Algoritmos

Pedro Hokama

Fontes

- [*clrs*] Algoritmos: Teoria e Prática (Terceira Edição) Thomas H. Cormen, Charles Eric Leiserson, Ronald Rivest, Ronald L. Rivest e Clifford Stein.
- [*timr*] Algorithms Illuminated Series, Tim Roughgarden

Apresentação Baseada:

- Stanford Algorithms
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt7Q0xr1PIAriY5623cKiH7V`
  `https://www.youtube.com/playlist?list=PLXFMmlk03Dt5EMI2s2WQBsLsZl7A5HEK6`
- Conjunto de Slides dos Professores Cid C. de Souza, Cândida N. da Silva, Orlando Lee, Pedro J. de Rezende

O que é um algoritmo?

- Um conjunto de passos bem definidos para resolver um problema computacional.

Exemplos:

- Você tem vários números e precisa deles ordenados.
- Você tem um mapa e precisa encontrar o menor caminho entre uma origem e um destino.
- Você tem várias tarefas distintas, cada uma com uma data de entrega e cada uma demora um certo tempo para ser realizada. E você quer completar todas sem atraso.

Etimologia

- Uma provável origem da palavra algoritmo é a latinização do nome do persa *Muhammad ibn Musa al-Khwarizmi*
- Matemático, astronomo, geógrafo, e acadêmico na *House of Wisdom* em Bagdá.
- Por volta de 825 ele escreveu um tratado sobre o sistema numérico Arábico-Hindu que foi traduzido para o Latin no século 12 com o título *Algoritmi de numero Indorum*. Com o sentido de *Algoritmi (al-Khwarizmi) sobre os números dos Hindus*.

Por que estudar algoritmos?

Entender as bases de algoritmos e estruturas de dados é essencial para fazer um trabalho sério em qualquer ramo da ciência da computação.
Exemplos:

- Roteamento de redes, utiliza princípios de algoritmos de caminhos mínimos
- Criptografia de chave pública se baseia em algoritmos de teoria dos números
- Computação gráfica precisa de algoritmos geométricos.
- Bancos de Dados se baseiam em árvores balanceadas.
- Biologia computacional usa programação dinâmica para medir a similaridade entre genomas.
- etc, etc, etc...

Por que estudar algoritmos?

São a chave fundamental para inovação tecnológica moderna. Um exemplo importante:

- Os mecanismos de busca se baseiam nos fundamentos de algorítimos para computar de forma eficiente a relevância de várias páginas web para uma dada consulta.
- Desses o algoritmo mais famoso é o algoritmo *page rank* usado pelo Google.
- Talvez você já tenha pensado que a evolução dos hardwares é a única responsável pelo progresso da tecnologia. Porém não é bem assim.
- O próximo slide apresenta um trecho do relatório do conselho de ciência e tecnologia para a Casa Branca de dezembro de 2010: **Progress in Algorithms Beats Moore's Law**

Everyone knows Moore's Law – a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years. Fewer people appreciate the extraordinary innovation that is needed to translate increased transistor density into improved system performance. This effort requires new approaches to integrated circuit design, and new supporting design tools, that allow the design of integrated circuits with hundreds of millions or even billions of transistors, compared to the tens of thousands that were the norm 30 years ago. It requires new processor architectures that take advantage of these transistors, and new system architectures that take advantage of these processors. It requires new approaches for the system software, programming languages, and applications that run on top of this hardware. All of this is the work of computer scientists and computer engineers.
Even more remarkable – and even less widely understood – is that **in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed**.
The algorithms that we use today for speech recognition, for natural language translation, for chess playing, for logistics planning, have evolved remarkably in the past decade. It's difficult to quantify the improvement, though, because it is as much in the realm of quality as of execution time.
In the field of numerical algorithms, however, the improvement can be quantified. Here is just one example, provided by Professor Martin Grötschel of Konrad-Zuse-Zentrum für Informationstechnik Berlin. Grötschel, an expert in optimization, observes that a benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day. Fifteen years later – in 2003 – this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million. Of this, a factor of roughly 1,000 was due to increased processor speed, whereas a factor of roughly 43,000 was due to improvements in algorithms! Grötschel also cites an algorithmic improvement of roughly 30,000 for mixed integer programming between 1991 and 2008.
The design and analysis of algorithms, and the study of the inherent computational complexity of problems, are fundamental subfields of computer science.

Por que estudar algoritmos?

- É desafiador!
  - Existem uma miríade de Algoritmos e Técnicas de Projeto de algoritmos que mal imaginamos. Veremos algumas das mais importantes nessa disciplina.
- É divertido!

Multiplicação de Inteiros

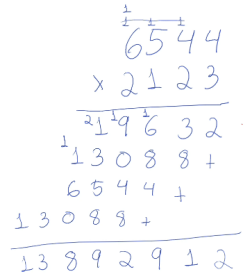
Vamos definir o problema da multiplicação de inteiros:

Multiplicação de Inteiros

Dado dois inteiros x e y de n dígitos cada. Encontrar o produto $x \cdot y$.

