Projeto e Análise de Algoritmos*

Introdução

Lehilton Pedrosa

Segundo Semestre de 2017

*Criado por C. de Souza, C. da Silva, O. Lee, F. Miyazawa et al.

A maior parte deste conjunto de slides foi inicialmente preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para cursos de Análise de Algoritmos. Além desse material, diversos conteúdos foram adicionados ou incorporados por outros professores, em especial por Orlando Lee e por Flávio Keidi Miyazawa. Os slides usados nessa disciplina são uma junção dos materiais didáticos gentilmente cedidos por esses professores e contêm algumas modificações, que podem ter introduzido erros.

O conjunto de slides de cada unidade do curso será disponibilizado como guia de estudos e deve ser usado unicamente para revisar as aulas. Para estudar e praticar, leia o livro-texto indicado e resolva os exercícios sugeridos.

Lehilton

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes "colaboradores" (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - Flávio Keidi Miyazawa
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Introdução à Análise de Algoritmos

Introdução

O que veremos nesta disciplina?

- Como provar a "correção" de um algoritmo
- Estimar a quantidade de recursos (tempo, memória) de um algoritmo = análise de complexidade
- Técnicas e ideias 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?

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

- recebe um conjunto de valores como entrada e
- produz um conjunto de valores como saída.

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 corresponde a uma entrada.

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.

A importância dos algoritmos para a computação

- Onde se encontram aplicações para o uso/desenvolvimento de algoritmos "eficientes"?
  - projetos de genoma de seres vivos
  - rede mundial de computadores
  - comércio eletrônico
  - planejamento da produção de indústrias
  - logística de distribuição
  - *games* e filmes
  - ...
- Humm, vamos projetar um novo *game* neste curso?

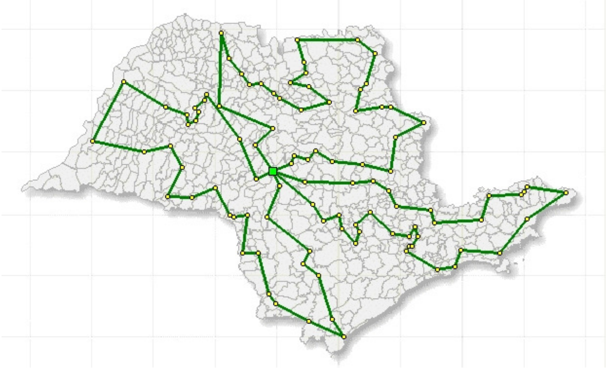
  Não!

Dificuldade intrínseca de problemas

- Infelizmente, existem certos problemas para os quais **não se conhecem** algoritmos eficientes. Um subconjunto importante desses são os chamados problemas $\mathcal{NP}$-difíceis.
  Curiosamente, **não foi provado** que tais algoritmos não existem! Interprete isso como um desafio para inteligência humana.
- Esses problemas têm 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.

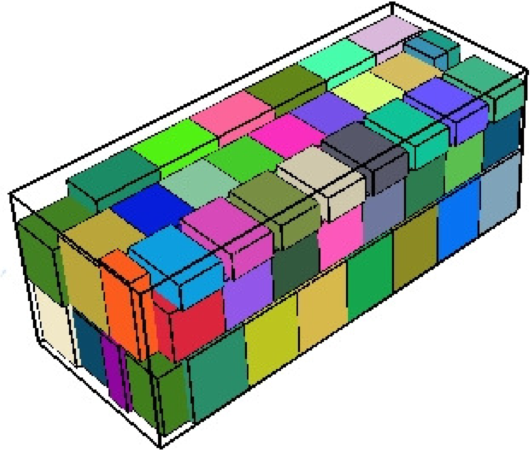
Dificuldade intrínseca de problemas

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)



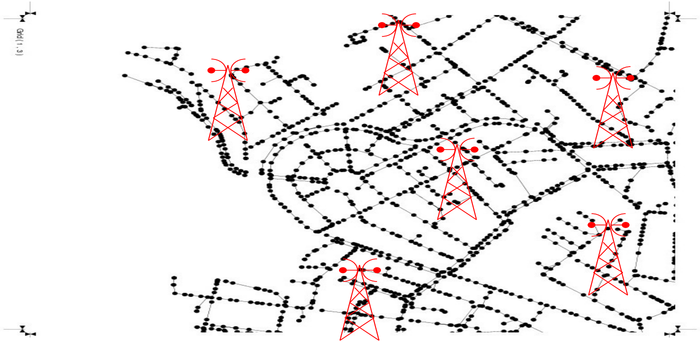
Dificuldade intrínseca de problemas

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

- O Éden: os computadores têm velocidade de processamento e memória infinita. Neste caso, qualquer algoritmo é igualmente bom e esta analisar o tempo de um algoritmo é inútil! Porém...
- O mundo real: computadores têm velocidades de processamento e memória limitadas.

