Projeto e Análise de Algoritmos

A. G. Silva, R. de Santiago

15 de março de 2024

Disciplina

INE410104 – Projeto e Análise de Algoritmos

- **Professores:** Alexandre Gonçalves Silva (INE-506), Rafael de Santiago (INE-412)
  - https://www.inf.ufsc.br/~alexandre.goncalves.silva/
    https://www.inf.ufsc.br/~r.santiago/
  - {alexandre.goncalves.silva,r.santiago}@ufsc.br
- **Carga horária:** 60 horas-aula (4 créditos)
- **Curso:** Programa de Pós-Graduação em Ciência da Computação
- **Requisitos:** não há
- **Período:** 1º semestre de 2024
- **Materiais (M/D):**
  - https://presencial.moodle.ufsc.br/course/view.php?id=9685
- **Horários:**
  - 6ª 08h20-11h50 (4 aulas) – Sala Virtual

Ementa

• Introdução a análise e projeto de algoritmos • Complexidade • Notação assintótica • Recorrências • Algoritmos de divisão e conquista • Algoritmos Gulosos • Programação Dinâmica • Problemas NP-Completos • Reduções • Técnicas para tratar problemas complexos

Objetivos

- **Geral:** Compreender o processo de análise de complexidade de algoritmos e conhecer as principais técnicas para o desenvolvimento de algoritmos, aplicações e análises de complexidade.

- **Específicos:** • Compreender o processo de análise de complexidade de algoritmos • Conhecer as principais técnicas para o desenvolvimento de algoritmos e suas análises • Compreender a diferença entre complexidade de problemas e complexidade de soluções • Conhecer e compreender as classes de complexidade de problemas • Conhecer algoritmos para tratar problemas complexos

Conteúdo programático

- Introdução (4 horas/aula)
- Notação Assintótica e Crescimento de Funções (4 horas/aula)
- Recorrências (4 horas/aula)
- Divisão e Conquista (8 horas/aula)
- Grafos. Buscas (8 horas/aula)
- Algoritmos Gulosos (8 horas aula)
- Programação Dinâmica (8 horas/aula)
- NP-Completude e Reduções (6 horas/aula)
- Algoritmos Aproximados e Heurísticas (6 horas/aula)

Metodologia e avaliação

Metodologia:

- Aulas expositivas, resolução de problemas, leituras extraclasse e trabalho de pesquisa.

Avaliação:

- O aluno será aprovado na disciplina se obtiver Nota Final (NF) igual ou superior a **7,0** e frequência igual ou superior a **75%**. A NF será calculada pela fórmula:

$$NF = 0,7\frac{Q_1 + Q_2}{2} + 0,3\frac{L_1 + L_2}{2}$$

onde Q_1 e Q_2 são questionários (provas/avaliações), L_1 e L_2 são listas de exercícios

Cronograma

- **15mar** – Apresentação da disciplina. Introdução. Apresentação da L_1.
- **22mar** – Prova de proficiência/validação.
- **29mar** – *Dia não letivo*
- **05abr** – Notação assintótica. Indução matemática.
- **12abr** – Indução matemática. Recorrências. Divisão e conquista.
- **19abr** – Divisão e conquista. Ordenação.
- **26abr** – Ordenação em tempo linear. Multiplicação de inteiros. Multiplicação de matrizes.
- **03mai** – Estatística de ordem. Dúvidas.
- **10mai** – Primeira avaliação Q_1. Entrega de L_1.
- **17mai** – Grafos. Buscas. Apresentação da L_2.
- **24mai** – Algoritmos gulosos.
- **31mai** – *Dia não letivo*
- **07jun** – *Semana Acadêmica do PPGCC.*
- **14jun** – Programação dinâmica.
- **21jun** – NP-Completude e reduções.
- **28jun** – Algoritmos aproximados e heurísticas.
- **05jul** – Segunda avaliação Q_2. Entrega de L_2.
- **12jul** – Dúvidas e fechamento.