- Exemplo: 6544 e 2123
- Você provavelmente aprendeu no ensino fundamental a fazer essa conta. De fato o que você aprendeu foi um algoritmo que é capaz de resolver esse problema.

Multiplicação de 6544 e 2123 (!)

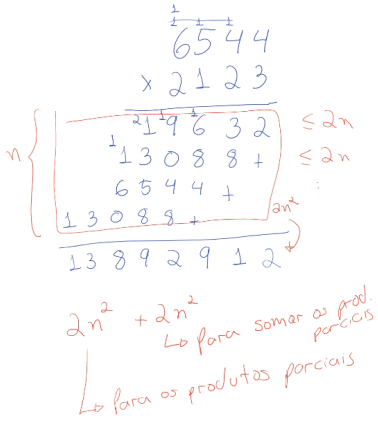


Resposta: 13892912

Multiplicação de Inteiros

- Intuitivamente você sabe que esse algoritmo está CORRETO, ou seja, dados quaisquer números x e y seguindo adequadamente os passos do algoritmo você terminaria com o produto de x e y.
- Mas quantas "contas"elementares tivemos que fazer para realizar essa multiplicação?
- Vamos chamar de operações elementares a soma ou multiplicação de números com um único digito. Quantas operações básicas são necessárias nesse algoritmo?

Número de operações na multiplicação de 6544 e 2123 (!)



Resposta: $4n^2$

Multiplicação de Inteiros

- **Conclusão:** Precisamos aproximadamente de uma constante (aprox. 4) vezes n^2 operações para realizar essa multiplicação. Então o que acontece se você dobrar o número de dígitos? E se você quadruplicar o número de dígitos?
- **Será que podemos fazer melhor?** De fato essa é a pergunta que faremos constantemente.

  Perhaps the most important principle for the good algorithm designer is to refuse to be content. (Aho, Hopcroft e Ullman. The Design and Analysis of Computer Algorithms, 1974)
- Veremos a seguir um algoritmo diferente (melhor?) para multiplicar inteiros.

Karatsuba

- Anatolii Alexeievitch Karatsuba, matemático nascido na União Soviética (1937-2008).
- Passou a maior parte da vida acadêmica na Faculdade de Mecânica e Matemática da Universidade Estadual de Moscow
- Descobriu em 1960 e publicou em 1962 um novo método que usa o paradigma de divisão e conquista para multiplicar números grandes.

Algoritmo de Karatsuba

Multiplicação de Inteiros

Dado dois inteiros x e y de n dígitos cada. Encontrar o produto $x \cdot y$.

- Dividir x em duas partes, digamos a e b de $n/2$ dígitos.

$$x = a \cdot 10^{n/2} + b, \text{ no nosso exemplo } 6544 = 65 \cdot 10^2 + 44$$

- Dividir y em duas partes, digamos c e d de $n/2$ dígitos.

$$y = c \cdot 10^{n/2} + d$$

$$x = a \cdot 10^{n/2} + b$$
$$y = c \cdot 10^{n/2} + d$$

$$\begin{aligned} x \cdot y &= (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d) \\ &= ac \cdot 10^{n/2} \cdot 10^{n/2} + ad \cdot 10^{n/2} + bc \cdot 10^{n/2} + bd \\ &= ac \cdot 10^{n} + 10^{n/2}(ad + bc) + bd \end{aligned}$$

- Então se calcularmos ac, ad, bc, e bd que são todas multiplicações de números com $n/2$ dígitos (portanto menor que os x e y com n dígitos) podemos fazer algumas operações simples e obter o resultado para o problema original.
- Obviamente podemos calcular ac, ad, bc, e bd com 4 chamadas recursivas para esse algoritmo. O caso base é quando temos números de 1 dígito que sabemos calcular.

$$x \cdot y = ac \cdot 10^n + 10^{n/2}(ad + bc) + bd$$

$$6544 \cdot 2123$$

$$a = 65, b = 44, c = 21, d = 23$$

$$65 \cdot 21 \cdot 10^n + 10^{n/2}(65 \cdot 23 + 44 \cdot 21) + 44 \cdot 23$$

$$65 \cdot 21 \cdot 10^4 + 10^2(65 \cdot 23 + 44 \cdot 21) + 44 \cdot 23$$

$$1365 \cdot 10^4 + 10^2(1495 + 924) + 1012$$

$$1365 \cdot 10^4 + 10^2(2419) + 1012$$

$$\begin{array}{r} 13650000 \\ 241900 \\ 1012 \\ \hline 13892912 \end{array}$$

- Será que esse algoritmo é mais rápido (em termos de número de operações elementares) do que o algoritmo do primário?
- Veremos nessa disciplina como analisar esse tipo de algoritmo
- Por enquanto acredite que o número de operações desses algoritmo é a mesma coisa que o algoritmo do primário. :(
- Mas perceba o seguinte, na formula

$$x \cdot y = ac \cdot 10^n + 10^{n/2}(ad + bc) + bd$$

não estamos de fato interessados no valor de ad e bc, mas apenas na sua soma.

