Itaú Unibanco

Itaú

Programa de formação

ITAú analytics

Módulo I – Fundamentos Computacionais
Sessão 4 - Aula 5 – Análise de Algoritmos (II)

Prof. Dr. Luiz Alberto Vieira Dias
Prof. Dr. Lineu Mialaret

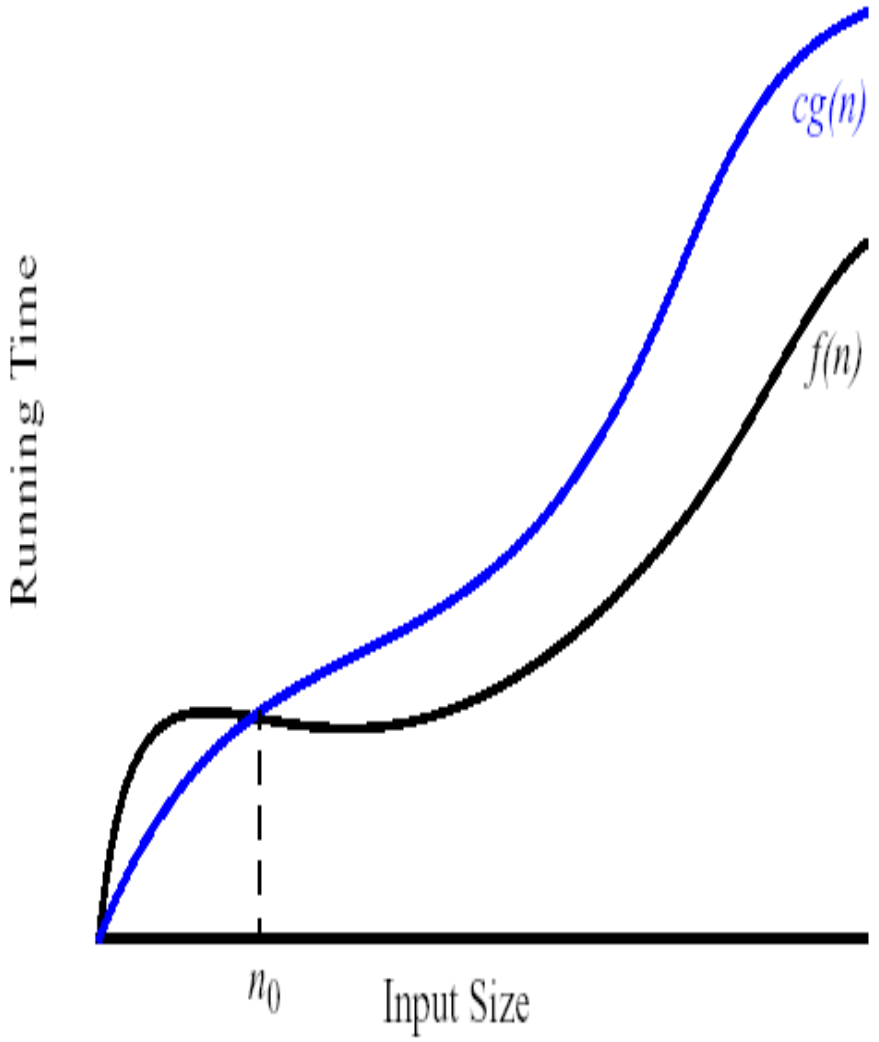
Análise de Algoritmos
Análise Assintótica

- Na Análise de Algoritmos foca-se na taxa de crescimento do tempo de execução como uma função do tamanho da entrada ***n***
  - Frequentemente basta saber que o tempo de execução de um algoritmo cresce proporcionalmente a ***n***
- A análise de algoritmos é realizada utilizando-se de formalismo matemático para caracterizá-la
  - Os tempos de execução dos algoritmos são caracterizados por meio de funções que mapeiam o tamanho da entrada ***n*** para valores que correspondem ao principal termo que determina a taxa de crescimento em termos de ***n***
  - Cada algoritmo contém um número de operações primitivas realizadas que são estimadas e independentes de hardware e software