Bibliografia

- DE SANTIAGO, R. Anotações para a Disciplina de Projeto e Análise de Algoritmos, 2020. Disponível em `http://www.inf.ufsc.br/~r.santiago/downloads/INE410104.pdf`
- SKIENA, S. S. The Algorithm Design Manual, Springer, 2ª Edição, London: Springer, 2008. DOI: `https://doi.org/10.1007/978-1-84800-070-4`
- JUNGNICKEL, D. Graphs, Networks and Algorithms, 3ª Edição, Berlin: Springer, 2008. DOI: `https://doi.org/10.1007/978-3-540-72780-4`
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R.L., STEIN, C., Introduction to Algorithms. 3*rd* edition, The MIT Press, 2009. (CLRS)

Bibliografia (cont...)

- S. Dasgupta, C.H. Papadimitriou, U.V. Vazirani, Algorithms, 1^{st} edition, McGraw-Hill, 2006.
- Jon Kleinberg, Éva Tardos, Algorithm Design, 1^{st} edition, Pearson, 2005.
- H.R. Lewis, C.H. Papadimitriou, Elementos de Teoria da Computação, 2 a Edição, Bookman, 2000.
- T.A. Sudkamp, Languages and Machines, Addison-Wesley, 1988.
- N.C. Ziviani, Projeto de Algoritmos com Implementações em Java e C++, Thompson Learning, 2007.

Introdução

O que veremos nesta disciplina?

- Como provar a “corretude” de um algoritmo
- Estimar a quantidade de recursos (tempo, memória) de um algoritmo $=$ análise de complexidade
- Técnicas e idéias gerais de projeto de algoritmos: indução, divisão-e-conquista, programação dinâmica, algoritmos gulosos etc
- Tema recorrente: natureza recursiva de vários problemas
- A dificuldade intrínseca de vários problemas: inexistência de soluções eficientes

Algoritmos

O que é um algoritmo (computacional) ?

Informalmente, um algoritmo é um procedimento computacional bem definido que:

- recebe um conjunto de valores como entrada,
- produz um conjunto de valores como saída,
- através de uma sequência de passos em um modelo computacional.

Equivalentemente, um algoritmo é uma ferramenta para resolver um problema computacional. Este problema define a relação precisa que deve existir entre a entrada e a saída do algoritmo.

Exemplos de problemas: teste de primalidade

Problema: determinar se um dado número é primo.

Exemplo:

Entrada: 9411461

Saída: É primo.

Exemplo:

Entrada: 8411461

Saída: Não é primo.

Exemplos de problemas: ordenação

Definição: um vetor $A[1 \dots n]$ é crescente se $A[1] \leq \dots \leq A[n]$.

Problema: rearranjar um vetor $A[1 \dots n]$ de modo que fique 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

Instância de um problema

Uma instância de um problema é um conjunto de valores que serve de entrada para esse.

Exemplo:
Os números 9411461 e 8411461 são instâncias do problema de primalidade.

Exemplo:
O vetor

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

é uma instância do problema de ordenação.

Dificuldade intrínseca de problemas

- Infelizmente, existem certos problemas para os quais **não se conhece** algoritmos eficientes capazes de resolvê-los. Exemplos são os problemas $\mathcal{NP}$-completos.

  Curiosamente, **não foi provado** que tais algoritmos não existem! Interprete isso como um desafio para inteligência humana.

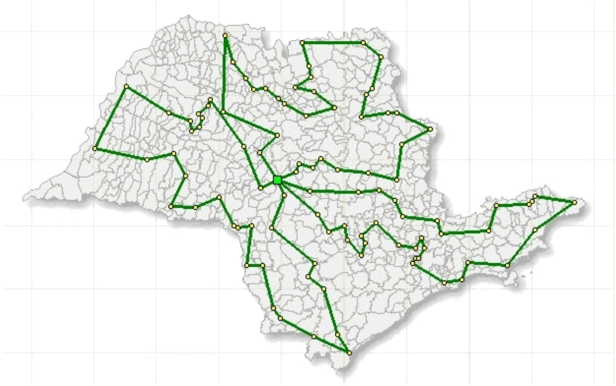
- Esses problemas tem a característica notável de que se um deles admitir um algoritmo "eficiente" então todos admitem algoritmos "eficientes".