- Então será que ao invés de obter 4 valores através de chamadas recursivas, podemos obter apenas os 3 que nos interessa?

Carl Friedrich Gauss

- Johann Carl Friedrich Gauss (1777 - 1855) foi um matemático, astrônomo e físico alemão.
- Dentre diversas contribuições para a ciência, Gauss notou que no produto de dois números complexos

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

parece envolver a multiplicação de 4 números.

- Mas de fato pode ser feito com 3 multiplicações, uma vez que

$$ad + bc = (a + b)(c + d) - ac - bd$$

Algoritmo de Karatsuba

$$x \cdot y = ac \cdot 10^n + 10^{n/2}(ad + bc) + bd$$

- Observe que:

$$ad + bc = (a + b)(c + d) - ac - bd$$

- Dessa forma podemos fazer o seguinte:
  1. calcular ac
  2. calcular bd
  3. calcular $(a + b)(c + d)$
  4. calcular $e = (a + b)(c + d) - ac - bd$
  5. somar $a.c.10^n + b.d + e.10^{n/2}$

1. calcular ac
2. calcular bd
3. calcular $(a+b)(c+d)$
4. calcular $e = (a+b)(c+d) - ac - bd$
5. somar $a.c.10^n + e.10^{n/2} + b.d$

$$6544 \cdot 2123$$

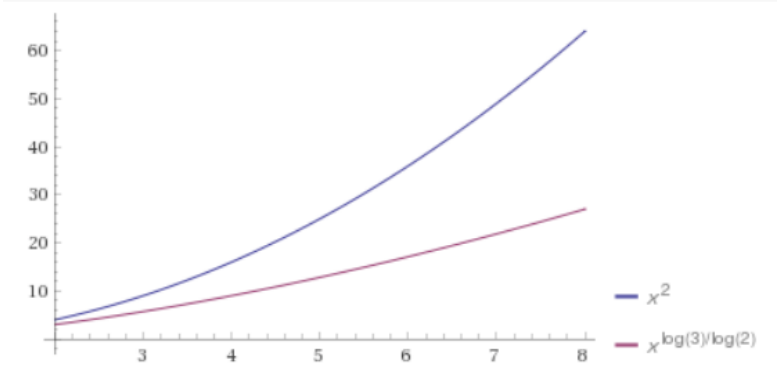
$$a = 65, b = 44, c = 21, d = 23$$

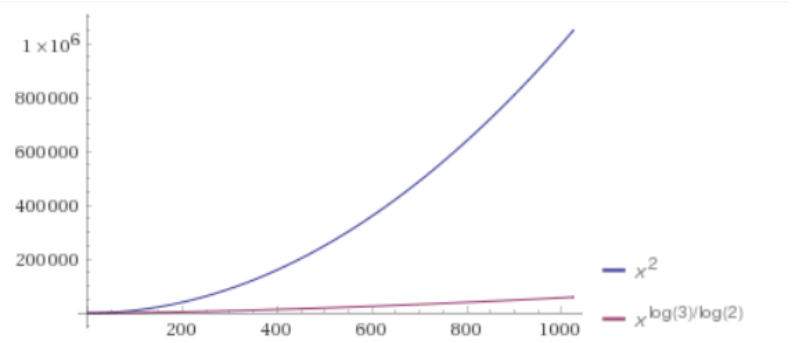
1. $ac = 1365$
2. $bd = 1012$
3. $(a+b)(c+d) = (109 \cdot 44) = 4796$
4. $e = 4796 - 1365 - 1012 = 2419$
5. $13650000 + 241900 + 1012$

$$13892912$$

Algoritmo de Karatsuba

- Pelas provas que fizemos você deve estar convencido de que o algoritmo de fato funciona!
- Perceba que como um problema tão comum quanto a multiplicação de inteiros pode ter algoritmos diferentes para resolve-los.
- Mas será que esse algoritmo é mais rápido que o do primário, ou do que a versão que faz 4 chamadas recursivas?
- Veremos com detalhes essa análise posteriormente, mas por enquanto acredite, ao invés de cn^2 esse algoritmo faz $c'n^{\log_2 3} = c'n^{1.586}$





Objetivos da Disciplina

- Familiarizar o aluno com o vocabulário da Ciência da Computação
  - Notação O, *Big O*, Ó grande, Ózão. E seus colegas.
  - Encontrar uma forma de comparar dois algoritmos de forma simples porém precisa.
- Técnicas de Projeto de algoritmos:
  - Divisão e Conquista
  - Algoritmos Gulosos
  - Aleatorização de algoritmos
  - Programação dinâmica
- Algoritmos em Grafos
- Estruturas de dados

