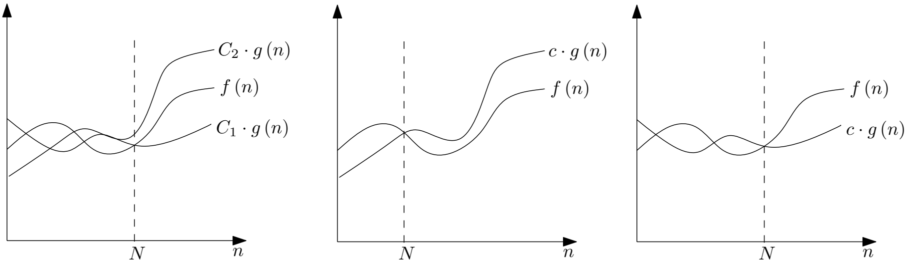
Sejam $f(n)$ e $g(n)$ duas funções. Escreve-se: $f(n) = \Omega(g(n))$ se existem inteiros positivos C e N tais que $f(n) \geq C \cdot g(n)$ para todo inteiro $n \geq N$.

Notação Θ-grande

Sejam $f(n)$ e $g(n)$ duas funções. Escreve-se: $f(n) = \Theta(g(n))$ se existem inteiros positivos C_1, C_2 e N tais que $C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$ para todo inteiro $n \geq N$.

Notações assintóticas

Interpretação gráfica



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

Notações assintóticas

Relacionamento entre as notações

Teorema 1

$f(n) = O(g(n))$ se, e somente se, $g(n) = \Omega(f(n))$.

Teorema 2

Para duas funções quaisquer $f(n)$ e $g(n)$, $f(n) = \Theta(g(n))$ se, e somente se, $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.

Exemplo: $f(n) = 3n^3 + 3n - 1 = \Theta(n^3)$.

- Mostrar que $f(n) = 3n^3 + 3n - 1 = O(n^3)$.
- Mostrar que $f(n) = 3n^3 + 3n - 1 = \Omega(n^3)$.

Notações assintóticas

Operações O-grande

Regra da soma

Suponha que $T_1 = O(f_1(n))$ e $T_2 = O(f_2(n))$. Além disso, suponha que f_2 não cresça mais rápido que f_1, isto é, $f_2(n) = O(f_1(n))$. Então podemos concluir que $T_1(n) + T_2(n) = O(f_1(n))$. De maneira geral, a regra da soma diz que $O(f_1(n) + f_2(n)) = O(\max(f_1(n), f_2(n)))$.

Regra do produto

Suponha que $T_1(n) = O(f_1(n))$ e $T_2(n) = O(f_2(n))$. Então podemos concluir que $T_1(n) \cdot T_2(n) = O(f_1(n) \cdot f_2(n))$.

Observação: O-grande será usado como um limite estrito *intuitivo* (a menor função válida que caracterize o tempo de execução do algoritmo).

Conteúdo

Conteúdo

Regras para análise

Regras gerais:

- Instruções básicas: $O(1)$.
- Sequências: aplicar a regra da soma.
- Condicionais: soma da complexidade do teste com a complexidade do ramo executado (assumir o pior caso).
- Laços: soma, sobre todas as iterações, do tempo de execução das instruções do laço com o tempo de avaliação do teste de terminação (normalmente $O(1)$).
  - Ou seja, a complexidade do laço é a soma da complexidade do teste com a complexidade do corpo, multiplicada pelo número de iterações.
  - Utilizar a regra do produto.

Exemplo:

```
for (i = 2; i < n; i++) {
  sum += i;
}
```

Atenção ao enunciado!

- Número de instruções executadas → equação em termos de n com o número *preciso* de instruções.
- Complexidade → expressão O-grande (ou Θ-grande).

Exemplo: encontrar o número de instruções e a complexidade:

```
for (i = 1; i < n; i++) {
  SmallPos = i;
  Smallest = Array[SmallPos];
  for (j = i+1; j <= n; j++)
    if (Array[j] < Smallest) {
      SmallPos = j;
      Smallest = Array[SmallPos];
    }
  Array[SmallPos] = Array[i];
  Array[i] = Smallest;
}
```

Outro exemplo

Encontrar o número de instruções executadas e a complexidade:

```
cin >> n;
for(i = 1; i <= n; i++)
  for(j = 1; j <= n; j++)
    A[i][j] = 0;
for(i = 1; i <= n; i++)
  A[i][j] = 1;
```

Conteúdo

Análise de pior caso e caso médio

Análise de pior caso

Análise que procura pelo maior número possível de passos necessários para a execução de um programa.