- Por que devo me preocupar com problemas $\mathcal{NP}$-sei-lá-o-quê?

  Problemas dessa classe surgem em inúmeras situações práticas, como problemas $\mathcal{NP}$-difíceis.

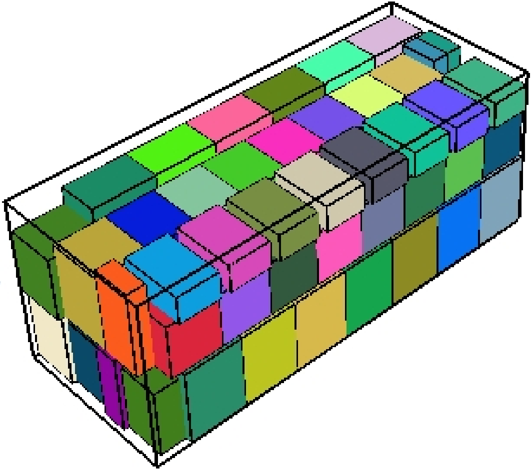
Dificuldade intrínseca de problemas

Exemplo de problema $\mathcal{NP}$-difícil: calcular as rotas dos caminhões de entrega de uma distribuidora de bebidas em São Paulo, minimizando a distância percorrida. (vehicle routing)



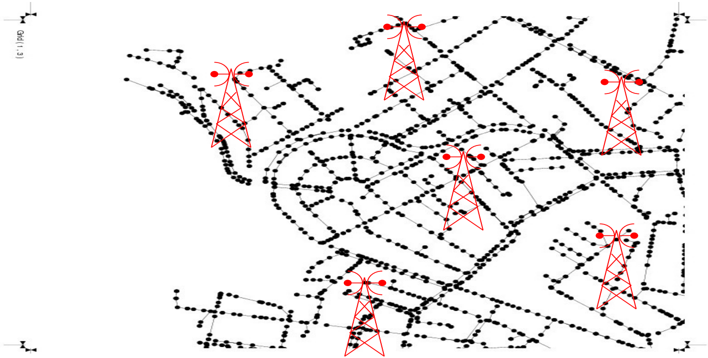
Dificuldade intrínseca de problemas

Exemplo de problema $\mathcal{NP}$-difícil: calcular o número mínimo de *containers* para transportar um conjunto de caixas com produtos. (bin packing 3D)



Dificuldade intrínseca de problemas

Exemplo de problema $\mathcal{NP}$-difícil: calcular a localização e o número mínimo de antenas de celulares para garantir a cobertura de uma certa região geográfica. (facility location)



e muito mais...

É importante saber indentificar quando estamos lidando com um problema $\mathcal{NP}$-difícil!

Algoritmos e tecnologia

- Condição ideal (irreal): os computadores têm velocidade de processamento e memória infinita. Neste caso, qualquer algoritmo é igualmente bom e esta disciplina é inútil!

- O mundo real: há computadores com velocidade de processamento na ordem de bilhões de instruções por segundo e trilhões de bytes em memória.

- Mas ainda assim temos uma limitação na velocidade de processamento e memória dos computadores.

Neste caso faz **muita** diferença ter um **bom** algoritmo.

Exemplo: **ordenação de um vetor de 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 $50n \log n$ instruções.

Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de um 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

E se tivermos os tais problemas $\mathcal{NP}$-difíceis ?