Habilidades que você vai adquirir

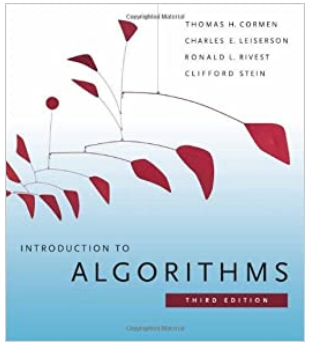
- Tornar-se um programador melhor
- Afiar a sua habilidade matemática e analítica
- Pensar de forma algorítmica
- Familiaridade com a literatura de ciência da computação, algoritmos clássicos, teorema fundamentais, personalidades importantes.
- Vai ajudar a passar em entrevistas de emprego!

Pré-requisitos

- Saber programação, ser capaz de entender uma ideia e transforma-la em código
- Estruturas de dados simples: Vetor, Listas, Pilha, Fila, Heaps, Árvores.
- Matemática discreta: Conjuntos, Números, Provas Matemáticas.

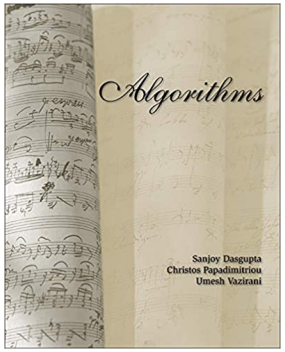
Referências

- Introduction to Algorithms (3rd Edition)
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein
- Tem edição em português.



Referências

- Algorithms
- Sanjoy Dasgupta, Christos Papadimitriou e Umesh Vazirani
- Tem uma versão livre na internet.
- Tem edição em português.

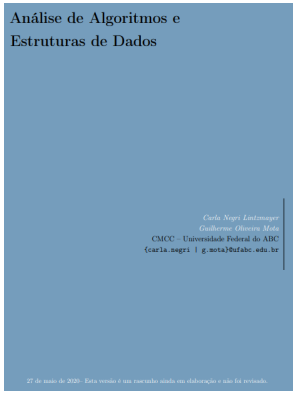


Referências

- Algorithms Illuminated
- Tim Roughgarden
- Tem uma série de vídeo aulas que cobrem o material do livro.

Referências

- Análise de Algoritmos e Estruturas de Dados
- Carla Negri Lintzmayer e Guilherme Oliveira Mota
- Em português e tem versão online.
- `http://professor.ufabc.edu.br/~carla.negri/cursos/materiais/Livro-Analise.de.Algoritmos.pdf`



John von Neumann

- John von Neumann (1903 - 1957) nascido na Hungria e de origem judaica. Naturalizado americano em 1937.
- Foi membro do Instituto de Estudos Avançados de Princeton, Nova Jérsei, do qual fazia parte Albert Einstein, Kurt Gödel e vários outros grandes cientistas.
- Dentre diversas contribuições para a matemática, ciência da computação, física, etc. Neumann descreveu em 1945 o algoritmo MergeSort.

MergeSort (Ordenação por Intercalação)

Por que veremos um algoritmo de 1945?

- É o algoritmo de escolha ainda hoje por ser realmente eficiente
- Muito melhor do que a complexidade quadrática (InsertionSort, SelectionSort, etc)

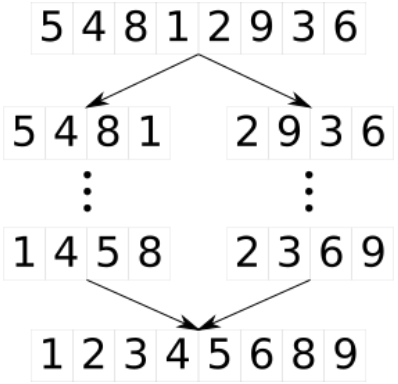
Por que veremos novamente o MergeSort?

- É claro sobre o método de divisão e conquista
- Vai preparar vocês para análises de complexidade mais complicadas.

Problema da Ordenação

Dado um arranjo de n inteiros distintos, encontrar o arranjo $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ que contenha os mesmos elementos mas ordenados de maneira não decrescente, ou seja, $\pi_i \leq \pi_j$ para qualquer $i < j$ e $i, j \in \{1, \ldots, n\}$.