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

Algoritmos e tecnologia

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 <u>dias</u> para 20 <u>minutos</u>!

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

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

Talvez queiramos usar um computador com velocidade de 1 Terahertz (mil vezes mais rápido que um computador de 1 Gigahertz)!

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

$f(n)$	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	$N_6 + 4{,}19$	$N_6 + 6{,}29$

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

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 pseudocódigo de alto nível, como no livro de CLRS

Usaremos essencialmente as duas últimas alternativas nesta disciplina.

Exemplo de pseudocódigo

Algoritmo ORDENA-POR-INSERÇÃO: rearranja um vetor $A[1\ldots n]$ 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
```

Correção de algoritmos

- Um algoritmo (para um certo problema) está correto se, para toda instância do problema, ele para e devolve uma resposta correta.
- Algoritmos aleatorizados são algoritmos que utilizam passos probabilísticos (como obter um número aleatório).

  Alguns tipos de algoritmos aleatorizados 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 um exemplo de algoritmo aleatorizado.

Complexidade de algoritmos

- Em geral, não basta saber que um dado algoritmo para. 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 sequencialmente,
- tipos básicos são números inteiros e pontos flutuantes,
- há um limite no tamanho de cada *palavra de memória*: se a entrada tem "tamanho" n, então cada inteiro/ponto flutuante é representado por $c \log n$ bits onde $c \geq 1$ é uma constante.

  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 caem 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.
- Normalmente se adota 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?
  - polinômios são funções bem "comportadas"
  - reveja a nossa tabela anterior!

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 robusta 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 frequência 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

Ordenando por inserção

chave	1							*j*			*n*
99	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

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	50	55	65	99

Ordena-Por-Inserção

Pseudocódigo

Ordena-Por-Inserção(A, n)

  para $j \leftarrow 2$ **até** n **faça**
      chave $\leftarrow A[j]$
      $\triangleright$ Insere $A[j]$ no subvetor ordenado $A[1 \dots j-1]$
      $i \leftarrow j - 1$
      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?

- Correção:
  - o algoritmo para (finitude)
  - o algoritmo faz o que promete?
- Complexidade de tempo:
  - quantas instruções são necessárias no pior caso para ordenar os n elementos?

Por enquanto só vamos somente verificar que o algoritmo para e analisar sua complexidade!

O algoritmo para

```
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 para em algum momento por causa do teste condicional $i \geq 1$.

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

Portanto, o algoritmo **para**.

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 os 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	Qnts 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	
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 é o tempo de uma 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:

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

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.

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	$3n^2 + 10n + 50$	$3n^2$
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

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 constante positiva d.
- Mais adiante discutiremos em detalhes o uso da notação assintótica em análise de algoritmos.

Projetando algoritmos

Entendendo e melhorando

Até agora:

- Vimos o algoritmo Ordena-Por-Inserção, que ordena números de maneira incremental
- Analisamos sua complexidade de pior caso e obtivemos $\Theta(n^2)$
- Vamos obter um tempo bem menor com o algoritmo de Ordenação por intercalação.
- Pra isso vamos usar projetar o algoritmo uma técnica distinta: divisão e conquista.

Ordenação por intercalação

O que significa intercalar dois (sub)vetores ordenados?

Problema: Dados $A[p \dots q]$ e $A[q+1 \dots r]$ crescentes, rearranjar $A[p \dots r]$ de modo que ele fique em ordem crescente.