$f(n)$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
n	$2{,}0\times10^{-11}$seg	$4{,}0\times10^{-11}$seg	$6{,}0\times10^{-11}$seg	$8{,}0\times10^{-11}$seg	$1{,}0\times10^{-10}$seg
n^2	$4{,}0\times10^{-10}$seg	$1{,}6\times10^{-9}$seg	$3{,}6\times10^{-9}$seg	$6{,}4\times10^{-9}$seg	$1{,}0\times10^{-8}$seg
n^3	$8{,}0\times10^{-9}$seg	$6{,}4\times10^{-8}$seg	$2{,}2\times10^{-7}$seg	$5{,}1\times10^{-7}$seg	$1{,}0\times10^{-6}$seg
n^5	$2{,}2\times10^{-6}$seg	$1{,}0\times10^{-4}$seg	$7{,}8\times10^{-4}$seg	$3{,}3\times10^{-3}$seg	$1{,}0\times10^{-2}$seg
2^n	$1{,}0\times10^{-6}$seg	1,0seg	13,3dias	$1{,}3\times10^{5}$séc	$1{,}4\times10^{11}$séc
3^n	$3{,}4\times10^{-3}$seg	140,7dias	$1{,}3\times10^{7}$séc	$1{,}7\times10^{19}$séc	$5{,}9\times10^{28}$séc

Supondo um computador com velocidade de 1 Terahertz (mil vezes mais rápido que um computador de 1 Gigahertz).

Algoritmos e tecnologia

E se usarmos um super-computador para resolver os problemas $\mathcal{NP}$-difíceis ?

Fixando o tempo de execução: Não iremos resolver problemas muito maiores.

$f(n)$	Computador atual	100×mais rápido	1000×mais rápido
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$

$$\frac{N_2^2}{v} = t \quad ; \quad \frac{x^2}{1000v} = t$$

$$\frac{x^2}{1000v} = \frac{N_2^2}{v} \quad \Rightarrow \quad x = \sqrt{1000N_2^2} \quad \Rightarrow \quad x = 31,6N_2$$

Algoritmos e tecnologia – Conclusões

- O uso de um algoritmo adequado pode levar a ganhos extraordinários de desempenho.
- Isso pode ser tão importante quanto o projeto de *hardware*.
- A melhora obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.
- As melhorias nos algoritmos produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, etc).

Descrição de algoritmos

Podemos descrever um algoritmo de várias maneiras:

- usando uma linguagem de programação de alto nível: `C`, `Pascal`, `Java` etc
- implementando-o em linguagem de máquina diretamente executável em *hardware*
- em português
- em um pseudo-código de alto nível, como no livro do CLRS

Usaremos essencialmente as duas últimas alternativas nesta disciplina.

Exemplo de pseudo-código

Algoritmo ORDENA-POR-INSERÇÃO: rearranja um vetor $A[1\,.\,.\,n]$ de modo que fique crescente.

```
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
```

Corretude de algoritmos

- Um algoritmo (que resolve um determinado problema) é determinístico se, para toda instância do problema, ele pára e devolve uma resposta correta.
- Algoritmos probabilísticos são algoritmos que utilizam passos probabilísticos (como obter um número aleatório).

  Estes algoritmos podem errar ou gastar muito tempo, mas neste caso, queremos que a probabilidade de errar ou de executar por muito tempo seja muuuuito pequena.
- O curso será focado principalmente em algoritmos determinísticos, mas veremos alguns exemplos de algoritmos probabilísticos.

Complexidade de algoritmos

- Em geral, não basta saber que um dado algoritmo pára. Se ele for muito leeeeeeeeeeeento terá pouca utilidade.
- Queremos projetar/desenvolver algoritmos eficientes (rápidos).
- Mas o que seria uma boa medida de eficiência de um algoritmo?
- Não estamos interessados em quem programou, em que linguagem foi escrito e nem qual a máquina foi usada!
- Queremos um critério uniforme para comparar algoritmos.

Modelo Computacional

- Uma possibilidade é definir um modelo computacional de um máquina.
- O modelo computacional estabelece quais os recursos disponíveis, as instruções básicas e quanto elas custam ($=$ tempo).
- Dentre desse modelo, podemos 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.

