Projeto e Análise de Algoritmos
Conceitos Básicos

Nelson Cruz Sampaio Neto
nelsonneto@ufpa.br

Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Programa de Pós-Graduação em Ciência da Computação

19 de agosto de 2019

Algoritmos

O que é um algoritmo?

É possível descrever um algoritmo de várias maneiras:

- Usando linguagem comum.
- Usando uma linguagem de programação de alto nível: C, Python, Java, entre outras.
- Implementando-o em linguagem de máquina diretamente executável em hardware.
- Por meio de um pseudocódigo.

Usaremos essencialmente pseudocódigo nesta disciplina!

Exemplo de pseudocódigo

Algoritmo ORDENA-POR-INSERÇÃO: Ordena de forma crescente um vetor $A[1 \ldots n]$.

```
ORDENA-POR-INSERÇÃO(A, n)
  para j ← 2 até n faça
      chave ← A[j]
      ▷ Insere A[j] no subvetor ordenado A[1 . . . j−1]
      i ← j − 1
      enquanto i ≥ 1  e A[i] > chave faça
          A[i + 1] ← A[i]
          i ← i − 1
      A[i + 1] ← chave
```

Análise de algoritmos

O que é importante analisar?

- **Finitude**: O algoritmo para?
- **Corretude**: O algoritmo faz o que promete?
- **Eficiência ou complexidade no tempo**: Calcula-se o tempo de execução do algoritmo (i.e. análise empírica), ou quantas instruções são executadas em função do tamanho da entrada (i.e. análise de complexidade).

Finitude

```
ORDENA-POR-INSERÇÃO (A, n)
1. para j = 2 até n faça
   ...
4.    i = j - 1
5.    enquanto i >= 1 e A[i] > chave faça
6.       ...
7.       i = i - 1
   ...
```

- No laço da linha 5 o valor de i diminui a cada iteração e o valor inicial é $i = j - 1 \geq 1$. Logo, a sua execução para em algum momento por causa do teste condicional $i \geq 1$.
- O laço na linha 1 evidentemente para porque o contador j atingirá o valor $n + 1$. Portanto, o algoritmo para.

Corretude

- Um algoritmo é correto se, para toda instância do problema, ele para e devolve uma resposta correta.

- Usamos o método de **loops invariantes** para nos ajudar a entender por que um algoritmo é correto. Como?
  - Inicialização: O invariante é verdadeiro antes da primeira iteração do laço (trivial, em geral).
  - Manutenção: Se o invariante for verdadeiro antes de uma iteração, ele permanecerá verdadeiro antes da próxima.
  - Término: Se o algoritmo para e o invariante vale no início da última iteração, então o algoritmo é correto.

- ORDENA-POR-INSERÇÃO: No começo de cada iteração do laço "para" (linha 1), o vetor $A[1...j-1]$ está ordenado.

Invariante de laço

- **Definição**: Um invariante é uma propriedade que relaciona as variáveis do algoritmo a cada execução completa do laço.
- O invariante deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo.
- Em geral, é mais difícil descobrir um invariante apropriado do que mostrar sua validade se ele for dado de "bandeja".

Eficiência

- Em geral, não basta saber que um dado algoritmo para e é correto. Se ele for muito lento, terá pouca utilidade.
- Queremos projetar algoritmos eficientes (ou "rápidos").
- Mas o que seria uma boa medida de eficiência?
- Existe um critério uniforme para comparar algoritmos?

Antes de responder essas perguntas, é preciso definir um modelo computacional de máquina.

Modelo computacional

- Simula máquinas convencionais de verdade.
- Estabelece quais são os recursos disponíveis, as instruções básicas e quanto elas custam (= **tempo**).
- Com o modelo, é possível estimar, através de uma análise matemática, o tempo que um algoritmo gasta em função do tamanho da entrada (= **análise de complexidade**).
- A análise de complexidade depende sempre do modelo computacional adotado.