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Intercalando

	p				q				r
A	22	33	55	77	99	11	44	66	88

B									

Intercalando

	k								
A									
	i								j
B	22	33	55	77	99	88	66	44	11

Intercalando

		k							
A	11								
	i							j	
B	22	33	55	77	99	88	66	44	11

Intercalando

			k						
A	11	22							

		i						j	
B	22	33	55	77	99	88	66	44	11

Intercalando

				k					
A	11	22	33						

			i					j	
B	22	33	55	77	99	88	66	44	11

Intercalando

					k				
A	11	22	33	44					

			i				j		
B	22	33	55	77	99	88	66	44	11

Intercalando

						k			
A	11	22	33	44	55				

				i			j		
B	22	33	55	77	99	88	66	44	11

Intercalando

						k			
A	11	22	33	44	55	66			

				i		j			
B	22	33	55	77	99	88	66	44	11

Intercalando

								k	
A	11	22	33	44	55	66	77		
					i	j			
B	22	33	55	77	99	88	66	44	11

Intercalando

									k
A	11	22	33	44	55	66	77	88	

					i = j				
B	22	33	55	77	99	88	66	44	11

Intercalando

A	11	22	33	44	55	66	77	88	99

					j	i			
B	22	33	55	77	99	88	66	44	11

Intercalação

Pseudocódigo

```
INTERCALA(A, p, q, r)
    para i ← p até q faça
        B[i] ← A[i]
    para j ← q + 1 até r faça
        B[r + q + 1 − j] ← A[j]
    i ← p
    j ← r
    para k ← p até r faça
        se B[i] ≤ B[j]
            então A[k] ← B[i]
                  i ← i + 1
            senão A[k] ← B[j]
                  j ← j − 1
```

Complexidade de Intercala

Entrada:

	p				q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p				q				r
A	11	22	33	44	55	66	77	88	99

Tamanho da entrada: $n = r - p + 1$

Consumo de tempo: $\Theta(n)$

Algoritmos recursivos

> "To understand recursion, we must first understand recursion."
>
> *(anônimo)*

- O que é o paradigma de **divisão-e-conquista**?
- Como mostrar a correção de um algoritmo recursivo?
- Como analisar o consumo de tempo de um algoritmo recursivo?
- O que é uma **fórmula de recorrência**?
- O que significa *resolver* uma fórmula de recorrência?

Recursão e o paradigma de divisão-e-conquista

- Um algoritmo recursivo resolve uma instância de um problema executando a si mesmo com instâncias menores deste mesmo problema.
- Algoritmos de divisão-e-conquista possuem três etapas em cada nível de recursão:
  1. **Divisão:** o problema é dividido em subproblemas semelhantes ao problema original, porém tendo como entrada instâncias de tamanho menor.
  2. **Conquista:** cada subproblema é resolvido recursivamente a menos que o tamanho de sua entrada seja suficientemente "pequeno", quando este é resolvido diretamente.
  3. **Combinação:** as soluções dos subproblemas são combinadas para obter uma solução do problema original.

Exemplo de divisão-e-conquista: *Mergesort*

- Mergesort é um algoritmo para resolver o problema de ordenação e um exemplo clássico do uso do paradigma de divisão-e-conquista. (*to merge* = intercalar)
- Descrição do Mergesort em alto nível:
  1. **Divisão**: divida o vetor com n elementos em dois subvetores de tamanho $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$, respectivamente.
  2. **Conquista**: ordene os dois vetores recursivamente usando o Mergesort;
  3. **Combinação**: intercale os dois subvetores para obter um vetor ordenado usando o algoritmo Intercala.

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

Ordenando por intercalação

	p				*q*				*r*
A	66	33	55	44	99	11	77	22	88

	p		*q*		*r*				
A	66	33	55	44	99				

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

	$p = r$								
A	66								

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

		$p = r$							
A		33							

Ordenando por intercalação

	p				q				r
A	66	33	55	44	99	11	77	22	88

	p		q		r				
A	66	33	55	44	99				

	p	q	r						
A	66	33	55						

	p	r							
A	66	33							

Ordenando por intercalação

	p				q				r
A	33	66	55	44	99	11	77	22	88

	p		q		r				
A	33	66	55	44	99				

	p	q	r						
A	33	66	55						

	p	r							
A	33	66							

Ordenando por intercalação

	p				q				r
A	33	66	55	44	99	11	77	22	88

	p		q		r				
A	33	66	55	44	99				

	p	q	r						
A	33	66	55						

Ordenando por intercalação

	p				q				r
A	33	66	55	44	99	11	77	22	88

	p		q		r				
A	33	66	55	44	99				

	p	q	r						
A	33	66	55						

			$p = r$						
A			55						

Ordenando por intercalação

	p				q				r
A	33	66	55	44	99	11	77	22	88

	p		q		r				
A	33	66	55	44	99				

	p	q	r						
A	33	66	55						

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

	p	q	r						
A	33	55	66						

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

				$p = r$					
A				44					

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

					$p = r$				
A					99				

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

				p	r				
A				44	99				

Ordenando por intercalação

	p				q				r
A	33	55	66	44	99	11	77	22	88

	p		q		r				
A	33	55	66	44	99				

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

	p		q		r				
A	33	44	55	66	99				

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

						$p = r$			
A						11			

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

							$p = r$		
A							77		

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

						p	r		
A						11	77		

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

								$p = r$	
A								22	

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

									$p = r$
A									88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

								p	r
A								22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	77	22	88

						p			r
A						11	77	22	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	22	77	88

