SIN110 Algoritmos e Grafos

aula 01

Análise de Algoritmos

SIN110 Algoritmos e Grafos: apresentação

… algoritmos estudados …

COM110 Fundamentos de Programação
→ Resolução de problemas e desenvolvimento de algoritmos. Introdução às linguagens de programação. Mapeamento de algoritmos em programas computacionais. Estruturas de dados básicas: vetores, matrizes e registros. Noções de recursividade.

COM111 Algoritmos e Estruturas de Dados I
→ Introdução às estruturas de dados. Tipos abstratos de dados. Pilhas. Recursividade. Avaliação de expressões. Filas e Listas. Árvores Binárias. Alocação Dinâmica.

COM112 Algoritmos e Estruturas de Dados II
→ Métodos de Ordenação; Pesquisa de Dados: árvores de pesquisa (AVL, Red-Black, Splay, B e outras variações). Hashing. Organização de arquivos. Noções de Complexidade.

SIN110 Algoritmos e Grafos: apresentação

Ementa

Algoritmos: conceitos, análise e eficiência, técnicas de projeto (divisão e conquista, programação dinâmica, método guloso, backtraking).

Grafos: conceitos, representação, pesquisas em largura e profundidade, ordenação topológica, árvore geradora, caminhos mínimos.

SIN110 Algoritmos e Grafos: apresentação

Aulas:

- Presencial: segundas-feiras, 21h – 23h30, LDC2
- AVA: nead.unifei.edu.br/moodle ~ SIN110

Prof R Claudino

- ✉ claudino@unifei.edu.br
- 3629-1834

SIN110 Algoritmos e Grafos: apresentação

Objetivos:

- *Algoritmos*:
  - métodos para analisar a eficiência de algoritmos
  - aprender técnicas de desenvolvimento
  - analisar algoritmos conhecidos para diversos tipos de problemas
- *Grafos*:
  - Conceitos da Teoria de Grafos
  - Modelar soluções de problemas com grafos
  - Implementar os principais algoritmos de grafos.

CRONOGRAMA

mês	dom	seg	ter	qua	qui	sex	sab	aula	SIN110 Algoritmos & Grafos	
jul	28	29	30	31	1	2	3	--	---	--
ago	4	5	6	7	8	9	10	--	atendimento PRDA/ajustes matricula	--
	11	12	13	14	15	16	17	1	Análise de algoritmos	--
	18	19	20	21	22	23	24	2	Algoritmos iterativos	--
	25	26	27	28	29	30	31	3	Algoritmos recursivos	E01
set	1	2	3	4	5	6	7	4	Téc Proj: indução	E02
	8	9	10	11	12	13	14	5	Téc Proj: divisão e conquista	E03
	15	16	17	18	19	20	21	6	Téc Proj: programação dinâmica	E04
	22	23	24	25	26	27	28	7	Téc Proj: método guloso	E05
out	29	30	1	2	3	4	5	8	Téc Proj: backtraking	E06
	6	7	8	9	10	11	12	9	Grafos: modelagem, representação	--
	13	14	15	16	17	18	19	10	Grafos: buscas em profundidade	E07
	20	21	22	23	24	25	26	11	Grafos: buscas em largura	E08
	27	28	29	30	31	1	2	--	SED (dia do Servidor Público)	--
nov	3	4	5	6	7	8	9	12	Grafos: aplicações/ordenação topológica	E09
	10	11	12	13	14	15	16	13	Grafos: árvore geradora	E10
	17	18	19	20	21	22	23	14	Grafos: caminhos mínimos	E11
	24	25	26	27	28	29	30	15	Grafos: fluxo em redes	E12
dez	1	2	3	4	5	6	7	16	Avaliação	--
	8	9	10	11	12	13	14	--	Aval Substitutiva	--

SIN110 Algoritmos e Grafos: apresentação

Avaliação:

- exercícios de fixação (12 Ex's da 4ª à 15ª aula)

- avaliação com toda matéria no encerramento

→ média = 0,3*Ex's + Avaliação*0,7

Aprovação:

média final ≥ 6,0 pontos e presença em 75% das aulas

Prevista prova substitutiva da avaliação, se necessário.

SIN110 Algoritmos e Grafos: apresentação

Bibliografia:

- CORMEN, T. H. ; LEISERSON, C. E. ; RIVEST R. L. ; STEIN, C. ***Algoritmos: teoria e prática***. 3ª ed. Campus. 2012
- BOAVENTURA NETTO, Paulo Oswaldo. **Grafos: teoria, modelos,algoritmos**. 4ª ed. Edgard Blücher. 2006
- ZIVIANE N. **Projetos de Algoritmos**. Cengage Learning. 3ª ed, 2013.
- WEISS, M. A. **Data Structures and Algorithm Analysis in C**; 2nd ed, Addison – Wesley, 1997
- DROZDEK, Adam. **Estrutura de dados e algoritmos em C++.** LTC. 2007
- BASSARD, G. BRATLEY, P. **Fundamentals of Algorithmic**; Prentice-Hall.1996.
- MANBER, U. **Introduction to Algorithms: a creative approach**; Addison-Wesley, 1989.
- SZWARCFITER, Jayme L. **Grafos e Algoritmos Computacionais.** Campus. 1984.
- SEDGEWICK, R. **Algorithms in C++ parts 1 – 4: Fundamentals, Data Structures, Sorting, Searching**, 3rd edition; Addison-Wesley. 1998.