Análise de Algoritmos
Notação Big-Oh

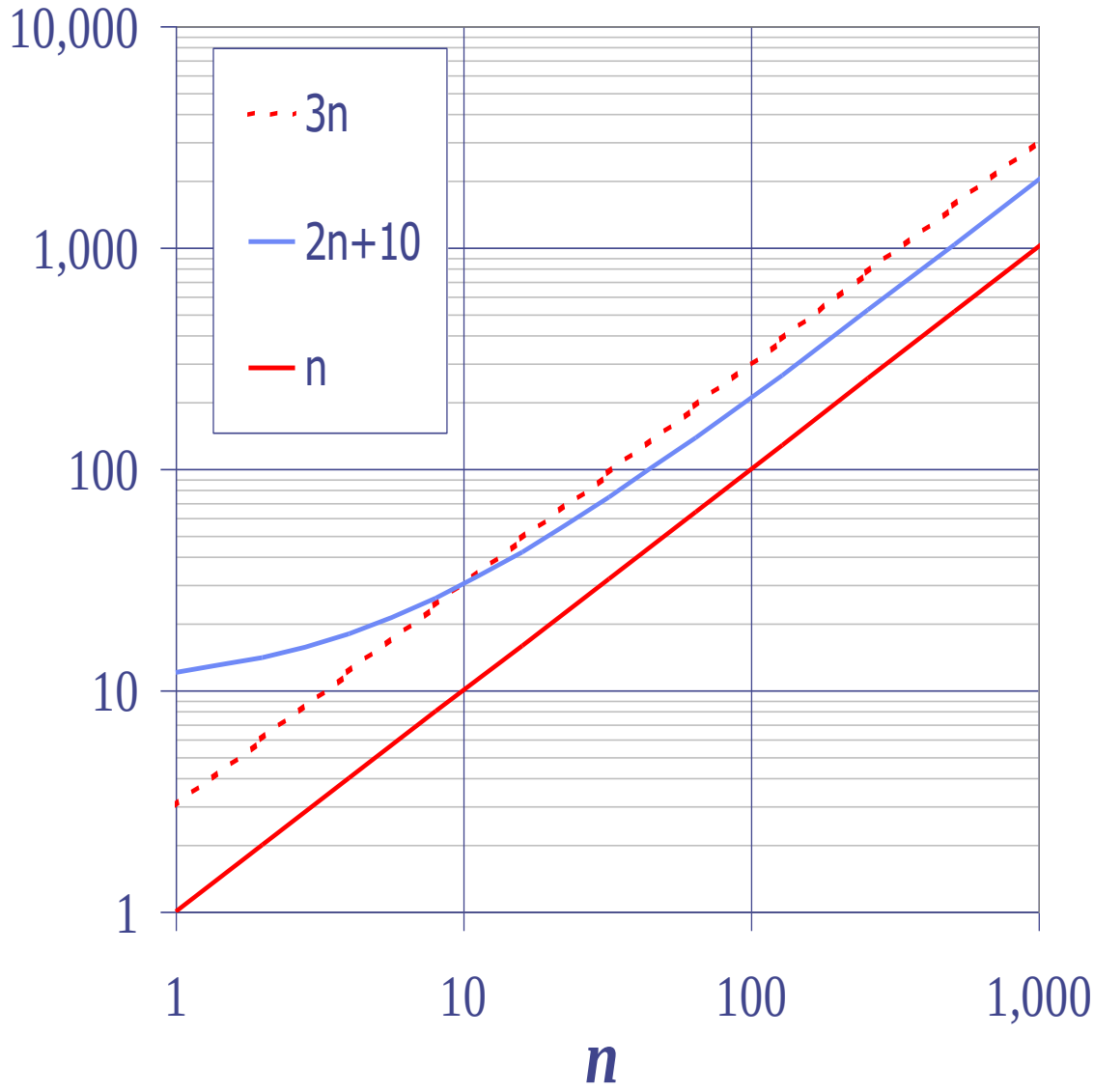
- Sejam dadas as funções **f(*n*)** e **g(*n*)**, que mapeiam inteiros positivos para números reais positivos
- Diz-se que **f(*n*)** é **O(g(*n*))** se existe uma constante real **c** > 0 e uma constante inteira $n_0 \geq 1$ tal que

  f(*n*) ≤ cg(*n*) para $n \geq n_0$



Análise de Algoritmos
Notação Big-Oh (cont.)

- Exemplo: $2n + 10$ é $O(n)$
  - $f(n) = 2n + 10$
  - $g(n) = n$
  - $f(n) \leq cg(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
- Escolhendo-se $c = 3$ e $n_0 = 10$



Análise de Algoritmos
Notação Big-Oh (cont.)