Modelo computacional

- Salvo mencionado o contrário, usaremos sempre o Modelo Abstrato RAM (Random Access Machine):
  - Possui um único processador;
  - Executa instruções sequencialmente;
  - Tipos básicos são números inteiros e reais;
  - Executa operações aritméticas, comparações, movimentação de dados de tipo básico e fluxo de controle (chamada e retorno de rotina, teste if/else, etc.) **em tempo constante**;
  - ...
- Veja maiores detalhes do modelo RAM no Livro Texto.

- Vamos analisar a **eficiência** (ou complexidade no tempo) do algoritmo que encontra o maior elemento de um vetor A com n elementos.
- Basicamente, vamos encontrar o tempo de execução de cada passo do algoritmo em função do tamanho da entrada.

```
MAXIMO (A, n)                     Custo    #execuções
1. max = A[1]                      c1
2. para i = 2 até n faça           c2
3.   se max < A[i]                 c3
4.     então max = A[i]            c4
5. escreva ("máximo: " max)        c5
```

- Vamos analisar a **eficiência** (ou complexidade no tempo) do algoritmo que encontra o maior elemento de um vetor A com n elementos.
- Basicamente, vamos encontrar o tempo de execução de cada passo do algoritmo em função do tamanho da entrada.

```
MAXIMO (A, n)                         Custo      #execuções
1. max = A[1]                           c1            1
2. para i = 2 até n faça                c2            n
3.    se max < A[i]                     c3          n - 1
4.       então max = A[i]               c4      0 <= t <= n - 1
5. escreva ("máximo: " max)             c5            1
```

- O tempo total de execução $T(n)$ do algoritmo é dado por

$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5.$$

- Esse tempo de execução é da forma $an + b$ para constantes a e b que dependem apenas dos custos.

- Em outras palavras, o algoritmo tem como complexidade no tempo uma **função linear** no tamanho da entrada.

- Mas nem sempre entradas de tamanho igual (i.e. mesmo valor de n) apresentam o mesmo tempo de execução.
- Veja o caso da busca linear, cujo algoritmo é mostrado abaixo. A ideia é verificar a existência do elemento x num vetor A.

```
LINEAR (A, n, x)                          Custo   #execuções
1. i = 1                                    c1
2. enquanto i <= n e x <> A[i] faça         c2
3.   i = i + 1                              c3
4. se i <= n                                c4
5.   então escreva ("encontrado")           c5
6.   senão escreva ("ausente")              c6
```

- Mas nem sempre entradas de tamanho igual (i.e. mesmo valor de n) apresentam o mesmo tempo de execução.
- Veja o caso da busca linear, cujo algoritmo é mostrado abaixo. A ideia é verificar a existência do elemento x num vetor A.

```
LINEAR (A, n, x)                         Custo   #execuções
1. i = 1                                   c1        1
2. enquanto i <= n e x <> A[i] faça        c2   1 <= t <= n + 1
3.   i = i + 1                             c3   0 <= t <= n
4. se i <= n                               c4        1
5.   então escreva ("encontrado")          c5      0 ou 1
6.   senão escreva ("ausente")             c6      1 ou 0
```

- No **melhor caso**, quando o elemento procurado encontra-se na primeira posição, o tempo total de execução $T(n)$ é

$$T(n) = c_1 + c_2 + c_4 + c_5 \ ou \ c_6.$$

- Nesse caso, o tempo de execução é uma **constante**, ou seja, não depende do tamanho da entrada.

- Considerando o **pior caso**, quando o elemento procurado não encontra-se no vetor, o tempo total de execução $T(n)$ é