Máquinas RAM

Salvo mencionado o contrário, usaremos o Modelo Abstrato RAM (Random Access Machine):

- simula máquinas convencionais (de verdade),
- possui um único processador que executa instruções seqüencialmente,
- tipos básicos são números inteiros e reais,
- há um limite no tamanho de cada *palavra de memória*: se a entrada tem "tamanho" n, então cada inteiro/real é representado por $c \log n$ bits onde $c \geq 1$ é uma contante.
- Note que não podemos representar números reais, a menos de aproximações. Assim, não poderemos representar π, $\sqrt{2}$ etc, de maneira exata.

  Isto é razoável?

Máquinas RAM

- executa operações aritméticas (soma, subtração, multiplicação, divisão, piso, teto), comparações, movimentação de dados de tipo básico e fluxo de controle (teste *if/else*, chamada e retorno de rotinas) em **tempo constante**,
- Certas operações ficam em uma **zona cinza**, por exemplo, exponenciação,
- veja maiores detalhes do modelo RAM no CLRS.

Tamanho da entrada

Problema: Primalidade

Entrada: inteiro n

Tamanho: número de bits de $n \approx \lg n = \log_2 n$

Problema: Ordenação

Entrada: vetor $A[1 \dots n]$

Tamanho: $n \lg U$ onde U é o maior número em $A[1 \dots n]$

Medida de complexidade e eficiência de algoritmos

- A complexidade de tempo ($=$ eficiência) de um algoritmo é o número de instruções básicas que ele executa em função do tamanho da entrada.
- Adota-se uma “atitude pessimista” e faz-se uma análise de pior caso.
  Determina-se o tempo máximo necessário para resolver uma instância de um certo tamanho.
- Além disso, a análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE $=$ análise assintótica.

Medida de complexidade e eficiência de algoritmos

- Um algoritmo é chamado eficiente se a função que mede sua complexidade de tempo é limitada por um **polinômio** no tamanho da entrada.
  Por exemplo: n, $3n-7$, $4n^2$, $143n^2-4n+2$, n^5.
- Mas por que polinômios?
  Resposta padrão: (polinômios são funções bem "comportadas").

Vantagens do método de análise proposto

- O modelo RAM é robusto e permite prever o comportamento de um algoritmo para instâncias GRANDES.
- O modelo permite comparar algoritmos que resolvem um mesmo problema.
- A análise é mais robustas em relação às evoluções tecnológicas .

Desvantagens do método de análise proposto

- Fornece um limite de complexidade pessimista sempre considerando o pior caso.
- Em uma aplicação real, nem todas as instâncias ocorrem com a mesma freqüência e é possível que as "instâncias ruins" ocorram raramente.
- Não fornece nenhuma informação sobre o comportamento do algoritmo no caso médio.
- A análise de complexidade de algoritmos no caso médio é complicada e depende do conhecimento da distribuição das instâncias.

Começando a trabalhar

Ordenação

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 estudando o algoritmo de ordenação baseado no método de inserção.

Inserção em um vetor ordenado

1						j				n
20	25	35	40	44	55	38	99	10	65	50

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

1						j				n
20	25	35	38	40	44	55	99	10	65	50

- Agora $A[1 \dots j]$ está ordenado.

Como fazer a inserção

$chave = 38$

1					i	j				n
20	25	35	40	44	55	38	99	10	65	50

1				i		j				n
20	25	35	40	44		55	99	10	65	50

1			i			j				n
20	25	35	40		44	55	99	10	65	50

1		i				j				n
20	25	35		40	44	55	99	10	65	50

1		i				j				n
20	25	35	38	40	44	55	99	10	65	50

Ordenação por inserção

chave	1							*j*			*n*
99	20	25	35	38	40	44	55	99	10	65	50

chave	1							*j*			*n*
99	20	25	35	38	40	44	55	99	10	65	50

chave	1								*j*		*n*
10	20	25	35	38	40	44	55	99	10	65	50

chave	1								*j*		*n*
10	10	20	25	35	38	40	44	55	99	65	50

Ordenação por inserção

chave	1									*j*	*n*
65	10	20	25	35	38	40	44	55	99	65	50

chave	1									*j*	*n*
65	10	20	25	35	38	40	44	55	65	99	50

chave	1										*j*
50	10	20	25	35	38	40	44	55	65	99	50

chave	1										*j*
50	10	20	25	35	38	40	44	50	55	65	99

Ordena-Por-Inserção

Pseudo-código

ORDENA-POR-INSERÇÃO(A, n)