- $7n - 2$

  $7n - 2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

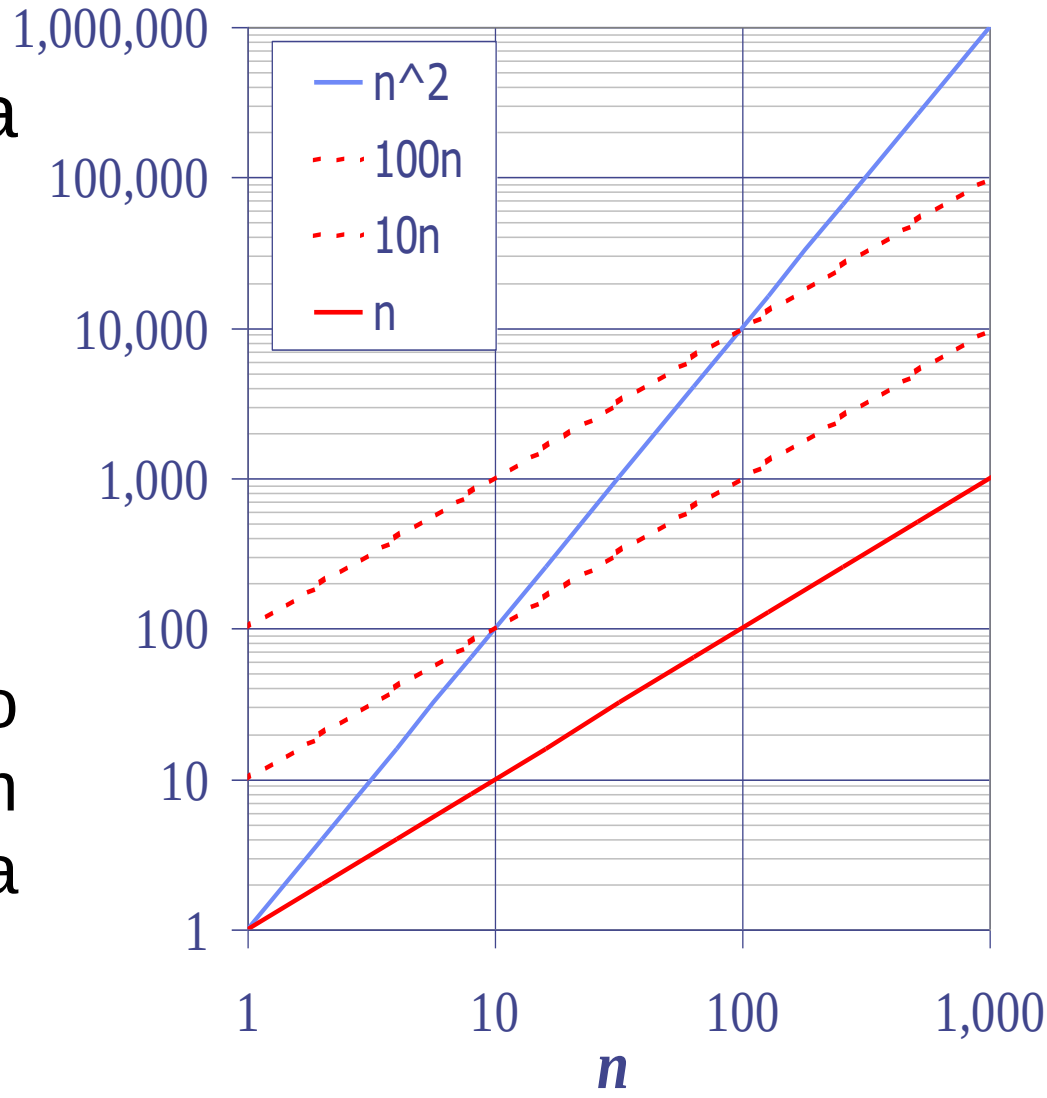
  this is true for $c = 8$ and $n_0 = 2$

Análise de Algoritmos
Notação Big-Oh (cont.)

- Exercício: Mostrar que a função n^2 não é $O(n)$

Análise de Algoritmos
Notação Big-Oh (cont.)

- Exemplo: Mostrar que a função n^2 não é $O(n)$
  - $f(n)$ n^2
  - $g(n) = n$
  - $n^2 \leq cn$
  - $n \leq c$
  - A inequação acima não pode ser satisfeita em razão de c ser uma constante



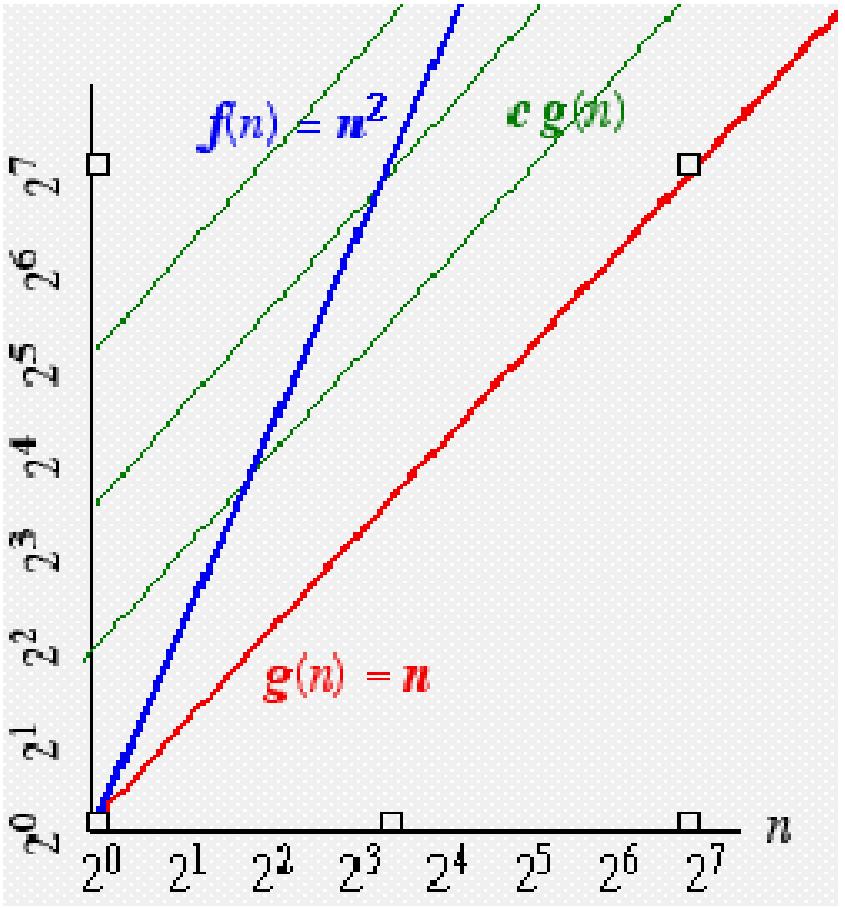
Análise de Algoritmos
Notação Assintótica (cont.)