Análise de caso médio

Análise que procura o número médio de instruções executadas, dependendo das diferentes formas em que a entrada de tamanho n pode estar organizada.

Análise de pior caso e caso médio

Exemplo

Exemplo: pesquisa de elemento dentro de vetor:

```
i = 0;
while((i < n) && (x != a[i]))
  i++;
if(i < n)
  location = i;
else
  location = -1;
```

Conteúdo

Análise de programas com chamadas não recursivas

Regras:

- Analisar cada função para obter seu tempo de execução $O(f(n))$.
  - Começar pelas funções que não chamam outras funções.
- Avaliar as funções que chamam as previamente avaliadas.
  - O tempo de execução da chamada terá a complexidade calculada para a função.
- Utilizar as regras da soma e do produto sempre que necessário.

Análise de programas com chamadas não recursivas

Exemplo

```
int a, n, x;

int bar(int x, int n) {
  int i;

  for(i = 1; i < n; i++)
    x = x + i;
  return x;
}
```

```
int foo(int x, int n) {
  int i;
  for(i = 1; i <= n; i++)
    x = x + bar(i, n);
  return x;
}

void main(void) {
  n = GetInteger();
  a = 0;
  x = foo(a, n);
  printf("%d", bar(a, n));
}
```

Análise de programas com chamadas não recursivas

Exercício

Qual a complexidade do trecho de código abaixo?

```
sum = 0;
for(i = 1; i <= f(n); i++)
  sum += i;
```

Considere que o tempo de execução de $f(n)$ é $O(n)$ e o *valor* de $f(n)$ é $n!$.

Conteúdo

Conteúdo

O problema da mochila

1	Salgadinhos	200 calorias	100 gramas
2	Coca-Cola *Diet*	1 caloria	200 gramas
...	...	...	...
200	Espaguete desidratado	500 calorias	450 gramas

O problema da mochila

1	Salgadinhos	200 calorias	100 gramas
2	Coca-Cola *Diet*	1 caloria	200 gramas
...	...	...	...
200	Espaguete desidratado	500 calorias	450 gramas

- Não existe solução polinomial conhecida.

O problema da mochila

1	Salgadinhos	200 calorias	100 gramas
2	Coca-Cola *Diet*	1 caloria	200 gramas
...	...	...	...
200	Espaguete desidratado	500 calorias	450 gramas

- Não existe solução polinomial conhecida.
- Não existe *prova* de que não haja solução polinomial.

O problema da mochila

1	Salgadinhos	200 calorias	100 gramas
2	Coca-Cola *Diet*	1 caloria	200 gramas
...	...	...	...
200	Espaguete desidratado	500 calorias	450 gramas

- Não existe solução polinomial conhecida.
- Não existe *prova* de que não haja solução polinomial.
- Problemas NP.

Classificação de funções e problemas

Categorias de funções:

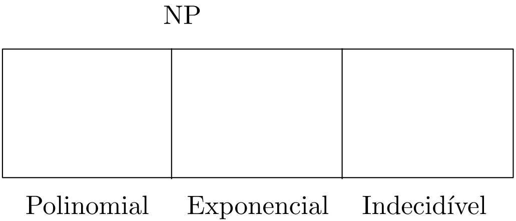
- Exponenciais: c^n, $n!$, n^n.
- Polinomiais: n^c.
  - Lineares: n.
  - Sublineares: $\log n$.
  - Constantes: crescimento independente de n.

Categorias de problemas:

- Intratáveis/Exponenciais.
- Polinomiais (P).
- Não-determinísticos polinomiais (NP).
- Indecidíveis.

UNIVERSIDADE FUMEC

Categorias de problemas



Exemplos de problemas NP:

- Satisfabilidade (*SAT*).
- Mochila.
- Cliques em teoria dos grafos.
- Caixeiro viajante (CV)

UNIVERSIDADE FUMEC

Problemas NP

Exemplo

O problema do caixeiro viajante

Dados um mapa de cidades e um custo de viagem entre cada par de cidades, é possível visitar cada cidade exatamente uma vez e retornar para casa por menos de k reais?