SIN110 Algoritmos e Grafos

Análise de Algoritmos

Algoritmos

- Sequencia de ações executáveis para obtenção de uma solução para um determinado tipo de problema.
- Segundo Dijkstra, *um algoritmo corresponde a uma descrição de um padrão de comportamento, expresso em termos de um conjunto finito de ações*. Por exemplo, na operação $x + y$ observamos um padrão de comportamento mesmo que a operação seja realizada para valores diferentes de x e y.
- *Um algoritmo é uma ferramenta para resolver um determinado problema computacional.*

  A descrição do problema define como deve ser a relação entre a entrada e a saída do algoritmo.

Algoritmos

Problema: rearranjar um vetor $A[1 \ldots n]$ em ordem crescente.

Entrada:

1										n
33	55	33	44	33	22	11	99	22	55	77

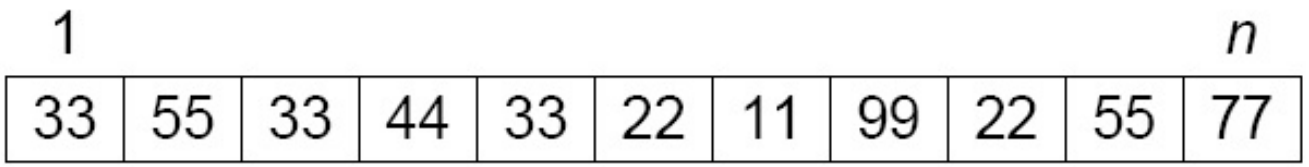
Saída:

1										n
11	22	22	33	33	33	44	55	55	77	99

Algoritmos

Instância de um problema

Dizemos que o vetor



é uma instância do problema de ordenação.

Em geral, uma instância de um problema é um conjunto de valores que serve de entrada para o problema (respeitando as restrições impostas na descrição deste).

Análise de Algoritmos

O que se analisa ...

- Como estimar a quantidade de **recursos** (tempo, memória) que um algoritmo consome/gasta = análise de complexidade
- Como provar a "corretude" de um algoritmo
- Como projetar algoritmos eficientes (= rápidos) para vários problemas computacionais

Análise de Algoritmos

Uso/desenvolvimento de algoritmos eficientes é desejável em áreas como:

- projetos de genoma de seres vivos
- rede mundial de computadores
- sistemas de informação geográfica
- comércio eletrônico
- planejamento da produção de indústrias
- logística de distribuição
- ...

Algoritmos e Tecnologia

O avanço da tecnologia permite a construção de máquinas cada vez mais rápidas.

Isto possibilita que um algoritmo para determinado problema possa ser executado mais rapidamente.

Paralelamente a isto, há o projeto/desenvolvimento de algoritmos ”intrinsecamente mais eficientes” para determinado problema.

Isto leva em conta apenas as características inerentes ao problema, desconsiderando detalhes de software/hardware.

Algoritmos e Tecnologia

Vamos comparar esses dois aspectos em um exemplo com a ordenação de um vetor com n elementos:

- Suponha que os computadores A e B executam $1G$ e $10M$ instruções por segundo, respectivamente. Ou seja, A é **100 vezes mais rápido** que B.
- Algoritmo 1: implementado em A por um excelente programador em linguagem de máquina (ultra-rápida). Executa $2n^2$ instruções.
- Algoritmo 2: implementado na máquina B por um programador mediano em linguagem de alto nível dispondo de um compilador "meia-boca". Executa $50n \log_{10} n$ instruções.

Algoritmos e Tecnologia

O que acontece se ordenamos um vetor com 1 milhão de elementos? Qual algoritmo é mais rápido?

- Algoritmo 1 na máquina *A*:
$$\frac{2.(10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000 \text{ segundos}$$
- Algoritmo 2 na máquina *B*:
$$\frac{50.(10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos}$$
- Ou seja, *B* foi **VINTE VEZES** mais rápido do que *A*!
- Se o vetor tiver 10 milhões de elementos, esta razão será de 2.3 dias para 20 minutos!

Algoritmos e Tecnologia

O uso de um *algoritmo* em lugar de outro pode levar a ganhos extraordinários de *desempenho*.

Isso pode ser tão importante quanto o projeto de hardware

A melhoria obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.

Complexidade de Algoritmos

Queremos projetar / desenvolver *algoritmos eficientes* (rápidos).

Más o que seria uma boa *medida de eficiência* de um algoritmo?

Não interessa quem programou, em qual linguagem foi escrito e nem qual máquina foi usada!

Queremos um critério uniforme para *comparar algoritmos*.

Algoritmo e o modelo computacional

Modelo abstrato inclui apenas o que é relevante para o processamento de algoritmos.