$$T(n) = c_1 + c_2(n+1) + c_3 n + c_4 + c_5 \ ou \ c_6.$$

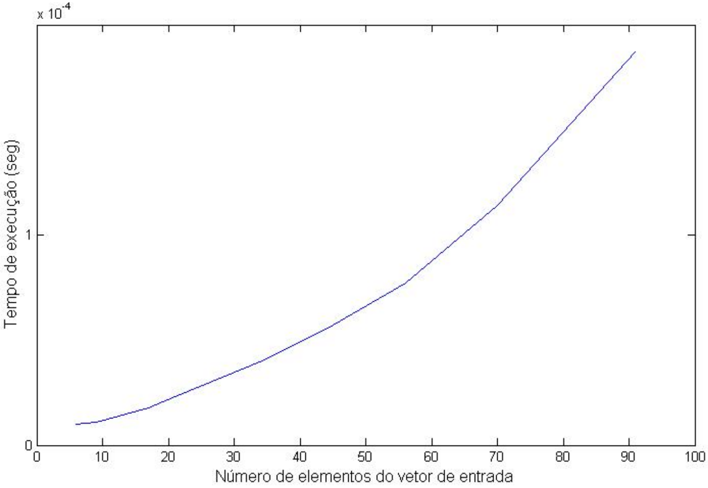
- Já aqui o tempo de execução é da forma $an + b$, ou seja, uma **função linear** no tamanho da entrada.

Exemplo: Algoritmo BubbleSort

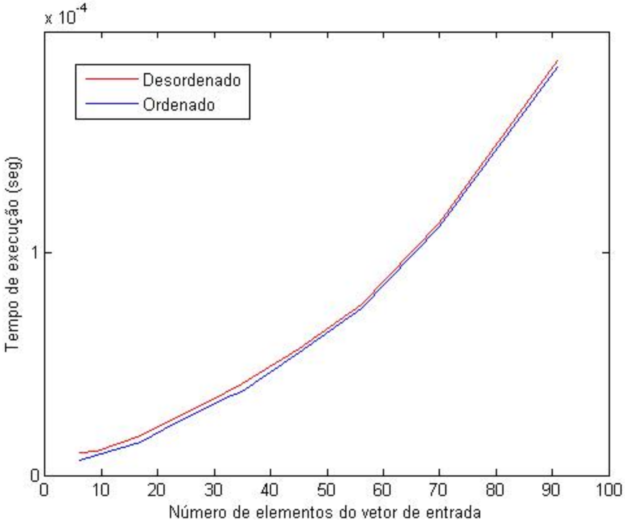
- O algoritmo para? Por quê?
- Qual é o loop invariante do algoritmo?
- Qual é a complexidade no tempo do algoritmo?
- Faça o gráfico: valor de entrada (n) x tempo de execução.
- Existe pior e melhor caso? Por exemplo, a ordenação inicial do vetor de entrada influencia na eficiência do algoritmo?

```
BUBBLESORT (A, n)
para i = 1 até n - 1 faça
   para j = 1 até n - i faça
      se A[j] > A[j + 1] então
            temp = A[j]
            A[j] = A[j + 1]
            A[j + 1] = temp
```

Exemplo: Algoritmo BubbleSort



Exemplo: Algoritmo BubbleSort



Caso médio

- O **caso médio** (ou caso esperado) corresponde à média dos tempos de execução de todas as entradas de tamanho n.
- Na análise do caso médio, uma distribuição de probabilidades sobre o conjunto de entrada de tamanho n é suposta, e o custo médio é obtido com base nessa distribuição.
- É comum supor uma distribuição de probabilidades em que todas as entradas possíveis são igualmente prováveis. Porém, na prática, isso nem sempre é verdade.
- Por essa razão, a análise do caso médio é geralmente mais difícil de obter do que as análises de melhor e pior caso.

Exemplo: Caso médio

- Uma pesquisa linear com sucesso examina aproximadamente metade dos registros no caso médio.

- Considere que toda pesquisa recupera um registro:

  Caso p_i seja a probabilidade de que o i-ésimo registro seja procurado e que para recuperar o i-ésimo registro são necessárias i comparações, então

  $$f(n) = 1p_1 + 2p_2 + 3p_3 + ... + np_n.$$

  Se cada registro tiver a mesma probabilidade de ser acessado que todos os outros, então $p_i = \frac{1}{n}$, $0 \leq i \leq n$. Nesse caso,

  $$f(n) = \frac{1}{n}(1 + 2 + 3 + ... + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}.$$

- Salvo contrário, considera-se o **pior caso** e o **comportamento assintótico** dos algoritmos (instâncias de tamanho grande).
- O estudo assintótico permite "jogar para debaixo do tapete" os valores das constantes e os termos não dominantes.