MergeSort



Pseudo-Código para o MergeSort:

Algoritmo 1: MergeSort

Entrada: Um arranjo com n
Saída: Um arranjo com os mesmos números ordenados

```
1 se n ≥ 2 então
2     A = Recursivamente ordenar a primeira metade do arranjo de entrada;
3     B = Recursivamente ordenar a segunda metade do arranjo de entrada;
4     C = Intercalar (Merge) as duas partes ordenadas A e B em uma;
5 devolva C;
```

Pseudo-Código para o Merge:

Algoritmo 2: Merge

Entrada: A e B arranjos ordenados com $m/2$
Saída: Arranjo C de tamanho m com os mesmos elementos de A e B mas ordenados

```
1  i = 1;
2  j = 1;
3  para k de 1 até m faça
4      se A[i] < B[j] então
5          C[k] = A[i];
6          i++;
7      senão
8          C[k] = B[j];
9          j++;
10 devolva C;
11 Exercício: verificar finalizações (se A ou B acabarem etc).
```


MergeSort

- Qual o tempo de execução do MergeSort? Quantas operações básicas faz o MergeSort? Qual o número de linhas de código executadas pelo MergeSort?
- Primeiro nos perguntaremos qual o tempo de execução do Merge?

Pseudo-Código para o Merge:

Algoritmo 3: Merge

Entrada: A e B arranjos ordenados com $m/2$
Saída: Arranjo C de tamanho m com os mesmos elementos de A e B mas ordenados

```
1  i = 1;
2  j = 1;
3  para k de 1 até m faça
4  |   se A[i] < B[j] então
5  |   |   C[k] = A[i];
6  |   |   i++;
7  |   senão
8  |   |   C[k] = B[j];
9  |   |   j++;
10 |   fim
11 fim
12 devolva C;
13 Exercício: verificar finalizações (se A ou B acabarem etc).
```

Um incremento de k e uma comparação

Essas 2 ou essas 2

m vezes

Um total de 4m + 2 ⩽ 6m

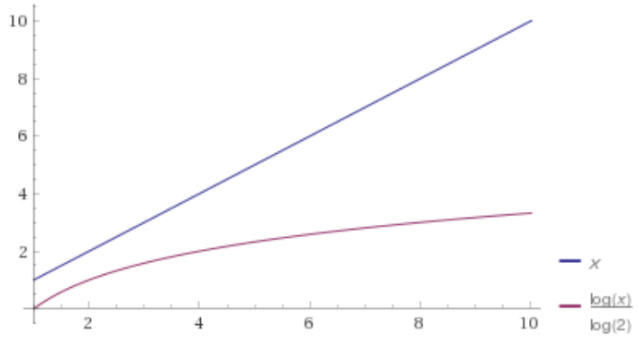
- Analisar o MergeSort é um pouco mais desafiador pois cada problema faz 2 chamadas recursivas, causando uma explosão de subproblemas. A boa notícia é que cada vez que dividimos um subproblema em dois, cada um deles tem metade do tamanho.
- De fato é esse equilíbrio entre a quantidade de subproblemas e o tamanho de cada subproblema que vai ditar a complexidade de um algoritmo recursivo.

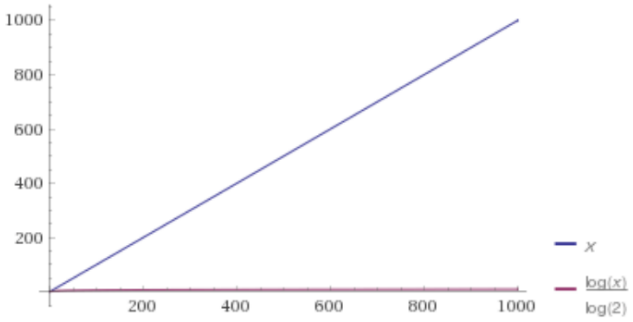
Complexidade do MergeSort

Teorema

MergeSort exige menos de $6n \log_2 n + 6n$ *operações para ordenar* n *números.*

- Antes de provar esse teorema, nos perguntamos, será que esse limitante é bom?
- Relembre que os algoritmos mais triviais exigiam uma contante vezes n^2, enquanto o MergeSort é uma constante vezes $n \log_2 n$.
- Outra breve lembrança é do que é um $\log_2$, de maneira informal podemos dizer que $\log_2$ de um número, é a quantidade de vezes que você precisa dividir por 2 até chegar em 1.
- Então o $\log_2 4$ é 2, $\log_2 8 = 3$, $\log_2 16384 = 14$. Ou seja $\log_2 n$ é uma função que cresce devagar.

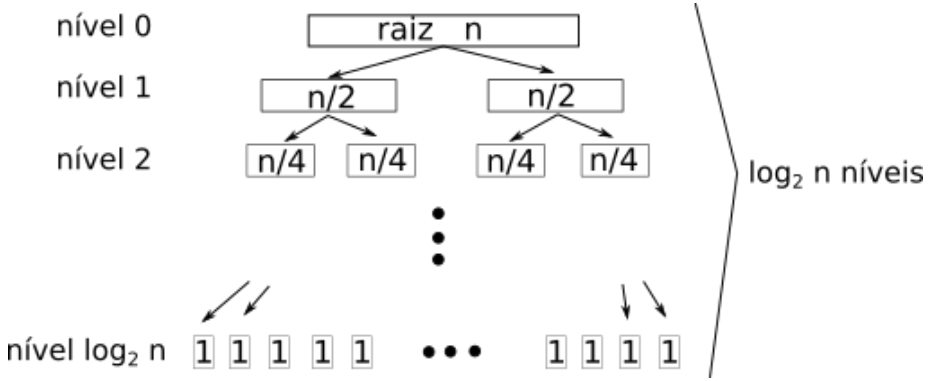




Complexidade do MergeSort

- Para demonstrar a complexidade do MergeSort iremos usar um recurso conhecido como árvore de recursão
- Ressalva: Alguns autores não consideram uma árvore de recursão como uma prova completa para uma afirmação.