Elementos do modelo:

- um único processador
- memória

Obs.:

- ignora-se dispositivos de entrada/saída, assume-se a codificação do algoritmo e os dados já armazenados na memória.
- Na modelagem não é relevante saber como algoritmo e dados foram armazenados

Algoritmo e o modelo computacional

RAM *(Random Access Machine)*

Computação no modelo

- Processador busca instrução/dado na memória
- Uma única instrução é executada de cada vez
- Cada instrução é executada sequencialmente

Função complexidade de tempo: cada operação executada incluindo cálculos aritméticos, lógicos e acesso a memória implica um custo de tempo.

Função complexidade de espaço: cada operação e dado armazenado implicada um custo de espaço.

Complexidade de tempo e de espaço

- Complexidade de tempo não representa tempo diretamente, mas o número de vezes que determinada instrução é executada.

- Complexidade de espaço representa a quantidade de memória (numa unidade qualquer) que é necessário para armazenar as estruturas de dados associadas aos algoritmos.

Complexidade de algoritmos: um exemplo

Problema: ordenar um vetor em ordem crescente

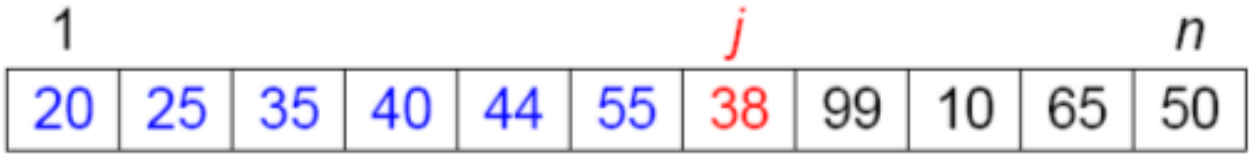
Entrada: um vetor $A[1 \ldots n]$

Saída: vetor $A[1 \ldots n]$ rearranjado em ordem crescente

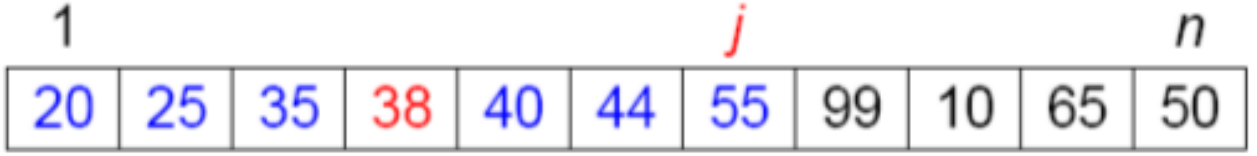
Vamos começar analisando o algoritmo de ordenação baseado no método de inserção (Insertion sort).
Isto nos permitirá destacar alguns dos aspectos mais importantes no estudo de algoritmos para esta disciplina.

Complexidade de algoritmos: um exemplo

Inserção em um vetor ordenado



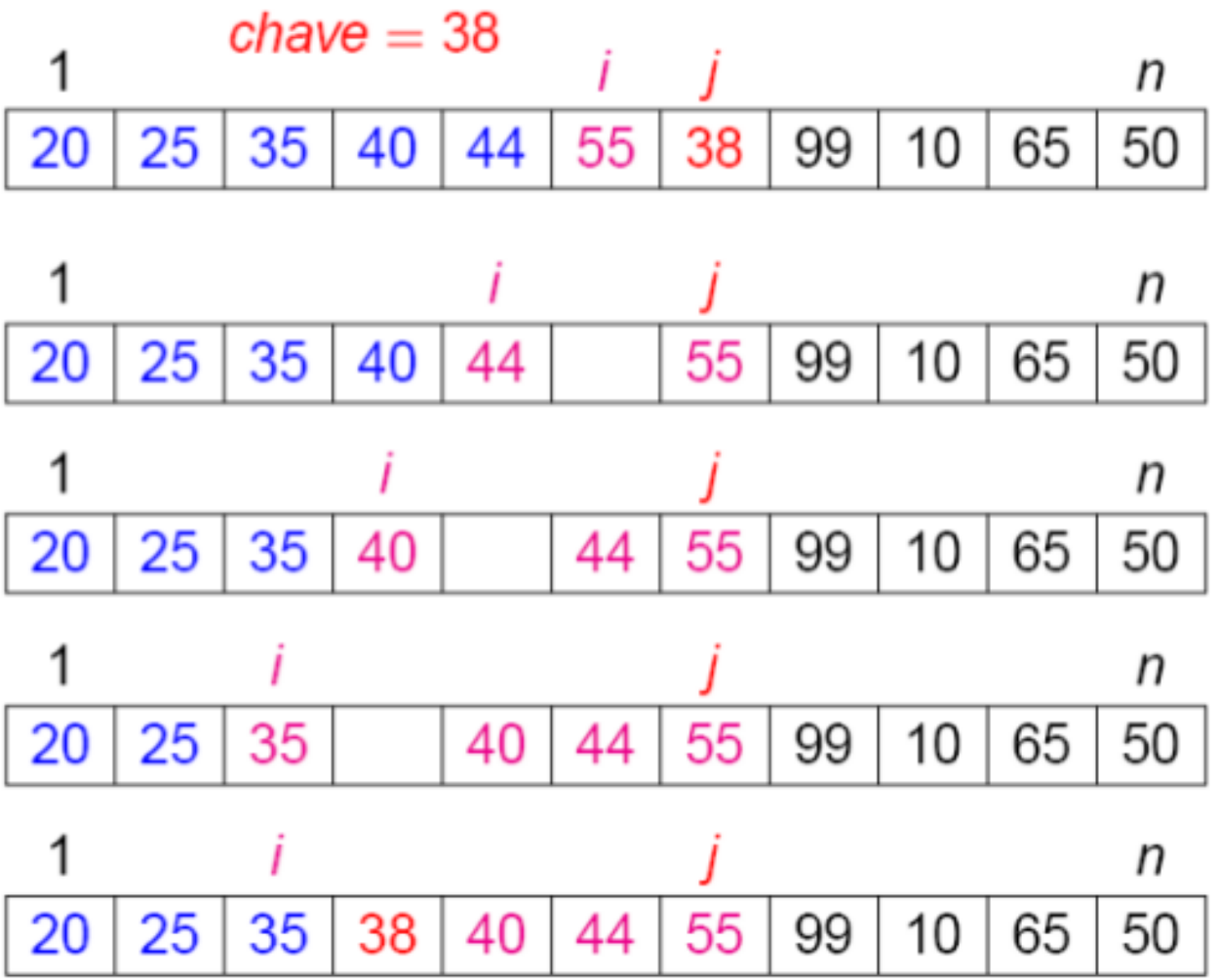
- O subvetor $A[1 \ldots j-1]$ está ordenado.
- Queremos inserir a $chave = 38 = A[j]$ em $A[1 \ldots j-1]$ de modo que no final tenhamos:



