

SME0827 - Estruturas de Dados

Análise de Algoritmos Aula 06 – Parte 1

Professor: André C. P. L. F. de Carvalho, ICMC-USP
PAE: Moisés Rocha dos Santos
Monitor:

© André de Carvalho - ICMC/USP

1

1

Aula deste módulo

- Introdução
- Análise de Algoritmos
- Notação Assintótica
- Análise Assintótica
- Indução
- Conclusão

© André de Carvalho - ICMC/USP

2

2

Introdução

- Implementações recursivas da série de Fibonacci
 - Baseada na definição matemática
 - Ineficiente
 - Baseada no conceito de sequência aditiva
 - Eficiente
- Implementações recursivas mal sucedidas podem dar a recursão uma reputação injusta
 - Recursão pode aumentar muito a eficiência, reduzindo o tempo de processamento

© André de Carvalho - ICMC/USP

3

3

Principais objetivos

- Ser capaz de medir a eficiência de um algoritmo
- Poder comparar algoritmos diferentes de acordo com eficiência de cada um
- Obter uma aproximação do tempo de execução de um algoritmo
- Aprender a provar eficiência por indução

© André de Carvalho - ICMC/USP

4

4

Introdução

- Suponha que você está começando um projeto de Ciência de Dados e precisa:
 - Escolher entre um dentre vários algoritmos que extraem as mesmas estatísticas dos dados
 - Como você escolhe um deles?

© André de Carvalho - ICMC/USP

5

5

Introdução

- Suponha que você está começando um projeto de Ciência de Dados e precisa:
 - Escolher entre um dentre vários algoritmos que extraem as mesmas estatísticas dos dados
 - Como você escolhe um deles?
 - Escolhe de forma aleatória?
 - Pede a sugestão de um colega?
 - Escolhe o algoritmo mais fácil de implementar?
 - Escolhe o algoritmo mais eficiente?

© André de Carvalho - ICMC/USP

6

6

Introdução

- Algoritmo eficiente
 - O que é eficiência?
 - Como pode ser medida?
 - A resposta é dada por uma das principais áreas da Computação:
 - Análise de algoritmos
 - Mede a eficiência de algoritmos analisando sua complexidade
 - Ex.: Análise de eficiência de algoritmos de busca

© André de Carvalho - ICMC/USP 7

7

Introdução

- Como medir a eficiência de um algoritmo?
 - Tempo de execução
 - Quantidade de tempo necessário para processar os dados
 - Utilização do espaço
 - Quantidade de memória necessária para armazenar programa e dados

© André de Carvalho - ICMC/USP 8

8

Tempo de execução

- Como medir o tempo de execução de um algoritmo?
- Estudo experimental
 - Escrever o algoritmo em Python
 - Executar ele várias vezes, variando tamanho e composição da entrada
 - Plotar o tempo que ele leva para terminar
 - Python tem alguma função para extrair tempo?

© André de Carvalho - ICMC/USP 9

9

Tempo de execução

- Usar função time do módulo time
 - Retorna número de (frações) segundos de uso da CPU entre dois instantes de tempo

```
from time import time
start-time = time() # registra tempo inicial
run algorithm
end-time = time() # registra tempo final
elapsed = end-time - start-time
```

© André de Carvalho - ICMC/USP 10

10

Tempo de execução

- Problema em usar a função time
 - Outros processos podem estar usando a CPU no mesmo período (wall clock)
 - Alternativa: retornar ciclos de CPU usados pelo algoritmo
 - Com a função clock do módulo time
 - Problema: também pode ser inconsistente para o mesmo algoritmo com a mesma entrada
 - Depende do hardware e do software básico
 - Alternativa: usar módulo timeit

© André de Carvalho - ICMC/USP 11

11

Tempo de execução

Para uma boa análise estatística, usar um número suficiente de entradas representativas

© André de Carvalho - ICMC/USP 12

12

Tempo de execução

- O tempo de execução de um algoritmo depende principalmente do tamanho da entrada
 - Maior a entrada, maior o tempo
 - O tempo de execução, T , é uma função do tamanho da entrada, n
 - $T(n) = f(n)$
 - Como definir $f(n)$ para um dado algoritmo?