						p			r
A						11	22	77	88

Ordenando por intercalação

	p				q				r
A	33	44	55	66	99	11	22	77	88

Ordenando por intercalação

	p				q				r
A	11	22	33	44	55	66	77	88	99

Ordenando por intercalação

	p				q				r
A	11	22	33	44	55	66	77	88	99

Mergesort

Relembrando: o objetivo é rearranjar $A[p \ldots r]$, com $p \leq r$, em ordem crescente.

```
MERGE-SORT(A, p, r)
    se p < r
        então q ← ⌊(p + r)/2⌋
              MERGE-SORT(A, p, q)
              MERGE-SORT(A, q + 1, r)
              INTERCALA(A, p, q, r)
```

	p				q				r
A	66	33	55	44	99	11	77	22	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \le r$, em ordem crescente.

```
MERGE-SORT(A, p, r)
  se p < r
     então q ← ⌊(p + r)/2⌋
           MERGE-SORT(A, p, q)
           MERGE-SORT(A, q + 1, r)
           INTERCALA(A, p, q, r)
```

	p				q				r
A	33	44	55	66	99	11	77	22	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \leq r$, em ordem crescente.

MERGE-SORT(A, p, r)
  se $p < r$
    então $q \leftarrow \lfloor (p+r)/2 \rfloor$
      MERGE-SORT(A, p, q)
      MERGE-SORT$(A, q+1, r)$
      INTERCALA(A, p, q, r)

	p				q				r
A	33	44	55	66	99	11	22	77	88

Mergesort

Relembrando: o objetivo é rearranjar $A[p \dots r]$, com $p \leq r$, em ordem crescente.

```
MERGE-SORT(A, p, r)
    se p < r
        então q ← ⌊(p + r)/2⌋
              MERGE-SORT(A, p, q)
              MERGE-SORT(A, q + 1, r)
              INTERCALA(A, p, q, r)
```

	p				q				r
A	11	22	33	44	55	66	77	88	99

Complexidade do MERGE-SORT

```
MERGE-SORT(A, p, r)
    se p < r
        então q ← ⌊(p + r)/2⌋
              MERGE-SORT(A, p, q)
              MERGE-SORT(A, q + 1, r)
              INTERCALA(A, p, q, r)
```

- Primeiro, defina o tamanho da entrada como $n := r - p + 1$
- Qual é a complexidade de MERGE-SORT?

$T(n) :=$ "o tempo de execução máximo de MERGE-SORT entre todas as instâncias de tamanho n"

- Uma instância que gasta esse tempo é um pior caso.

Complexidade do Mergesort

MERGE-SORT(A, p, r)
1 **se** $p < r$
2 **então** $q \leftarrow \lfloor (p+r)/2 \rfloor$
3 MERGE-SORT(A, p, q)
4 MERGE-SORT$(A, q+1, r)$
5 INTERCALA(A, p, q, r)

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(1)$
3	$T(\lceil n/2 \rceil)$
4	$T(\lfloor n/2 \rfloor)$
5	$\Theta(n)$

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) + \Theta(2)$$

Complexidade do Mergesort

- Obtemos o que chamamos de fórmula de recorrência:
  - É a descrição de uma função em termos de si mesma.

$$T(1) = \Theta(1)$$
$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) \quad \text{para } n = 2, 3, 4, \ldots$$

- Normalmente, algoritmos baseados em divisão-e-conquista têm complexidade $T(n)$ dada por uma recorrência.
- Basta então resolver a recorrência! Mas, o que significa resolver uma recorrência?
  - Significa encontrar uma "fórmula fechada" para $T(n)$.
  - No caso, $T(n) = \Theta(n \lg n)$.
  - Assim, o tempo do Mergesort é $\Theta(n \lg n)$ no pior caso.
- Veremos mais tarde como resolver recorrências.