On the other hand...

n^2 is not O(n) because there is no c and n_0 such that:

$$n^2 \leq cn \text{ for } n \geq n_0$$

(As the graph to the right illustrates, no matter how large a c is chosen there is an n big enough that $n^2 > cn$)



Análise de Algoritmos
Notação Assintótica (cont.)

- Meta da notação assintótica: simplificar a análise dispensando as informações desnecessárias
  - como "arredondamento": $1.000.001 \approx 1.000.000$
    $3n^2 \approx n^2$
- A notação "Big-Oh"
  - Dadas as funções f(n) e g(n), diz-se que
    f(n) é O(g(n)) se e somente se
    f(n) $\leq$ c g(n) para $n \geq n_0$
  - c e n_0 são constantes, f(n) e g(n) são funções sobre inteiros não negativos
- Outras notações:
  - Ω(f(n)): Big Omega - limite inferior
  - Θ(f(n)): Big Theta - igualdade

Análise de Algoritmos
Notação Assintótica (cont.)

- Embora **$7n - 3$** seja ***$O(n^5)$***, é esperado que tal aproximação seja de tão baixa ordem quanto possível
- A regra é desprezar os termos de mais baixa ordem e os fatores constantes
  - **$7n - 3$ é *$O(n)$***
  - **$8n^2\log n + 5n^2 + n$ é *$O(n^2 \log n)$***

- Classes especiais de algoritmos:
  - **logarítmico:** ***$O(\log n)$***
  - **linear :** ***$O(n)$***
  - **quadrático:** ***$O(n^2)$***
  - **polinomial:** ***$O(n^k)$*, $k \geq 1$**
  - **exponencial:** ***$O(a^n)$*, $n > 1$**

Análise de Algoritmos
Notação Assintótica (cont.)

- Usa-se a notação ***O*** para expressar o número de operações primitivas executadas como uma função do tamanho da entrada
- Por exemplo, pode-se dizer que o algoritmo *arrayMax* roda em ***O(n)*** unidades de tempo
- Comparando o tempo de execução assintótico:
  - um algoritmo que roda em ***O(n)*** é melhor do que um que roda em $O(n^2)$
  - semelhantemente, ***O(logn)*** é melhor ***O(n)***
  - hierarquia das funções:

    $\log n \ll n \ll n\log n \ll n^2 \ll n^3 \ll 2^n \ll n!$
- Cuidado!

  Tomar cuidado ao encontrar fatores constantes muito grandes. Um algoritmo que roda no tempo $1.000.000n$ é ainda $O(n)$ mas pode ser menos eficiente em seu conjunto de dados do que um algoritmo rodando no tempo $2n^2$, que é $O(n^2)$

Análise de Algoritmos
Notação Assintótica (cont.)

	Tipos de função f(*n*)						
input *n*	**1**	**log*n***	***n***	***n*log*n***	n^2	n^3	2^n
1	1	0	1	0	1	1	2
10	1	3,32	10	33	100	1000	1024
100	1	6,64	100	664	10000	1000000	$1,268 \times 10^{30}$
1000	1	9,97	1000	9970	1000000	10^9	$1,072 \times 10^{301}$

Comparação de ordem de grandeza para várias funções

Análise de Algoritmos
Notação Assintótica (cont.)

EFICIÊNCIA	Notação O	INTERAÇÕES	TEMPO ESTIMADO
Logarítmica	**O(log(n))**	14	microssegundos
Linear	**O(n)**	10000	0,1 segundo
Log. Linear	**O(n(log(n))**	14000	2 segundos
Quadrática	**$O(n^2)$**	10000^2	15-20 minutos
Polinomial	**$O(n^k)$**	10000^k	Horas
Exponencial	**$O(e^n)$**	2^{10000}	Intratável
Fatorial	**O(n!)**	10000!	Intratável

Medidas de Eficiência, assumindo velocidade de 1 microssegundo e 10 instruções num loop

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade

- **Função de Custo** ou **Função de Complexidade**
  - $f(n) =$ medida de custo necessário para executar um algoritmo para um problema de tamanho n
  - Se $f(n)$ é uma medida da quantidade de tempo necessário para executar um algoritmo em um problema de tamanho n, então f é chamada *função de complexidade de tempo de algoritmo*
  - Se $f(n)$ é uma medida da quantidade de memória necessária para executar um algoritmo em um problema de tamanho n, então f é chamada *função de complexidade de espaço de algoritmo*

- Observação: tempo não é tempo!
  - É importante ressaltar que a complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Constante

- $f(n) = O(1)$
  - O uso do algoritmo independe do tamanho de n
  - As instruções do algoritmo são executadas um número fixo de vezes
- O que significa um algoritmo ser $O(2)$ ou $O(5)$?