Árvore de Recursão



Aproximadamente quantos níveis tem essa árvore de recursão? Sendo n o número de elementos no vetor.

a) Um número constante (independente de n)
b) $\log_2 n$
c) $\sqrt{n}$
d) n

Resposta:

- $\log_2 n + 1$ (nível 0)

Qual o padrão? Em cada nível $j = 0, 1, \ldots, \log_2 n$, existem ___ subproblemas, cada um com tamanho ___.

a) 2^j e 2^j
b) $n/2^j$ e $n/2^j$
c) 2^j e $n/2^j$
d) $n/2^j$ e 2^j

Teorema

MergeSort exige menos de $6n\log_2 n + 6n$ operações para ordenar n números.

Demonstração.

- Em cada nível $j = 0, 1, \ldots, \log_2 n$ existem 2^j subproblemas, cada um de tamanho $n/2^j$.
- Total de operações no nível j:

$$\leq 2^j \cdot 6\left(\frac{n}{2^j}\right)$$

$$= 6(n)$$

- Total de operações: número de níveis · operações por nível

$$(\log_2 n + 1)6n$$

$$6n\log_2 n + 6n$$

□

Princípios da Análise de Algoritmos

- O que fizemos no caso do MergeSort foi uma análise de Pior Caso, ou seja, qualquer que seja a entrada sabemos que o algoritmo executaram em tempo $\leq 6n\log_2 n + 6n$.
- Esse limite também é aplicado se um adversário tentasse atribuir números de forma a deixar o algoritmo lento.
- Essa análise é particularmente interessante por não precisar entender a aplicação do problema, padrões de entrada e ela é usualmente mais útil e mais fácil que as alternativas:
  - Análise de Caso Médio (Exige assumir alguma distribuição da entrada, ter conhecimento do domínio, mais difícil de ser feita)
  - Desempenho em *Benchmarks*
  - Análise de Melhor Caso (inútil na maior parte do tempo)

Princípios da Análise de Algoritmos

- Além disso, as constantes dependem muito da qualidade da implementação, da linguagem, do compilador utilizado e da arquitetura da máquina, coisas que não estamos nos preocupando aqui.
- A termos de menor ordem tem pouco impacto na predição do comportamento dos algoritmos.
- A ideia é então ignorar as constantes e os termos de menor ordem o que:
  - torna a análise geralmente mais simples e
  - mantêm a capacidade de predição.
- Ressalva, constantes e termos de menor onde podem ser importantes em aplicações críticas, ou partes muito executadas de um código.

Princípios da Análise de Algoritmos

- Você poderia imaginar que com o avanço dos hardwares, bastaria eu usar um computador melhor.
- Na verdade quanto maior o poder computacional, MAIOR é a disparidade entre algoritmos eficientes.
- Podemos pensar no tamanho do problema que podemos resolver com computadores mais potentes usando diferentes algoritmos.
- Suponha que você tem dois algoritmos para um problema.

Algoritmo A	Algoritmo B
n	n^2

- Suponha que você fez um grande investimento e comprou um computador 4 vezes mais potente. Com o algoritmo A você pode resolver um problema 4 vezes maior, enquanto com o algoritmo B você só pode resolver um problema 2 vezes maior.

Análise assintótica de funções quadráticas - termos de menor ordem

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

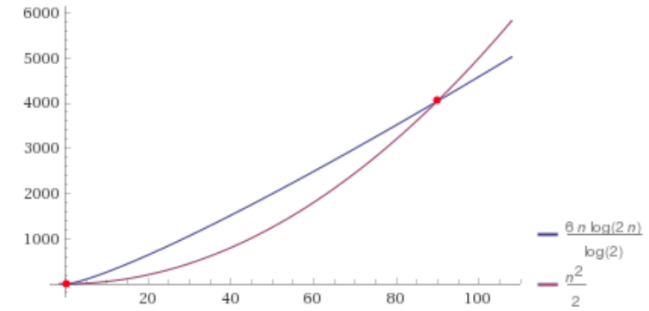
- Como se vê, $3n^2$ é o termo dominante quando n é grande.
- De um modo geral, podemos nos concentrar nos termos dominantes e esquecer os demais.