© André de Carvalho - ICMC/USP 13

13

Tempo de execução

- Tempo de execução para entradas do mesmo tamanho pode variar
 - Depende da composição da entrada
 - Alternativa: tirar a média de várias execuções de entradas do mesmo tamanho
 - Problema:
 - Quantas entradas diferentes?
 - Geradas com qual distribuição de probabilidade?

© André de Carvalho - ICMC/USP 14

14

Exemplo

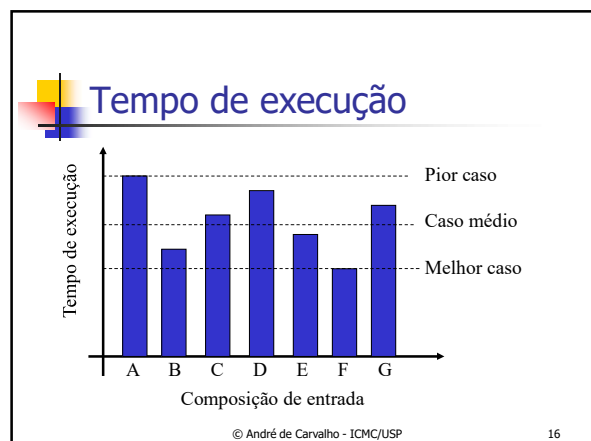
- Executar o algoritmo LinearSearch para:
 - LinearSearch (7, [1,3,7,10])
 - LinearSearch (0, [1,3,7,10,5,6,23,70,12,45,0,2])
 - LinearSearch (15, [1,3,7,10])

```
def LinearSearch (key, lista = []):
    for i in range (len(lista)):
        if (key == lista[i]):
            return (i)
    return (-1)
```

```
listInit = [1,3,7,10]
print(LinearSearch (7,listInit))
2
```

© André de Carvalho - ICMC/USP 15

15



16

Tempo de execução

- Como estimar o tempo de execução de um algoritmo?
 - Pelo menor tempo de execução (**melhor caso**)
 - Otimista: pode trazer surpresas desagradáveis
 - Pelo maior tempo de execução (**pior caso**)
 - Pessimista: pode trazer boas surpresas
 - Medida mais comum
 - Garante que o algoritmo é pelo menos tão bom quanto a análise indica

© André de Carvalho - ICMC/USP 17

17

Tempo de execução

- Como estimar o tempo de execução de um algoritmo?
 - Pelo tempo médio de execução
 - Prevê como o algoritmo se comporta em média para todas as possíveis composições
 - Difícil de medir

© André de Carvalho - ICMC/USP 18

18

Pior caso

- A análise do pior caso é muito mais fácil do que a análise média dos casos
 - Precisa apenas identificar a entrada do pior caso, muitas vezes simples
 - Normalmente leva a algoritmos melhores
 - Se um algoritmo é o melhor no pior caso, em geral é melhor na média
 - Crucial em algumas aplicações
 - Ex.: controle de tráfego aéreo

© André de Carvalho - ICMC/USP 19

19

Exemplo: ordenação

- Reordenar os elementos de uma lista, colocando-os em uma sequência definida
 - Dado o vetor

56	25	37	58	95	19	73	30
0	1	2	3	4	5	6	7

 Tamanho da lista n = 8
 - Ordenar elementos em ordem crescente

19	25	30	37	56	58	73	95
0	1	2	3	4	5	6	7

© André de Carvalho - ICMC/USP 20

20

Problemas de ordenação

- Algoritmos de ordenação
 - Existe uma grande variedade
 - Ordenação por seleção
 - Ordenação por inserção
 - Ordenação por combinação
 - Quicksort
 - Bubblesort
 - Desempenhos e filosofia diferentes