- Agora $A[1 \ldots j]$ está ordenado.

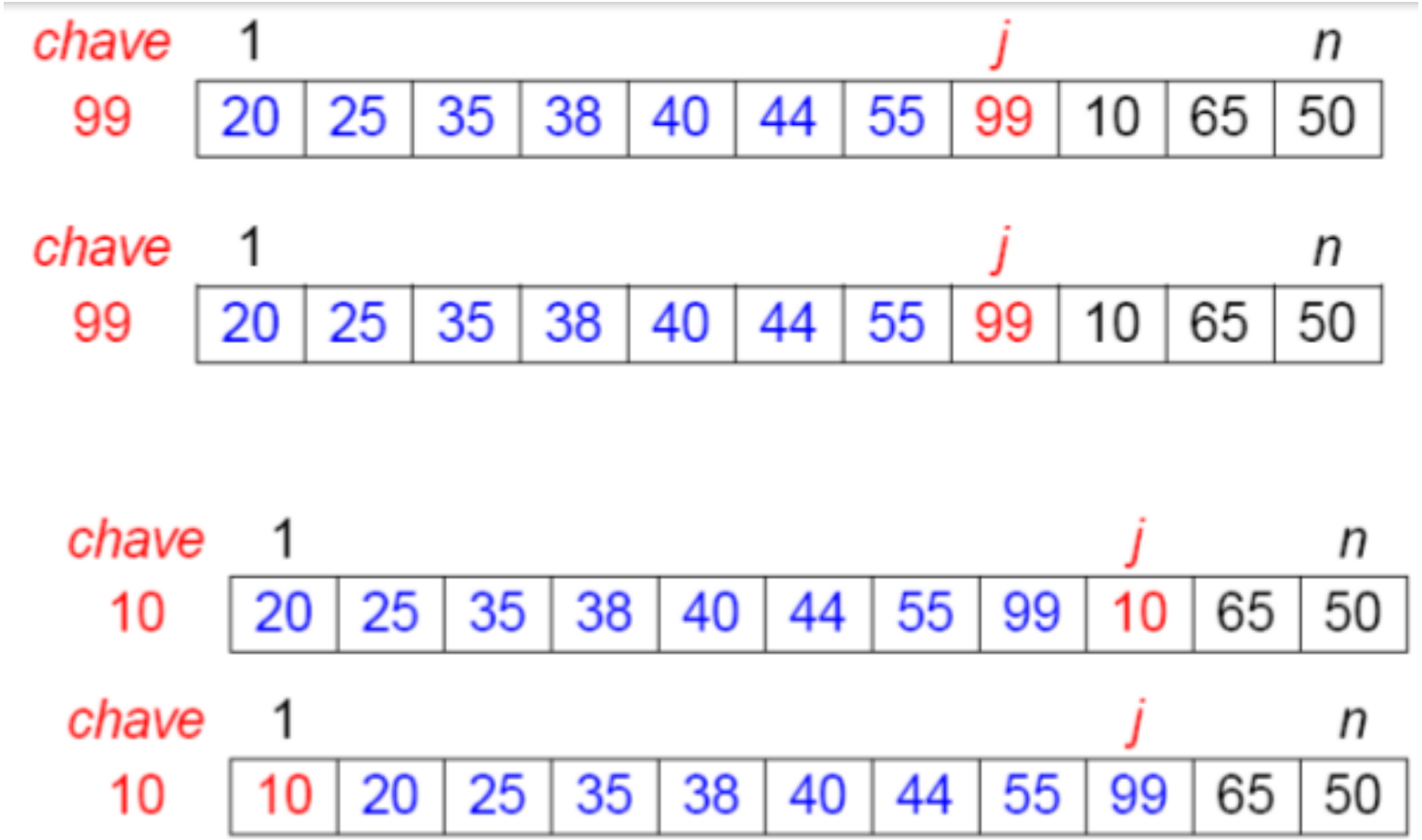
Complexidade de algoritmos: um exemplo

Como fazer a inserção



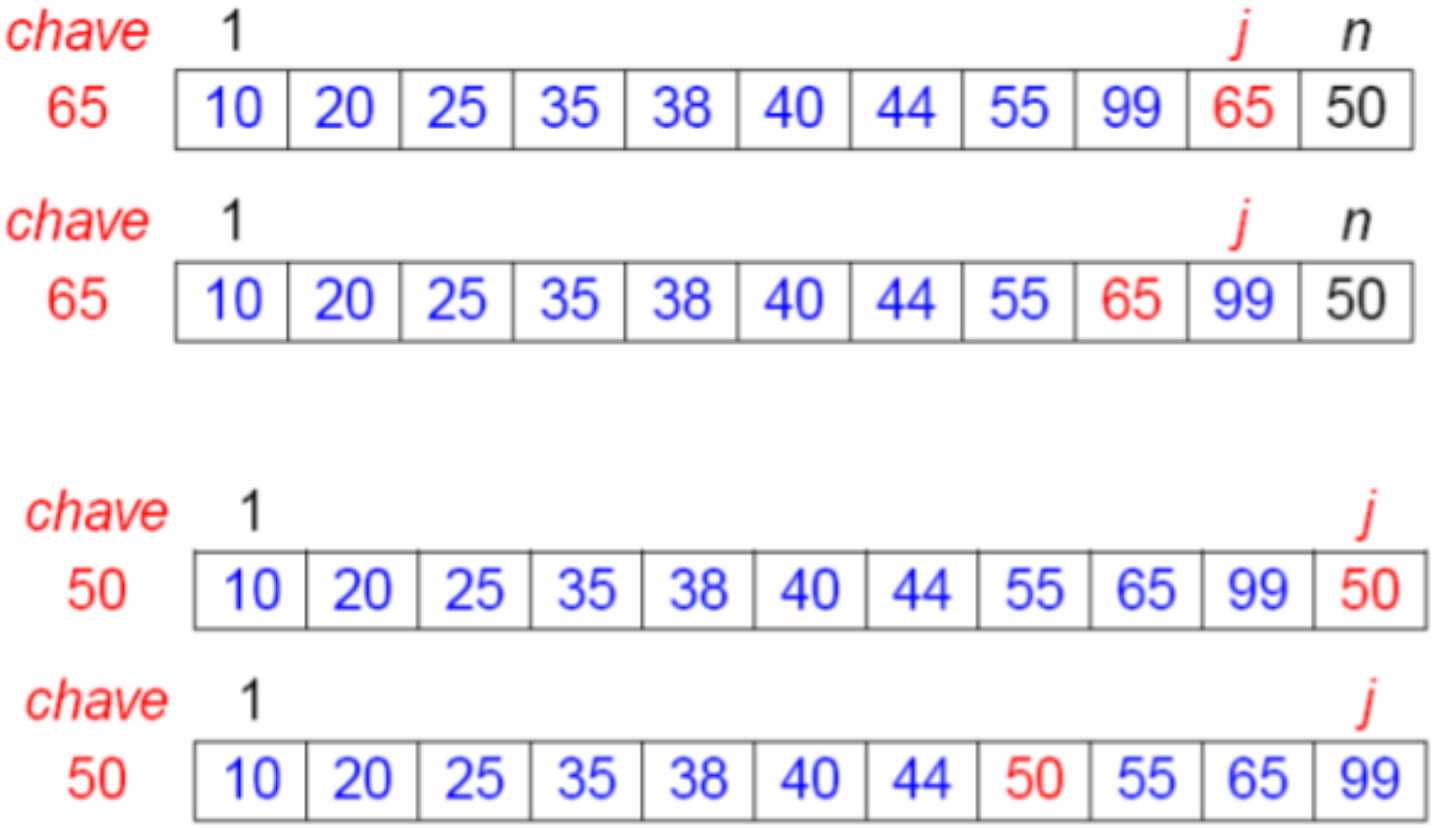
Complexidade de algoritmos: um exemplo

Ordenação por inserção



Complexidade de algoritmos: um exemplo

Ordenação por inserção



Complexidade de algoritmos: um exemplo

Descrevendo um algoritmo

Podemos formalizar o algoritmo *Ordena-por-inserção* de várias formas:

- Usando uma linguagem de progrmação de alto nível; C, Java, Python, etc.
- Implementando em linguagem de máquina diretamente executável no *hardware*.
- Em pseudocódigo

Adotaremos essa última forma.