n	$4n + 52$	$4n$
64	308	256
512	2100	2048
2048	8244	8192
4096	16436	16384
8192	32820	32768
16384	65588	65536
32768	131124	131072

- De um modo geral, podemos nos concentrar nos **termos dominantes** e esquecer os demais.
- Considere a função quadrática $3n^2 + 10n + 50$:

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

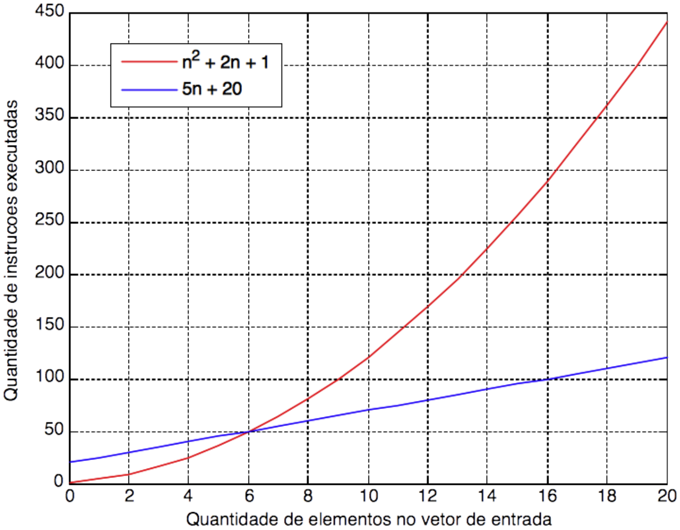
Comparando algoritmos

- Suponha que o Fulano desenvolveu um algoritmo para resolver o problema da ordenação por inserção, chamado *order_Ful*.
- Agora, suponha que o Beltrano fez outro algoritmo para resolver o mesmo problema, chamado *order_Bel*.
- Como saber qual dos dois algoritmos é mais eficiente?
- Podemos cronometrar o tempo?
  - Sim! Mas a análise empírica é uma ideia trabalhosa. Por quê?
  - Porque teríamos que medir o tempo de *order_Ful* e *order_Bel* para 1 elemento, 2 elementos, ..., 10 elementos, ..., 100 elementos, ..., 1000 elementos, ...

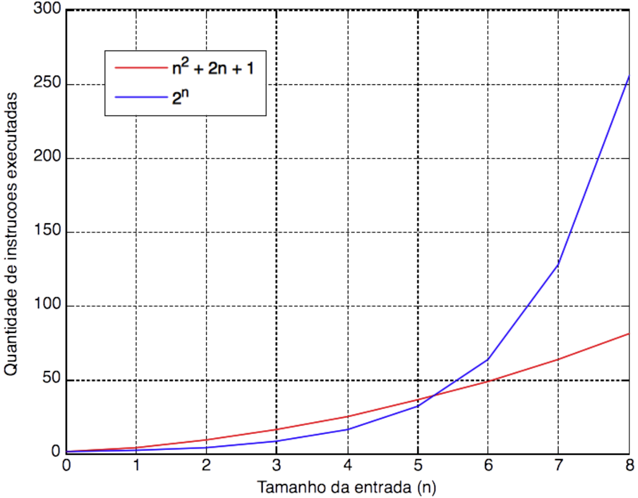
Comparando algoritmos

- Em geral, faz-se uma análise de complexidade no tempo.
- Calcula-se, para cada algoritmo, quantas instruções são executadas dada uma entrada de tamanho n.
- Suponha que o algoritmo *order_Ful* executa uma quantidade de instruções de acordo com a função: $f(n) = n^2 + 2n + 1$.
- Já o outro algoritmo *order_Bel* tem como complexidade no tempo a função: $f(n) = 5n + 20$.
- Por exemplo, se passarmos 50 elementos para *order_Ful*, ele fará $f(50) = 50^2 + 2 \;.\; 50 + 1 = 2.601$ instruções. Já *order_Bel* é mais eficiente, pois executará apenas 270 instruções.