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Logarítmica

- $f(n) = O(\log n)$
  - Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores
  - Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande
- Supondo que a base do logaritmo seja 2:
  - Para n = 1000, log2 ≈ 10
  - Para n = 1000000, log2 ≈ 20
- Exemplo:
  - Algoritmo de pesquisa binária

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Linear

- $f(n) = O(n)$
  - Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada
  - Esta é a melhor situação possível para um algoritmo que tem que processar/produzir *n* elementos de entrada/saída
  - Cada vez que *n* dobra de tamanho, o tempo de execução também dobra

- Exemplo:
  - Algoritmo de pesquisa seqüencial

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Linear Logarítmica

- $f(n) = O(n \log n)$
  - Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções
  - Caso típico dos algoritmos baseados no paradigma divisão-e-conquista
- Supondo que a base do logaritmo seja 2:
  - Para n = 1000000, log2 ≈ 20000000
  - Para n = 2000000, log2 ≈ 42000000
- Exemplo:
  - Algoritmo de ordenação MergeSort

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Quadrática

- $f(n) = O(n^2)$
  - Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um laço dentro de outro
  - Para n = 1000, o número de operações é da ordem de 1000000
  - Sempre que n dobra o tempo de execução é multiplicado por 4
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos
- Exemplos:
  - Algoritmos de ordenação simples como seleção e inserção

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Cúbica

- $f(n) = O(n^3)$
  - Algoritmos desta ordem de complexidade geralmente são úteis apenas para resolver problemas relativamente pequenos
  - Para n = 100, o número de operações é da ordem de 1000000
  - Sempre que n dobra o tempo de execução é multiplicado por 8
  - Algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos
- Exemplo:
  - Algoritmo para multiplicação de matrizes

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Expoencial

- $f(n) = O(2^n)$
  - Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático
  - Eles ocorrem na solução de problemas quando se usa a força bruta para resolvê-los
  - Para $n = 20$, o tempo de execução é cerca de 1000000
  - Sempre que n dobra o tempo de execução fica elevado ao quadrado

- Exemplo:
  - Algoritmo do Caixeiro Viajante

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Expoencial

- $f(n) = O(n!)$
  - Um algoritmo de complexidade $O(n!)$ é dito ter complexidade exponencial, apesar de O(n!) ter comportamento muito pior do que $O(2^n)$
  - Geralmente ocorrem quando se usa força bruta na solução do problema.

- Considerando:
  - n = 20, tem-se que 20! = 2432902008176640000, um número com 19 dígitos
  - n = 40 tem-se um número com 48 dígitos

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Constante

- Exercício: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?

Análise de Algoritmos
Notação Assintótica – Funções de Complexidade Constante

- Exercício: um programa leva $100n$ unidades de tempo para ser executado e outro leva $2n^2$. Qual dos dois programas é melhor?
- Resposta: Depende do tamanho do problema.
  - Para $n < 50$, o programa com tempo $2n^2$ é melhor do que o que possui tempo $100n$
  - Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é $O(n^2)$
  - Entretanto, quando n cresce, o programa com tempo de execução $O(n^2)$ leva muito mais tempo que o programa $O(n)$

Análise de Algoritmos
Pseudocódigo

- É frequente se solicitar aos programadores que descrevam algoritmos de uma forma que seja especialmente adequada a outros programadores
- Essas descrições não são programas de computador, mas são mais estruturadas do que linguagem coloquial ou natural (simples prosa)
- Pseudocódigo constitui-se nas descrições de alto nível, combinando linguagem natural e estruturas familiares de uma linguagem de programação, numa forma clara e informativa, para descrever algoritmos
- Deve ser legível a um leitor humano
- Deve comunicar ideias de alto nível e não detalhes de implementação de baixo nível

Análise de Algoritmos
Pseudocódigo (cont.)

- Pseudocódigo é uma mistura de linguagem natural e estruturas de programação de alto nível usada para descrever as ideias principais da implementação genérica de uma estrutura de dados ou algoritmo
- No entanto não existe uma definição precisa da linguagem de pseudocódigo por causa do uso de linguagem natural
- Ao mesmo tempo, para aumentar sua legibilidade e clareza, o pseudocódigo mistura linguagem natural com construções-padrão de linguagens de programação, como apresentado a seguir:

Análise de Algoritmos
Pseudocódigo – Estruturas Básicas em Java

- Expressions: use standard mathematical symbols to describe numeric and boolean expressions
  - use ← for assignment ("=" in Java)
  - use = for the equality relationship ("==" in Java)
- Method Declarations:
  - **Algorithm** name(*param1*, *param2*)
- Programming Constructs:
  - decision structures: **if** ... **then** ... **[else** ... **]**
  - while-loops: **while** ... **do**
  - repeat-loops: **repeat** ... **until** ...
  - for-loop: **for** ... **do**
  - array indexing: **A[*i*]**
- Methods:
  - calls: object method(args)
  - returns: **return** value

Análise de Algoritmos
Pseudocódigo – Estruturas Básicas em Python

- Control flow
  - **if** . . . **then** . . . [**else**] . . .
  - **while** . . . **do** . . . . .
  - **for** . . . **do** . . .
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***. . .])

  Input . . .

  Output . . .

- Method call

  var.method (***arg*** [, ***arg***. . .])
- Return value

  return ***expression***
- Expressions:

  ← Assignment

  = Equality testing

  n^2 Superscripts and other mathematical formatting allowed

Análise de Algoritmos
Pseudocódigo - Exemplo

- O problema do elemento máximo de um arranjo ou *array* (ou lista) consiste simplesmente em descobrir qual é o maior elemento de um arranjo *A* que armazena *n* elementos inteiros
- Para resolver esse problema, pode-se usar o algoritmo chamado *arrayMax*, que percorre os elementos do *array* (lista) *A* usando um laço *for*

Análise de Algoritmos
Pseudocódigo - Exemplo (cont.)