Complexidade de algoritmos: um exemplo

Pseudocódigo

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

Complexidade de algoritmos: um exemplo

Análise do algoritmo

O que é importante analisar/considerar?

- Corretude do algoritmo: é preciso mostrar que para toda instância do problema, o algoritmo pára e devolve uma resposta correta.
- Complexidade de tempo do algoritmo: quantas intruções são necessárias no pior caso para ordenar os *n* elementos?

Complexidade de algoritmos

Para um dado problema, podemos ter vários algoritmos para resolvê-lo, cada um deles com uma complexidade diferente.

A menor dessas é a *cota superior de complexidade n*

Esta cota nos diz que, para instâncias arbitrárias de tamanho *n*, podemos resolver o problema em tempo menor ou igual à *cota superior(n)*.

Complexidade de algoritmos

Algoritmos são analisados para se determinar a sua complexidade, deixando-se em aberto a possibilidade de se reduzir ainda mais a *cota superior(n)*, e descobrir um novo algoritmo cuja complexidade pessimista seja menor do que qualquer algoritmo já conhecido.

Pesquisa-se quão rapidamente podem-se resolver *todas* as instâncias de um dado problema, independentemente do algoritmo que exista ou que se possa descobrir no futuro.

Complexidade de algoritmos

Isto é, interessa uma complexidade inerente ou intrínseca do problema que é chamada *cota inferior de complexidade n* do problema.

Esta cota nos diz que nenhum algoritmo pode resolver o problema com complexidade pessimista menor do que *cota inferior(n)*, para entradas arbitrárias de tamanho *n*.

Esta cota é uma propriedade do problema considerado, e não de um algoritmo particular.

Por exemplo, o *problema da ordenação de n números*, demonstra-se que sua *cota inferior* é uma função proporcional a *nlgn*.

Complexidade de algoritmos

A distinção entre cotas superior e inferior considera que ambas são *mínimas* sobre a complexidade máxima (pessimista) para entradas de tamanho *n*.

Cota inferior(n) é o mínimo, sobre todos os algoritmos possíveis da complexidade máxima;

Cota superior(n) é o mínimo sobre todos os algoritmos existentes, da complexidade máxima.

Aprimorar uma *cota superior* significa descobrir um algoritmo que tenha uma *cota inferior* melhor do que a existente

Complexidade de algoritmos

Concentramos em técnicas que nos permitam aumentar a precisão com a qual o mínimo, sobre todos os algoritmos possíveis, pode ser limitado.

Esta distinção nos leva às diferenças nas técnicas desenvolvidas em análise de complexidade.

Embora existam aparentemente duas funções de complexidade de problemas, as *cotas superior* e *inferior*, o objetivo final é fazer estas duas funções coincidirem: *cota inferior(n) = cota superior(n)*, Quando isto ocorre, temos o algoritmo "ótimo", para a maioria dos problemas, este objetivo ainda não foi alcançado.

Complexidade de algoritmos: análise do exemplo

O algoritmo pára

```
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
8       ...
```

No laço **enquanto** na linha 5 o valor de i diminui a cada iteração e o valor inicial é $i = j - 1 \geq 1$. Logo, a sua execução pára em algum momento por causa do teste condicional $i \geq 1$.

O laço na linha 1 evidentemente pára (o contador j atingirá o valor $n + 1$ após $n - 1$ iterações).

Portanto, o algoritmo **pára**.

Complexidade de algoritmos: análise do exemplo

Ordena – por - inserção

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

O que falta fazer?

- Verificar se ele produz uma resposta correta.
- Analisar sua complexidade de tempo.

Complexidade de algoritmos: análise do exemplo

Invariante de laço e provas de corretude

- Definição: um invariante de um laço é uma propriedade que relaciona as variáveis do algoritmo a cada execução completa do laço.
- Ele deve ser escolhido de modo que, ao término do laço, tenha-se uma propriedade útil para mostrar a corretude do algoritmo.
- A prova de corretude de um algoritmo requer que sejam encontrados e provados invariantes dos vários laços que o compõem.
- Em geral, é mais difícil descobrir um invariante apropriado do que mostrar sua validade se ele for dado de bandeja. . .

Complexidade de algoritmos: análise do exemplo

Exemplo de invariante

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

No começo de cada iteração do laço **para** das linha 1–8, o subvetor $A[1 \ldots j-1]$ está ordenado.

Complexidade de algoritmos: análise do exemplo

Corretude de algoritmos por invariante

A estratégia "típica" para mostrar a corretude de um algoritmo iterativo através de invariantes segue os seguintes passos:

1. Mostre que o invariante vale no início da primeira iteração (trivial, em geral)
2. Suponha que o invariante vale no início de uma iteração qualquer e prove que ele vale no ínicio da próxima iteração
3. Conclua que se o algoritmo pára e o invariante vale no ínicio da última iteração, então o algoritmo é correto.