para $j \leftarrow 2$ **até** n **faça**
    $chave \leftarrow A[j]$
    $\triangleright$ Insere $A[j]$ no subvetor ordenado $A[1..j-1]$
    $i \leftarrow j - 1$
    enquanto $i \geq 1$ **e** $A[i] > chave$ **faça**
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow chave$

Análise do algoritmo

O que é importante analisar ?

- Finitude: o algoritmo pára?
- Corretude: o algoritmo faz o que promete?
- Complexidade de tempo: quantas intruções são necessárias no pior caso para ordenar os n elementos?

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**.

Ordena-Por-Inserção

ORDENA-POR-INSERÇÃO(A, n)

para $j \leftarrow 2$ **até** n **faça**
    $chave \leftarrow A[j]$
    ▷ Insere $A[j]$ no subvetor ordenado $A[1..j-1]$
    $i \leftarrow j - 1$
    enquanto $i \geq 1$ **e** $A[i] > chave$ **faça**
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] \leftarrow chave$

O que falta fazer ?

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

Invariantes 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. . .

Exemplo de invariante

ORDENA-POR-INSERÇÃO(A, n)

1 **para** $j \leftarrow 2$ **até** n **faça**
2 $\quad$ *chave* $\leftarrow A[j]$
3 $\quad$ $\triangleright$ Insere $A[j]$ no subvetor ordenado $A[1..j-1]$
4 $\quad$ $i \leftarrow j - 1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] >$ *chave* **faça**
6 $\quad\quad$ $A[i+1] \leftarrow A[i]$
7 $\quad\quad$ $i \leftarrow i - 1$
8 $\quad$ $A[i+1] \leftarrow$ *chave*

Invariante principal de ORDENA-POR-INSERÇÃO: (i1)

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

Corretude de algoritmos por invariantes

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!

Corretude da ordenação por inserção

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

Invariante principal: (i1)

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

1						j				n
20	25	35	40	44	55	38	99	10	65	50

- 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 $A[1 \dots n]$ está ordenado!

Melhorando a argumentação

ORDENA-POR-INSERÇÃO(A, n)

1 **para** $j \leftarrow 2$ **até** n **faça**
2 $\quad$ $chave \leftarrow A[j]$
3 $\quad$ $\triangleright$ Insere $A[j]$ no subvetor ordenado $A[1 \dots j-1]$
4 $\quad$ $i \leftarrow j - 1$
5 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**
6 $\quad\quad$ $A[i+1] \leftarrow A[i]$
7 $\quad\quad$ $i \leftarrow i - 1$
8 $\quad$ $A[i+1] \leftarrow chave$

Um invariante mais preciso: (i1')

No começo de cada iteração do laço **para** das linhas 1–8, o subvetor $A[1 \dots j-1]$ é uma permutação ordenada do subvetor original $A[1 \dots j-1]$.

Esboço da demonstração de (i1')

1. Validade na primeira iteração: neste caso, temos $j = 2$ e o invariante simplesmente afirma que $A[1 \dots 1]$ está ordenado, o que é evidente.
2. Validade de uma iteração para a seguinte: segue da discussão anterior. O algoritmo empurra os elementos maiores que a chave para seus lugares corretos e ela é colocada no espaço vazio.

   Uma demonstração mais formal deste fato exige invariantes auxiliares para o laço interno **enquanto**.
3. Corretude do algoritmo: na última iteração, temos $j = n + 1$ e logo $A[1 \dots n]$ está ordenado com os elementos originais do vetor. Portanto, o algoritmo é correto.