- Exemplo de Pseudocódigo (Algoritmo *arrayMax*):

```
Algoritmo arrayMax(A,n):
Entrada: um array A contendo n ≥ 1 inteiros
Saída:   o maior inteiro no array A
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax <  A[i] then
      currentMax ← A[i]
return currentMax
```

Análise de Algoritmos
Pseudocódigo - Exemplo em Java

- Fragmento de código em Java (Algoritmo *arrayMax*):

```
public class ArrayMaxProgram
{      // programa de teste para um algoritmo que encontra
       // o maior elemento em um array
  static int arrayMax(int[] A, int n)
  {  // encontra o maior elemento em um array A de n
     // inteiros pesquisando as células de A enquanto
    // mantém a trilha do maior elemento encontrado
   int currentMax = A[0];    //executado uma vez
   for (int i=1; i < n; i++)//executado uma, duas, três
     vezes
     if (currentMax < A[i]) //executado n-1 vezes
        currentMax = A[i];// executado no máximo, n-1
     vezes
      return currentMax;      // executado uma vez }
```

Análise de Algoritmos
Pseudocódigo – Exemplo em Java (cont.)

- Fragmento de código em Java (Algoritmo *arrayMax*):

```
public static void main(String args[])
{// testando o método chamado qdo. o programa é executado
 int[] num = {10, 15, 3, 5, 56, 107, 22, 16, 85};
 int n = num.length;
 System.out.print("Array:");
 for (int i=0; i < n; i++)
     System.out.println(".");
 System.out.println("O maior elemento é " +
           arrayMax(num, n) + "."); }}
```

Análise de Algoritmos
Operações Primitivas

- Caso deseje-se analisar um algoritmo sem efetuar experimentos, deve-se seguir os seguintes passos:

1. Codifique o algoritmo em alguma linguagem de programação de alto nível (como a linguagem Java)
2. Compile o programa em alguma linguagem executável de baixo nível (como o bytecode da *Java Virtual Machine - JVM* ou do Python)
3. Determine para cada **instrução *i*** da linguagem de baixo nível, o **tempo t_i** necessário para executar a instrução
4. Determine para cada **instrução *i*** da linguagem de baixo nível, o número de vezes n_i que a **instrução *i*** é executada quando o algoritmo é executado
5. Some os produtos $n_i.t_i$ para todas as instruções. Isto fornece o tempo de execução do algoritmo

- Esta abordagem pode dar uma estimativa mais precisa do tempo de execução do algoritmo, mas é complicada de ser realizada

Análise de Algoritmos
Operações Primitivas (cont.)

- Essa abordagem anterior requer um conhecimento detalhado da linguagem de baixo nível gerada pela compilação de um programa e o ambiente no qual ele é executado
- Assim, ao invés desse tipo de abordagem, pode-se efetuar a análise diretamente sobre o código fonte ou sobre o pseudocódigo, usando-se o conceito de operações primitivas
- **Operações Primitivas** são as computações de alto nível que são independentes da linguagem de programação e podem ser identificadas no pseudocódigo

Análise de Algoritmos
Operações Primitivas (cont.)

- Pode-se definir um conjunto de operações primitivas de alto nível que são independentes da linguagem de programação utilizada, como por exemplo:
  - Atribuir um valor a uma variável
  - Chamar um método
  - Efetuar uma operação aritmética (exemplo: somar 2 inteiros)
  - Comparar dois números
  - Indexar um array
  - Seguir uma referência a objeto
  - Retornar de um método

Análise de Algoritmos
Operações Primitivas (cont.)

- Desta forma, inspecionando-se o pseudocódigo, pode-se contar o número de operações primitivas executadas por um algoritmo
- De forma mais específica, uma operação primitiva corresponde a uma instrução de baixo nível com um tempo de execução constante, mas que depende do ambiente de software e hardware

Análise de Algoritmos
Operações Primitivas (cont.)

- Em vez de tentar determinar o tempo de execução específico de cada operação primitiva, simplesmente **conta-se** quantas operações primitivas serão executadas e usa-se este número ***t*** como uma **estimativa** de alto nível do tempo de execução do algoritmo
- Essa contagem de operações está relacionada com o tempo de execução em um hardware e software específicos, pois cada operação corresponde a uma instrução realizada em tempo constante e existe um número fixo de operações primitivas

Análise de Algoritmos
Operações Primitivas (cont.)