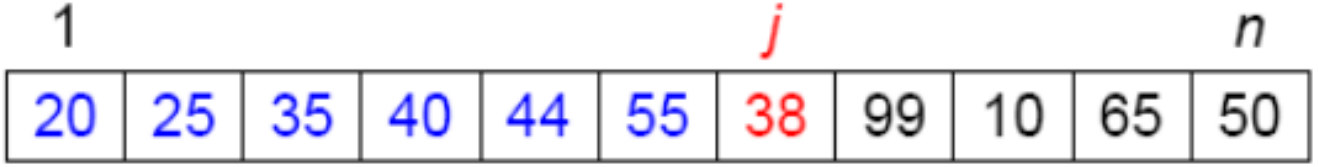
Note que (1) e (2) implicam que o invariante vale no início de qualquer iteração do algoritmo. Isto é similar ao método de indução matemática ou indução finita!

Complexidade de algoritmos: análise do exemplo

Corretude da Ordenação-por-inserção

Vamos verificar a corretude do algoritmo de ordenação por inserção usando a técnica de **prova por invariantes de laços**.

No começo de cada iteração do laço **para** das linha 1–8, o subvetor $A[1 \ldots j-1]$ está ordenado.



- Suponha que o invariante vale.
- Então a corretude do algoritmo é "evidente". **Por quê?**
- No ínicio da última iteração temos $j = n + 1$. Assim, do invariante segue que o (sub)vetor $A[1 \ldots n]$ está ordenado!

Complexidade de algoritmos: análise do exemplo

Análise do algoritmo

O que é importante analisar/considerar?

- Corretude do algoritmo

- Complexidade de tempo do algoritmo: quantas intruções são necessárias no pior caso para ordenar os n elementos?

Complexidade de algoritmos: análise do exemplo

Complexidade do algoritmo

- Vamos tentar determinar o tempo de execução (ou complexidade de tempo) de ORDENA-POR-INSERÇÃO em função do tamanho de entrada.
- Para o Problema de Ordenação definimos como tamanho de entrada a número de elementos do vetor.
- A complexidade de tempo de um algoritmo é o número de instruções básicas (operações elementares ou primitivas) que executa a partir de uma entrada.
- Exemplo: comparação e atribuição entre números ou variáveis numéricas, operações aritméticas, etc.

Complexidade de algoritmos: análise do exemplo

Vamos contar ...

ORDENA-POR-INSERÇÃO(A, n)	Custo	# execuções
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	?
2 $\quad chave \leftarrow A[j]$	c_2	?
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	?
4 $\quad i \leftarrow j-1$	c_4	?
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	?
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6	?
7 $\quad\quad i \leftarrow i-1$	c_7	?
8 $\quad A[i+1] \leftarrow chave$	c_8	?

O valor c_k representa o custo (tempo) de cada execução da linha k.

Denote por t_j o número de vezes que o teste no laço **enquanto** na linha 5 é feito para aquele valor de j.

Complexidade de algoritmos: análise do exemplo

Vamos contar ...

ORDENA-POR-INSERÇÃO(A, n)	Custo	Vezes
1 **para** $j \leftarrow 2$ **até** n **faça**	c_1	n
2 $\quad chave \leftarrow A[j]$	c_2	$n-1$
3 $\quad \triangleright$ Insere $A[j]$ em $A[1 \ldots j-1]$	0	$n-1$
4 $\quad i \leftarrow j-1$	c_4	$n-1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_5	$\sum_{j=2}^{n} t_j$
6 $\quad\quad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\quad\quad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $\quad A[i+1] \leftarrow chave$	c_8	$n-1$

O valor c_k representa o custo (tempo) de cada execução da linha k.

Denote por t_j o número de vezes que o teste no laço **enquanto** na linha 5 é feito para aquele valor de j.

Complexidade de algoritmos: análise do exemplo

Tempo de execução total

Logo, o tempo total de execução $T(n)$ de Ordena-Por-Inserção é a soma dos tempos de execução de cada uma das linhas do algoritmo, ou seja:

$$\begin{aligned} T(n) \quad = \quad & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ & + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) \\ & + c_8(n-1) \end{aligned}$$

Como se vê, entradas de tamanho igual (i.e., mesmo valor de n), podem apresentar tempos de execução diferentes já que o valor de $T(n)$ depende dos valores dos t_j.

Complexidade de algoritmos: análise do exemplo

O pior caso ...

Quando o vetor A está em ordem decrescente, ocorre o pior caso para Ordena-Por-Inserção. Para inserir a *chave* em $A[1 \ldots j-1]$, temos que compará-la com todos os elementos neste subvetor. Assim, $t_j = j$ para $j = 2, \ldots, n$.

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

O tempo de execução no pior caso é da forma $an^2 + bn + c$ onde a, b, c são constantes que dependem apenas dos c_i.

Portanto, **no pior caso**, o tempo de execução é uma **função quadrática** no **tamanho da entrada**.

Complexidade de algoritmos: análise do exemplo

... e o melhor caso

O melhor caso de Ordena-Por-Inserção ocorre quando o vetor A já está ordenado. Para $j = 2, \ldots, n$ temos $A[i] \leq chave$ na linha 5 quando $i = j - 1$. Assim, $t_j = 1$ para $j = 2, \ldots, n$.