Algoritmo:

```
Escolha um dos possíveis caminhos;
Calcule o custo total do caminho escolhido;
se o custo calculado não é maior que o custo permitido então
    retorna sucesso;
senão
    retorna nada;
```

Problemas NP

Características e definição

Características:

- Cada problema é solúvel por enumeração.
- Há 2^n casos a considerar na enumeração.
  - Cada possibilidade pode ser testadas para resposta “sim” ou “não” em tempo pequeno.
- Problemas vêm de vários campos (lógica, grafos, teoria dos números, etc.)
- Se for possível “adivinhar uma solução” (processo não determinístico), pode ser resolvido em tempo pequeno.
  - A solução pode ser verificada em tempo polinomial.

Problemas NP

Conjunto de problemas que podem ser resolvidos por algoritmos não-determinísticos em tempo polinomial.

Ou: conjunto dos problemas cuja solução pode ser *verificada* em tempo polinomial.

Conteúdo

Problemas NP-completo

Definições

Definição informal

Problemas NP "difíceis": se um problema NP-completo tiver solução polinomial determinística, todos os problemas NP também terão.

Problema de decisão

Problemas que fornecem resultados "sim/não".

Tranformação polinomial

Sejam dois problemas Π_1 e Π_2. Se $\Pi_1 \propto \Pi_2$, então Π_1 pode ser reescrito como Π_2 e a resposta para Π_1 será "sim" se, e somente se, a resposta para Π_2 for "sim".

UNIVERSIDADE FUMEC

Transformação polinomial

```
Convert_To_P2 p1 = ... /* Toma uma instância de P1 e a
                          converte para uma instância de P2
                          em tempo polinomial */

Solve_P2 p2 = ... /* Resolve o problema P2 */

Solve_P1 p1 = Solve_P2(Convert_To_P2 p1);
```

Teoremas:

- Se $\Pi_1 \propto \Pi_2$ então $\Pi_2 \in P \rightarrow \Pi_1 \in P$.
- Se $\Pi_1 \propto \Pi_2$ então $\Pi_2 \notin P \rightarrow \Pi_1 \notin P$.
- Transitividade: se $\Pi_1 \propto \Pi_2$ e $\Pi_2 \propto \Pi_3$, então $\Pi_1 \propto \Pi_3$.

Problemas NP-completo

Definição formal

Um problema de decisão Π é NP-completo se, e somente se, $\Pi \in$ NP e, para todo $\Pi' \in$ NP, $\Pi' \propto \Pi$.

Para provar que Π é NP:

1. Encontre um problema NP-completo conhecido Π_{NP}.
2. Encontre uma transformação tal que $\Pi_{\mathrm{NP}} \propto \Pi$.
3. Prove que a transformação é polinomial.

Questão: qual é o "primeiro" problema NP-completo a utilizar?

Problemas NP-completo

Significado da NP-completude

Resultados importantes:

- $P \subseteq NP$.
- SAT é NP-completo!.
  - Se há um algoritmo polinomial determinístico para SAT, então há um algoritmo determinístico polinomial para todo problema em NP.
  - Ou seja: para todo problema $\Pi \in NP$, $\Pi \propto SAT$.
  - Há outros problemas NP-completo: mochila, cliques, ...
  - Se for provado que algum desses problemas não possui solução polinomial determinística, o mesmo acontecerá com *todo* problema NP.

Problemas NP-completo

Significado da NP-completude

Resultados importantes:

- $\text{P} \subseteq \text{NP}$.
- SAT é NP-completo!.
  - Se há um algoritmo polinomial determinístico para SAT, então há um algoritmo determinístico polinomial para todo problema em NP.
  - Ou seja: para todo problema $\Pi \in \text{NP}$, $\Pi \propto \text{SAT}$.
  - Há outros problemas NP-completo: mochila, cliques, ...
  - Se for provado que algum desses problemas não possui solução polinomial determinística, o mesmo acontecerá com *todo* problema NP.

Questão fundamental

$\text{P} = \text{NP}$?