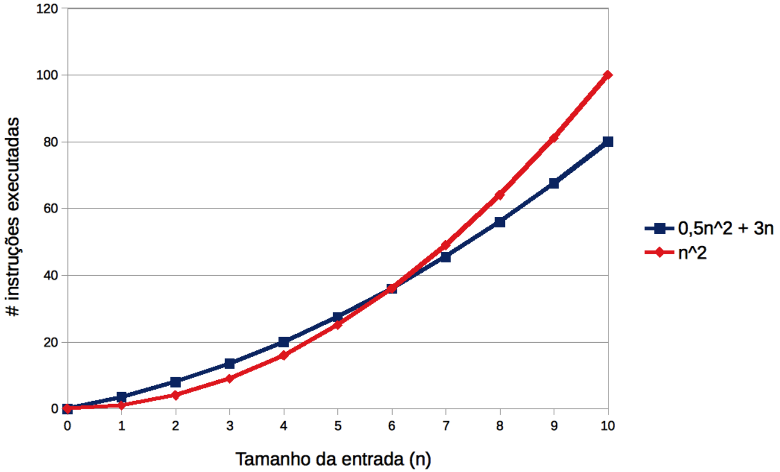
Comparando algoritmos



Comparando algoritmos



Comparando algoritmos



Conclusões

- A complexidade no tempo (= eficiência) de um algoritmo é dado pelo **número de instruções básicas que ele executa em função do tamanho da entrada**.
- Adota-se uma "atitude pessimista" (**análise do pior caso**).
- A análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE (**análise assintótica**).
- Um algoritmo é chamado eficiente se a função que mede sua complexidade no tempo é limitada por um **polinômio**.

  Ex: $n, 7n - 3, 14n^2 + 4n - 8, n^5$.

Conclusões

- Existe uma solução eficiente para qualquer tipo de problema?
- Não. Existem certos problemas para os quais **não** se conhece algoritmos eficientes capazes de resolvê-los.
- Problema do caixeiro viajante: Tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem.
- Mas por que é importante identificar quando estamos lidando com um problema "intratável"?

Exercícios

1. Qual é o menor valor de entrada n (considere $n > 0$) tal que um algoritmo cujo tempo de execução é $10n^2$ é mais rápido que um algoritmo cujo tempo de execução é 2^n?

   Qual desses algoritmos você considera mais eficiente?

Exercícios

2. Implemente em uma linguagem de programação a sua escolha o algoritmo abaixo que verifica se o valor de entrada é, ou não, um número primo. Existe pior e melhor caso?

```
PRIMO (n)
j = 2
enquanto (j < n e mod(n,j) <> 0) faça
    j = j + 1
se (j = n)
    então escreva ("é primo")
    senão escreva ("não é primo")
```

Exercícios

3. Implemente em uma linguagem de programação a sua escolha o algoritmo abaixo que lista os números primos menores ou igual ao valor de entrada. Existe pior e melhor caso?

```
LISTA_PRIMO (n)
x = 1
para i = 2 até n faça
   j = 2
   enquanto (j < i e mod(i,j) <> 0) faça
      j = j + 1
   se (j = i)
      A(x) = i
      x = x + 1
retorne A[x]
```

Exercícios

4. Cada um dos algoritmos abaixo recebe um inteiro positivo e devolve outro inteiro positivo. Os dois algoritmos devolvem o mesmo número se receberem o mesmo valor de entrada *n*? Qual dos dois algoritmos é mais eficiente?

```
Soma-Quadrados-A (n)
    x = 0
    para j = 1 até n faça
        x = x + j * j
    retorne x
```

```
Soma-Quadrados-B (n)
    x = n * (n + 1) * (2n + 1)
    x = x/6
    retorne x
```

Exercícios

5. Compare assintoticamente o tempo de execução (apresente dados, tabelas e gráficos) dos algoritmos de ordenação:
   - Inserção;
   - BubbleSort; e
   - Seleção.

   Em seguida, responda os seguintes questionamentos.

   a. Qual desses algoritmos é o mais eficiente?

   b. A ordem inicial do vetor de entrada influencia no desempenho dos algoritmos?

   c. Qual desses algoritmos você usaria na sua aplicação?