Invariantes auxiliares

No início da linha 5 valem os seguintes invariantes:

(i2) $A[1\dots i]$, *chave* e $A[i+2\dots j]$ contém os elementos de $A[1\dots j]$ antes de entrar no laço que começa na linha 5.

(i3) $A[1\dots i]$ e $A[i+2\dots j]$ são crescentes.

(i4) $A[1\dots i] \leq A[i+2\dots j]$

(i5) $A[i+2\dots j] \; > \;$ *chave*.

$$\left.\begin{array}{l}\text{Invariantes (i2) a (i5)}\\ +\text{condição de parada na linha 5}\\ +\text{atribuição da linha 7}\end{array}\right\} \Longrightarrow \text{ invariante (i1')}$$

Demonstração? Mesma que antes.

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 vamos usar como tamanho de entrada a dimensão do vetor e ignorar o valores dos seus elementos (modelo RAM).
- 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.

Vamos contar ?

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

A constante 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.

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 $\qquad A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $\qquad i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $\quad A[i+1] \leftarrow$ *chave*	c_8	$n-1$

A constante 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.

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:

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

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.

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**.

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 $j = 2, \ldots, n$.

Lembre-se que:

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

e

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}.$$

Pior caso – continuação

Temos então que

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

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 assintótica de algoritmos

- Como dito anteriormente, na maior parte desta disciplina, estaremos nos concentrando na **análise de pior caso** e no **comportamento assintótico** dos algoritmos (instâncias de tamanho grande).
- O algoritmo Ordena-Por-Inserção tem como complexidade (de pior caso) uma função quadrática $an^2 + bn + c$, onde a, b, c são constantes absolutas que dependem apenas dos custos c_i.
- O estudo assintótico nos permite "jogar para debaixo do tapete" os valores destas constantes, i.e., aquilo que independe do tamanho da entrada (neste caso os valores de a, b e c).
- **Por que podemos fazer isso ?**

Análise assintótica de funções quadráticas

Considere a função quadrática $3n^2 + 10n + 50$:

n	$3n^2 + 10n + 50$	$3n^2$	Diferença percentual
64	12978	12288	5,32%
128	50482	49152	2,63%
512	791602	786432	0,65%
1024	3156018	3145728	0,33%
2048	12603442	12582912	0,16%
4096	50372658	50331648	0,08%
8192	201408562	201326592	0,04%
16384	805470258	805306368	0,02%
32768	3221553202	3221225472	0,01%

Como se vê, $3n^2$ é o termo dominante quando n é grande.

De um modo geral, podemos nos concentrar nos termos dominantes e esquecer os demais.

Notação assintótica

- Usando notação assintótica, dizemos que o algoritmo Ordena-Por-Inserção tem complexidade de tempo de pior caso $\Theta(n^2)$.
- Isto quer dizer duas coisas:
  - a complexidade de tempo é limitada (superiormente) assintoticamente por algum polinômio da forma an^2 para alguma constante a,
  - para todo n suficientemente grande, existe alguma instância de tamanho n que consome tempo pelo menos dn^2, para alguma contante positiva d.
- Mais adiante discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.

Agradecimentos

- Esses slides são fruto de um trabalho iniciado pelos Profs. Cid C. de Souza e Cândida N. da Silva. Desde então, várias partes foram motificadas, melhoradas ou colocadas ao gosto particular de cada docente. Em particular pelos Profs. Orlando Lee, Pedro J. de Rezende, Flávio K. Miyazawa e, mais recentemente, pelo Prof. Alexandre. G. Silva.
- Vários outros professores colaboraram direta ou indiretamente para a preparação do material desses slides. Agradecemos, especialmente (em ordem alfabética): Célia P. de Mello, José C. de Pina, Paulo Feofiloff, Ricardo Dahab e Zanoni Dias.
- Como foram feitas modificações de conteúdo teórico e filosófico, os erros/imprecisões que se encontram nesta versão devem ser comunicados a `alexandre.goncalves.silva@ufsc.br`.