Princípios da Análise de Algoritmos

- Iremos fazer uma Análise **Assintótica** dos algoritmos, o que significa que estamos interessados no comportamento deles para instâncias **grandes**.
- Isso nos permite dizer que um algoritmo é mais rápido que outro assumindo que o tamanho n da instância é suficientemente grande.
- Por exemplo, podemos dizer com segurança que um algoritmo que executa em $6n \log_2 n + 6n$ é mais rápido que um que executa em $\frac{1}{2}n^2$
- Note que isso pode não ser verdade para n pequeno, mas a partir de algum $n = n_0$ sempre será verdade.
- De fato, para n pequeno, tanto faz o algoritmo que você use. Estamos interessados em resolver problemas grandes!

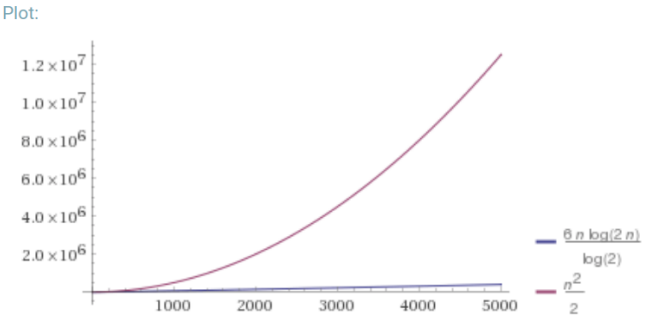
Princípios da Análise de Algoritmos

Plot:



Princípios da Análise de Algoritmos

Plot:



Análise Assintótica

- É a linguagem que os cientistas da computação (sérios) usam para discutir o desempenho em alto nível de algoritmos.
- É fundamental para o vocabulário do cientista da computação. Quando alguém diz "O MergeSort executa em $O(n \log n)$, e o InsertionSort executa em $O(n^2)$"o que exatamente ele está dizendo?
- A análise assintótica é uma ferramenta adequada pois:
  - É simples o bastante para suprimir detalhes de arquitetura/linguagem/compilador.
  - Mas é complexa o bastante para permitir a comparação entre diferentes algoritmos, especialmente em instâncias grandes.

Análise Assintótica

Ideia geral

Suprimir fatores constantes e termos de ordens inferiores.

- Por exemplo em:

$$6n \log_2 n + 6n$$

  6n é um termo de ordem inferior, 6 é constante então resultaria em:

$$n \log n$$

  Exercício: A base do log também não importa. Por que?
- Então quando dizemos que o tempo de execução do MergeSort é $O(n \log n)$, ou de maneira geral quando dizemos que um algoritmo é $O(f(n))$. Estamos dizendo que depois de eliminar os termos de ordem inferior e constantes acabamos apenas com $f(n)$.

Exemplos

Exemplo de um laço

Algoritmo 3: Procura

Entrada: Um vetor A de tamanho n e um inteiro t
Saída: Verdadeiro se A contêm t, Falso caso contrário

```
1 para k de 1 até n faça
2     se A[k] == t então
3         devolva Verdadeiro;
4 devolva Falso;
```

Qual o tempo de execução desse algoritmo?

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

- O tempo de execução do exemplo depende por exemplo se t está ou não em A, e se t estiver na primeira posição? Estamos interessados no pior caso!
- Quantas operações estamos fazendo nas linhas 1 e 2? Uma, duas, três? Isso é constante e é suprimido pela notação O.

Exemplos

Exemplo de dois laços consecutivos

Algoritmo 4: Procura2

Entrada: Dois vetores A e B de tamanho n e um inteiro t
Saída: Verdadeiro se A ou B contêm t, Falso caso contrário

```
1 para k de 1 até n faça
2     se A[k] == t então devolva Verdadeiro;
3 para k de 1 até n faça
4     se B[k] == t então devolva Verdadeiro;
5 devolva Falso;
```

Qual o tempo de execução desse algoritmo?

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

- Evidentemente, nesse problema a entrada é na verdade de tamanho $2n$, então o algoritmo não seria $O(2n)$? Essa constante é suprimida na notação O.

Exemplos

Exemplo de dois laços aninhados

Algoritmo 5: ProcuraComum

Entrada: Dois vetores A e B de tamanho n
Saída: Verdadeiro se A ou B têm um numero em comum, Falso caso contrário

```
1 para j de 1 até n faça
2     para k de 1 até n faça
3         se A[j] == B[k] então devolva Verdadeiro;
4 devolva Falso;
```

Qual o tempo de execução desse algoritmo?

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

- Pior caso ✓. Constantes ✓.
- Nesse caso se o tamanho dos vetores dobrar, o tempo de execução quadruplica!

Exemplos

Exemplo de dois laços aninhados

Algoritmo 6: ProcuraComum

Entrada: Um vetor A de tamanho n
Saída: Verdadeiro se A tem números duplicados, Falso caso contrário

```
1 para j de 1 até n faça
2     para k de j + 1 até n faça
3         se A[j] == A[k] então devolva Verdadeiro;
4 devolva Falso;
```

- Dessa vez, obviamente, procuramos no vetor A ao invés de B.
- Ao invés de olhar todas combinações de j e k, olhamos apenas aquelas em que $k \geq j + 1$.
- Isso é válido pois comparar, por exemplo, o primeiro com o terceiro elemento é a mesma coisa que comparar o terceiro com o primeiro. Então diminuímos nossas comparações pela metade!