© André de Carvalho - ICMC/USP 21

21

Ordenação por seleção

- Nome original: *selection sort*
- Um dos algoritmos de ordenação mais simples
 - Começa no início da lista
 - Para cada posição da lista ocupada por um item X
 - Procura item Y que deve ocupar aquela posição (Y é o menor dos itens menores que X)
 - Troca as posições de X e Y

© André de Carvalho - ICMC/USP 22

22

Ordenação por seleção

```
def selecao (lista):
    for ini in range (0, len (lista)-1):
        fim = ini
        for i in range (ini + 1, len(lista)):
            if (lista[i] < lista[fim]):
                fim = i
        temp = lista[ini]
        lista[ini] = lista[fim]
        lista[fim] = temp
    return lista
```

lista1 = [1,5,3,7]
print(selecao (lista1))
1,3,5,7

© André de Carvalho - ICMC/USP 23

23

Ordenação por seleção

```
def selecao2 (lista):
    for ini in range (0, len (lista)-1):
        fim = ini
        for i in range (ini + 1, len(lista)):
            if (lista[i] < lista[fim]):
                lista[fim], lista[ini] = lista[ini], lista[fim]
    return lista
```

lista1 = [1,5,3,7]
print(selecao2 (lista1))
1,3,5,7

© André de Carvalho - ICMC/USP 24

24

Ordenação por seleção

- Exemplo:

© André de Carvalho - ICMC/USP 25

25

Exercício

- Qual o tempo de execução das duas versões da ordenação por seleção
 - Pior caso, caso médio e melhor caso
 - Para 100 listas de números inteiros geradas aleatoriamente
 - Para $n = 100, 1000$ e 10000
 - Inclua depois um terceiro algoritmo, criado pelo grupo e faça os mesmos testes

© André de Carvalho - ICMC/USP 26

26

Exemplo ordenação por seleção

N	Tempo médio
10	0.12 mseg
20	0.39 mseg.
40	1.46 mseg.
100	8.72 mseg.
200	33.33 mseg.
400	135.42 mseg.
1000	841.67 mseg.
2000	3.35 seg.
4000	13.42 seg.
10000	83.90 seg.

© André de Carvalho - ICMC/USP 27

27

Medidas de desempenho

- Tempo não aumenta linearmente com o aumento no número de elementos
 - Cada vez que o número de elementos dobra, o tempo de processamento quadruplica
 - Polinomial
 - A multiplicação do tamanho por 10, multiplica o tempo por 100
 - 100.000 elementos → duas horas e meia
 - 1.000.000 elementos → 10 dias
- Algoritmo pouco eficiente para listas grandes

© André de Carvalho - ICMC/USP 28

28

Medidas de desempenho

- Análise do desempenho
 - Número de passos para a ordenação da lista
 - Primeiro elemento, N passos
 - Segundo elemento, N-1 passos
 - Terceiro elemento, N-2 passos
 - ...
 - N-ésimo elemento, 1 passo
 - No total, quantos passos?

© André de Carvalho - ICMC/USP 29

29

Medidas de desempenho

- Análise do desempenho
 - Número de passos para a ordenação do vetor
 - Primeiro elemento, N passos
 - Segundo elemento, N-1 passos
 - Terceiro elemento, N-2 passos

$$\text{Num_passos} = N + N-1 + N-2 + \dots + 3 + 2 + 1$$

$$\text{Num_passos} = \frac{N(N+1)}{2} = \frac{N^2 + N}{2}$$

© André de Carvalho - ICMC/USP 30

30

Medidas de desempenho

n	Número de passos
10	55
20	210
40	820
100	5050
200	20.100
400	80.200
1000	500.500
2000	2.001.000
4000	8.002.000
10000	50.005.000

© André de Carvalho - ICMC/USP 31

31

Medidas de desempenho

Tempo de execução de cada passo = $\frac{83.90 \text{ segundos}}{50.005.000}$
(ex. 10.000 passos)