Logo,

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

Este tempo de execução é da forma $an + b$ para constantes a e b que dependem apenas dos c_i.

Portanto, **no melhor caso**, o tempo de execução é uma **função linear** no **tamanho da entrada**.

Complexidade de algoritmos: exemplo 2

Quanto vale S após a execução do algoritmo?

```
1    S ← 0
2    para i ← 2 até n-2 faça
3        para j ← i até n faça
4            S ← S + 1
```

Complexidade de algoritmos: exemplo 2

Quanto vale S após a execução do algoritmo?

```
1   S ← 0
2   para i ← 2 até n-2 faça
3       para j ← i até n faça
```

Solução:

Se $n \geq 4$, então ao final da execução das linha 1 – 4, temos

$S = (n-1) + (n-2) + (n-3) + \ldots + 4 + 3 = (n+2)(n-3)/2$

$S = n^2/2 - n/2 - 3$

resposta função associada ao tamanho da instância *n*.

Complexidade de algoritmos: exemplo 3

Análise do BubleSort

```
BubleSort (A, n)
1   para i ← 1 até n-1 faça
2         para j ← n até i+1 faça
3               se A[j] < A[j-1
4                     então troca(A[j],A[j-1])
```

Complexidade de algoritmos: exemplo 3

```
BubleSort (A, n)
1   para i ← 1 até n-1 faça
2       para j ← n até i+1 faça
3           se A[j] < A[j-1
4               então troca(A[j],A[j-1])
```

execução

i	j	A_{j-1}	A_j	A[1..n]resultante
1	4	0	-2	5, 7, -2, 0
1	3	7	-2	5, -2, 7, 0
1	2	5	-2	**-2**, 5, 7, 0
2	4	7	0	**-2**, 5, 0, 7
2	3	5	0	**-2**, **0**, 5, 7
3	4	5	7	**-2**, **0**, **5**, **7**

Complexidade de algoritmos: exemplo 3

```
BubleSort (A, n)
1   para i ← 1 até n-1 faça
2         para j ← n até i+1 faça
3               se A[j] < A[j-1
4                     então troca(A[j],A[j-1])
```

Correção:

- *O algoritmo pára?*
- *Dá uma resposta correta?*

Complexidade de algoritmos: exemplo 3

```
BubleSort (A, n)
1   para i ← 1 até n-1 faça
2         para j ← n até i+1 faça
3               se A[j] < A[j-1
4                     então troca(A[j],A[j-1])
```

Correção:

- *O algoritmo pára?*

  → o laço principal (linha 1) e o laço interno (linha 2) executam um número finito de repetições controladas por contadores, i e j respectivamente; portanto ***pára****.*

Complexidade de algoritmos: exemplo 3

```
BubleSort (A, n)
1   para i ← 1 até n-1 faça
2         para j ← n até i+1 faça
3               se A[j] < A[j-1
4                     então troca(A[j],A[j-1])
```

Correção:

- *Dá uma resposta correta?*

  O percurso da esquerda para direita no vetor, via laço externo, aliado ao laço interno que faz em cada iteração "i", uma varredura da direita para esquerda com "j" e, a seleção da linha 3 com a troca da linha 4 garantem que o menor elemento seja deslocado seja posicionado em "i" dando a resposta esperada: ordena o vetor!

Complexidade de algoritmos: exemplo 3

```
BubleSort (A, n)
1   para i ← 1 até n-1 faça
2         para j ← n até i+1 faça
3               se A[j] < A[j-1
4                     então troca(A[j],A[j-1])
```

Complexidade de tempo:

- *contagem*
  linha 1: n execuções
  linha 2: n + (n-1) + ... + 2 = (n+2)(n-1)/2 execuções
  linha 3: (n-1) + (n-2)+... + 1 = (n)(n-1)/2 execuções
  linha 4: (n-1) + (n-2)+... + 1 = (n)(n-1)/2 execuções
- *Total = n + [(n+2)+n+n](n-1)/2 execuções*
  obtemos a função $T(n) = (3n^2 + n - 2)/2$ em função de n.

Exercícios

1. Analise o algoritmo Ordena – por – Seleção

```
Ordena-Por-Seleção(A,n)
1  para i ← 1 até n-1 faça
2      min ← i
3      para j ← i+1 até n faça
4          se A[j] < A[min]
5              então min ← j
6      aux ← A[min]
7      A[min] ← A[i]
8      A[i] ← aux
```

Exercícios

2. Quanto tempo consome o algoritmo abaixo que opera sobre um vetor A[1..n]?

```
Alg2 (A, n)
1  S ← 0
2  para i ← 1 até n faça
3      S ← S + A[i]
4  m  ←  S/n
5  k  ←  1
6  para i ← 2 até n faça
7      se(A[i] - m)² < (A[k] - m)²
8          então  k  ←  i
9  devolve k
```

Exercícios

3. Quanto vale S após a execução do algoritmo?

```
Soma(n)
1 S ← 0
2 i ← n
3 enquanto i > 0 faça
4   para j ← 1 até i faça
5       S ← S + 1
6   i ← ⌊i/2⌋
```