- Nesta abordagem, assume-se implicitamente que os tempos de execução de operações primitivas diferentes serão similares
- Assim, o número ***t*** de operações primitivas que um algoritmo realiza será proporcional ao tempo de execução daquele algoritmo

Análise de Algoritmos
Operações Primitivas - Exemplo

Exemplo de Pseudo Código (Algoritmo *arrayMax*):

```
Algoritmo arrayMax(A, n):
Entrada: um array contendo n inteiros
Saída:   o máximo elemento em A
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Análise de Algoritmos
Operações Primitivas – Exemplo - Análise

- Análise do Algoritmo *arrayMax*:
  - Inicializar a variável ***currentMax*** para ***A[0]*** corresponde a duas operações primitivas (indexar um *array* e atribuir um valor para uma variável) e é executado somente uma vez no começo do algoritmo. Isto é, essa inicialização contribui com duas unidades para a conta do número de operações do algoritmo

```
Algoritmo arrayMax(A, n):
Entrada: um array contendo n inteiros
Saída:   o máximo elemento em A
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

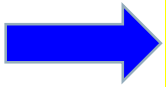
Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Análise do Algoritmo *arrayMax*:
  - No começo do loop ***for***, o contador ***i*** é inicializado em 1. Esta ação corresponde a executar uma operação primitiva (atribuir um valor para uma variável)

```
Algoritmo arrayMax(A, n):
Entrada: um array contendo n inteiros
Saída:   o máximo elemento em A
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

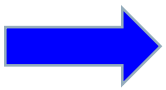
- Análise do Algoritmo *arrayMax*:
  - Antes de entrar no corpo do loop ***for***, a condição ***i < n*** é verificada. Esta ação corresponde a executar uma operação primitiva (comparar dois números). Uma vez que o contador ***i*** começa com 1 e é incrementado por 1 no final de cada iteração do loop, a comparação ***i < n*** é realizada ***n*** vezes. Assim, ela contribui com ***n*** unidades para a conta



```
Algoritmo arrayMax(A, n):
Entrada: um array contendo n inteiros
Saída:   o máximo elemento em A
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

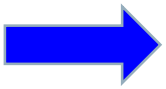
- Análise do Algoritmo *arrayMax*:
  - O corpo do loop ***for*** é executado ***n - 1*** vezes (para valores 1,2,3... ***n*** - 1 do contador). Em cada iteração, ***A[i]*** é comparado com ***currentMax*** (duas operações primitivas, indexação e comparação), ***A[currentMax]*** é possivelmente atribuída a ***currentMax*** (duas operações primitivas, indexação e atribuição) e o contador ***i*** é incrementado (duas operações primitivas, somatório e atribuição). Por esta razão, em cada iteração do loop, ou quatro ou seis operações primitivas são realizadas, dependendo se ***A[i]*** $\leq$ ***currentMax*** ou ***A[i]*** $>$ ***currentMax***. Portanto, o corpo do loop contribui entre ***4(n-1)*** e ***6(n-1)*** unidades para a conta



```
if currentMax < A[i] then
    currentMax ← A[i]
```

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Análise do Algoritmo *arrayMax*:
  - Retornar o valor da variável ***currentMax*** corresponde a uma operação primitiva e é executado uma só vez



```
Algoritmo arrayMax(A, n):
Entrada: um array contendo n inteiros
Saída:   o máximo elemento em A
currentMax ← A[0]
for i ← 1 to n-1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Resumindo:
  - o número de operações primitivas ***t*(*n*)** executadas pelo algoritmo **arrayMax** é
    - no mínimo

      $$2 + 1 + n + 4(n - 1) + 1 = 5n$$

      e no máximo

      $$2 + 1 + n + 6(n - 1) + 1 = 7n - 2$$
  - o melhor caso **(*t*(*n*) = *5n*)** ocorre quando ***A*[0]** é o elemento máximo, de maneira que a variável ***currentMax*** nunca sofre atribuição
  - o pior caso **(*t*(*n*) = *7n - 2*)** ocorre quando os elementos estão em ordem crescente, de modo que a variável ***currentMax*** sempre sofre atribuição em cada iteração do loop ***for***

Análise de Algoritmos
Operações Primitivas – Exemplo - Análise (cont.)

- Inspecionando-se o pseudocódigo, pode-se determinar o número máximo de operações primitivas executadas pelo algoritmo como função do tamanho da entrada

Algorithm ***arrayMax***(A, n)	# operations
currentMax $\leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2 + n$
if $A[i] >$ ***currentMax*** **then**	$2(n - 1)$
currentMax $\leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return ***currentMax***	1
Total	$7n - 1$

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Inspecionando-se o pseudocódigo do algoritmo **find_max** (em Python), pode-se determinar o número máximo de operações primitivas executadas pelo algoritmo como uma função do tamanho da entrada

```
1  def find_max(data):
2    """Return the maximum element from a nonempty Python list."""
3    biggest = data[0]                  # The initial value to beat
4    for val in data:                   # For each value:
5      if val > biggest                 # if it is greater than the best so far,
6        biggest = val                  # we have found a new best (so far)
7    return biggest                     # When loop ends, biggest is the max
```

- Passo 1: 2 ops, 3: 2 ops, 4: 2n ops, 5: 2n ops, 6: 0 até n ops, 7: 1 op

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