Tempo total = $0.167 \text{ mseg.} \times \frac{N^2 + N}{2}$

© André de Carvalho - ICMC/USP 32

32

- ## Notação assintótica
- Análise detalhada pode resultar em informações em excesso
 - Medidas qualitativas podem ser mais úteis
 - Baixa eficiência da ordenação por seleção para n grande
 - Tem pouco a ver com a capacidade de processamento de uma máquina específica
 - O problema é mais simples e básico:
 - Tempo de processamento cresce mais rapidamente que número de elementos da lista
- © André de Carvalho - ICMC/USP 33

33

- ## Notação assintótica
- Informações mais úteis são as que ajudam a entender
 - Como o desempenho é afetado por mudanças no tamanho do problema (n)
 - Complexidade computacional
 - Relaciona o desempenho de um algoritmo com o valor de n
 - Como o aumento de n afeta o desempenho do algoritmo
 - Pode ser
 - Tempo de processamento (medida mais usada)
 - Quantidade de memória necessária
- © André de Carvalho - ICMC/USP 34

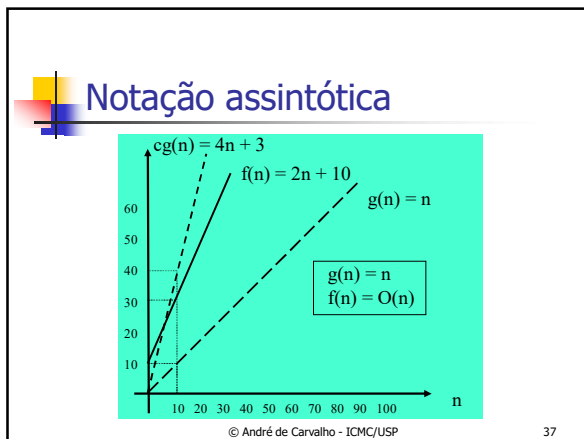
34

- ## Notação assintótica
- Notação Big O (grande O)
 - Usada para descrever classes de complexidade
 - Função simples que usa o tamanho do problema
 - Ex.: $O(n)$ /* Big O de n, da ordem de n */
 - Utilizado para estimar eficiência de forma qualitativa
 - Ideal para expressar a complexidade computacional de um algoritmo
 - Fornece uma medida de como uma mudança no valor de n afeta o desempenho de um algoritmo
- © André de Carvalho - ICMC/USP 35

35

- ## Notação assintótica
- Definição formal de Big O
 - Dadas 2 funções $f(n)$ e $g(n)$, dizemos que $f(n)$ é $O(g(n))$ se existe uma constante real $c > 0$ e uma constante inteira $n_0 \geq 1$, tal que $f(n) \leq c \cdot g(n)$ para todo inteiro $n \geq n_0$
 - Tradução:
 - Quando o tamanho n é grande o suficiente, haverá uma função $c \cdot g(n)$ maior que $f(n)$
 - $c \cdot g(n)$ é um limite superior (upper bound)
- © André de Carvalho - ICMC/USP 36

36



37

Classes de complexidade

- Algoritmos polinomiais
 - São executados em um tempo N^k para alguma constante K
 - Algoritmos de complexidade constante, linear, quadrática, cúbica, etc.
- Os piores algoritmos são os de complexidade exponencial
 - Ex.: complexidade $O(2^N)$