```
1.  def find_max(data): # ->2 ops
2.    """Return the maximum element from a nonempty Python list."""
3.    biggest = data[0]   # The initial value to beat -> 2 ops
4.    for val in data:      # For each value: -> 2n ops
5.      if val > biggest    # if it is greater than the best so far, -> 2n ops
6.        biggest = val     # we have found a new best (so far) ->0…n ops
7.    return biggest        # When loop ends, biggest is the max -> 1 op
```

(4n + 5) ops

a

(5n + 5) ops

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- O algoritmo **find_max** executa $5n + 5$ operações primitivas no pior caso (*worst case*) e $4n + 5$ no melhor caso (*best case*)
- Definindo-se
  - a = Tempo da operação primitiva mais rápida
  - b = Tempo da operação primitiva mais longa
- Seja $T(n)$ o tempo de pior caso de **find_max**
- Então tem-se que

$$a\,(4n + 5) \leq T(n) \leq b(5n + 5)$$

- Logo, o tempo de execução $T(n)$ é limitado por duas funções lineares

Análise de Algoritmos
Operações Primitivas – Exemplo - Análise (cont.)

- Mudando-se o ambiente de hardware e software
  - Afeta a função $T(n)$ por um fator constante, mas
  - Não altera a taxa de crescimento de $T(n)$
- A taxa de crescimento linear do tempo de execução $T(n)$ é uma propriedade intrínseca do algoritmo **find_max**

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Fragmento de código em Python (Algoritmo *find_max*):

```
def find_max(data):
"""Return the maximum element from a Python list."""
biggest = data[0]     # The initial value to beat
for val in data:      # For each value:
  if val > biggest:   # if it is greater than the
                        best so far
    biggest = val     # we have found a new best
                        (so far)
return biggest
```

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Fragmento de código em Python (Algoritmo *find_max*):

```
def find_max(data):    # 2 operações
"""Return the maximum element from a Python list."""
biggest = data[0]      # 2 operações
for val in data:       # 2n operações
  if val > biggest:    # 2n operações
     biggest = val     # 0 a n operações
return biggest         # 1 operação
```

Análise de Algoritmos
Operações Primitivas – Exemplo – Análise (cont.)

- Fragmento de código em Python (Algoritmo *find_max*):

```
def find_max(data):    # 2 operações
"""Return the maximum element from a Python list."""
biggest = data[0]      # 2 operações
for val in data:       # 2n operações
  if val > biggest:    # 2n operações
     biggest = val     # 0 a n operações
return biggest         # 1 operação
lista = (10, 15, 3, 5, 56, 107, 22, 16, 85)
print(find_max(lista))
```

```
========== RESTART: D:/Python36/Teste/find_max.py ==========
107
>>>
```

Análise de Algoritmos
Avaliação da Abordagem de Operações Primitivas

- Teve-se que entrar num elevado grau de detalhe para analisar o tempo de execução de um algoritmo simples como *arrayMax*. Essa análise levanta uma série de questões:
  - Esse nível de detalhe é mesmo necessário?
  - Qual a importância de se determinar exatamente o número de operações primitivas realizadas por um algoritmo?
  - Qual o grau de cuidado que se deve ter ao definir o conjunto de operações primitivas realizadas por um algoritmo?

Análise de Algoritmos
Avaliação da Abordagem de Operações Primitivas (cont.)

- Teve-se que entrar num elevado grau de detalhe para analisar o tempo de execução de um algoritmo simples como *arrayMax*. Essa análise levanta uma série de questões:
  - Quantas operações primitivas ocorrem na atribuição $y = a*x + b$?
  - Na atribuição anterior pode se argumentar que são duas operações aritméticas e uma atribuição, mas pode-se não estar considerando atribuições de resultados intermediários a variáveis temporárias, por exemplo

Análise de Algoritmos
Avaliação da Abordagem de Operações Primitivas (cont.)

- Em geral, cada passo numa descrição em pseudocódigo e cada comando numa linguagem de alto nível correspondem a um pequeno número de operações primitivas que não dependem do tamanho da entrada
- Dessa forma, pode-se realizar uma análise simplificada que estima o número de operações realizadas, exceto por um fator constante, simplesmente contando os passos do pseudocódigo ou os comandos da linguagem de alto nível usada
- No caso do algoritmo ***arrayMax***, essa análise simplificada indica que entre $\mathbf{5}\boldsymbol{n}$ e $\mathbf{7}\boldsymbol{n} - \mathbf{2}$ passos são executados para uma entrada de tamanho $\boldsymbol{n}$

Análise de Algoritmos
Avaliação da Abordagem de Operações Primitivas (cont.)

- Na análise de algoritmos, é importante concentrar-se na taxa de crescimento do tempo de execução como uma função do tamanho da entrada ***n***, obtendo-se um quadro geral do comportamento, em vez de se concentrar em detalhes gerais
- Frequentemente, basta saber apenas que o tempo de execução de um algoritmo do tipo ***arrayMax*** já apresentado cresce proporcionalmente a ***n***, com o verdadeiro tempo de execução sendo ***n*** vezes algum pequeno fator constante que depende do ambiente de hardware e software e que vai variar numa faixa de valores, dependendo da entrada específica

Análise de Algoritmos
Avaliação da Abordagem de Operações Primitivas (cont.)

- A abordagem empregada para analisar estruturas de dados e algoritmos utiliza uma notação matemática para funções quer têm a vantagem de desconsiderar fatores constantes
- Dessa forma, caracteriza-se o tempo de execução, e outras medidas de aferição da qualidade, como por exemplo, a quantidade de memória usada, em algoritmos e estruturas de dados, usando-se funções que mapeiam números inteiros em números reais, de uma forma que concentra a atenção no comportamento geral do tempo de execução e da memória exigida, por exemplo, sem um elevado grau de detalhe