© André de Carvalho - ICMC/USP

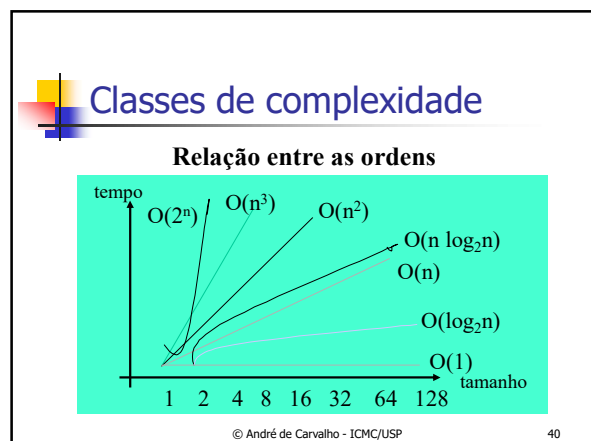
38

Classes de complexidade

- Problemas tratáveis
 - Existem algoritmos que rodam em tempo polinomial
- Problemas intratáveis
 - Não existem algoritmos que rodam em tempo polinomial
 - Algoritmos existentes para vários problemas comercialmente importantes são exponenciais
 - Ex. Problema do caixeiro viajante

© André de Carvalho - ICMC/USP

39



40

Classes de complexidade

Classe	Notação	Exemplo
constante	$O(1)$	Retornar o primeiro elemento de um vetor
logarítmica	$O(\log n)$	Busca binária em vetor ordenado
linear	$O(n)$	Busca linear em um vetor
$N \log N$	$O(n \log n)$	Ordenação por combinação
quadrático	$O(n^2)$	Ordenação por seleção
cúbico	$O(n^3)$	Algoritmos convencionais para multiplicação de vetores
exponencial	$O(2^n)$	Torre de Hanói

© André de Carvalho - ICMC/USP

41

Classes de complexidade

- Quanto maior o crescimento do tempo, mais complexo é o algoritmo
- Algoritmos cujo tempo independe do número de exemplos tem complexidade constante
 - Algoritmo constante tem complexidade $O(1)$
- Quando o tempo aumenta linearmente com o número de exemplos, a complexidade é linear
 - Algoritmos linear tem complexidade $O(n)$
 - Quando o tempo aumenta mais rapidamente, complexidade é não linear

© André de Carvalho - ICMC/USP

42

Exercício

- Algoritmo A usa $10n \log n$ operações, enquanto algoritmo B usa n^2 operações
 - Determinar o valor n_0 tal que A é melhor que B for $n \geq n_0$
- Mostre que 2^{n+1} é $O(2^n)$

© André de Carvalho - ICMC/USP 43

43

Notação assintótica

- Simplificações padrão da notação Big O
 - Big O não é uma medida quantitativa
 - Simplificar fórmula entre parênteses para medir apenas a qualidade do algoritmo
 - Regra da simplificação
 - Eliminar termos cuja contribuição deixe de ser significativa quando n se torna grande
 - Eliminar quaisquer fatores que sejam constantes

© André de Carvalho - ICMC/USP 44

44

Complexidade computacional

- Complexidade computacional da ordenação por seleção

Tempo de execução $\cong \frac{N^2 + N}{2} \rightarrow O(\frac{N^2 + N}{2})$ Expressão complicada

$$\frac{N^2 + N}{2} = \frac{N^2}{2} + \frac{N}{2} \rightarrow \frac{N^2}{2} \rightarrow N^2$$

Constante
Insignificante quando N se torna grande

Complexidade: $O(n^2)$

© André de Carvalho - ICMC/USP 45

45

Exercício

- Previsão da complexidade computacional a partir do código

Definir a complexidade computacional da função:

```
def media (lista):
    total = 0
    for i in range (0, n):
        total += lista[i]
    return total/n
```

© André de Carvalho - ICMC/USP 46

46


Complexidade computacional

- Complexidade da função media
 - Partes do código são executados um número constante de vezes
 - Não dependem do tamanho do problema
 - $O(1)$ (**tempo constante**)
 - Partes do código são executadas N vezes
 - Fazem parte de *loops*
 - Tempo de execução proporcional ao tamanho do *loop*
 - $O(n)$ (**tempo linear**)

© André de Carvalho - ICMC/USP 47

47

Questions



© André de Carvalho - ICMC/USP 48

48