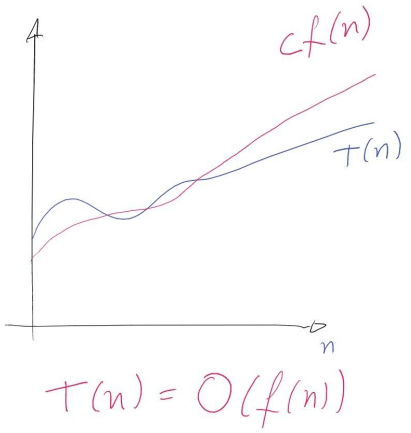
Qual o tempo de execução desse algoritmo?

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

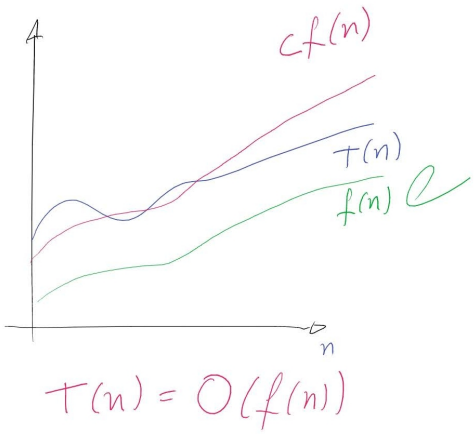
Exercício: É possível fazer esse algoritmo mais eficiente? Como?

Notação O

- Big Oh, Ózão, O grande.
- Notação O é utilizada em funções definidas nos inteiros positivos.
- Seja $T(n)$ uma função sobre $n = 1, 2, 3, \ldots$
- $T(n) : \mathbb{Z}_+^* \to \mathbb{R}$
- Pergunta: Quando podemos dizer que $T(n) = O(f(n))$?
- Resposta: Se eventualmente, para um n suficientemente grande, $T(n)$ é limitado superiormente por uma alguma constante vezes $f(n)$.



- $T(n) = O(f(n))$ se quanto multiplicado por alguma constante c tal que $c \cdot f(n)$ está sempre acima de $T(n)$



- É válido mesmo se $f(n)$ ou $c' \cdot f(n)$ esteja sempre abaixo de $T(n)$, o importante é existir uma constante que a deixe sempre acima.

Formalizando o O

Definição

$T(n) = O(f(n))$ se e somente se exitem constantes c, $n_0 > 0$ tais que

$$T(n) \leq c.f(n)$$

para todo $n \geq n_0$.

- Então para provar que $T(n) = O(f(n))$ você precisa exibir c e n_o que provem que $T(n) \leq c.f(n)$ para todo $n \geq n_0$.
- Dizer que c e n_0 são constantes, significa que não dependem de n. (Tudo bem se c depender de n_0 e vice versa).
- (muito) Formalmente $O(f(n))$ é um conjunto de funções que podem ser limitadas por $f(n)$, então o correto seria dizer que $T(n) \in O(f(n))$. Mas dizer que $T(n) = O(f(n))$ é um abuso de notação comumente aceito e que facilita a manipulação dessas funções.

Exemplo

Teorema

Se $T(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ *então* $T(n) = O(n^k)$

- Para provar precisamos exigir constantes c e $n_0 > 0$ tais que $T(n) \leq c.f(n)$ para todo $n \geq n_0$.
- Vamos então escolher $n_0 = 1$ e $c = |a_k| + |a_{k-1}| + \ldots + |a_1| + |a_0|$.

$$\begin{aligned} T(n) &= a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \\ &\leq |a_k| n^k + |a_{k-1}| n^{k-1} + \ldots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + |a_{k-1}| n^k + \ldots + |a_1| n^k + |a_0| n^k \text{ (válido para } n > n_0) \\ &= (|a_k| + |a_{k-1}| + \ldots + |a_1| + |a_0|) n^k \\ &= c n^k \end{aligned}$$

portanto, $T(n) \leq cn^k$ logo, $T(n) = O(n^k)$ □

Teorema

Para todo $k \geq 1$, n^k *não é* $O(n^{k-1})$.

- Podemos provar por contradição, ou seja, assumimos que a premissa é verdadeira porém a conclusão é falsa e chegamos a algum absurdo.
- Supomos que existe um $k \geq 1$ e constantes c, $n_0 > 0$ tal que $n^k = O(n^{k-1})$, ou seja,

$$\begin{aligned} n^k &\leq c n^{k-1} && \forall n \geq n_0 \\ n n^{k-1} &\leq c n^{k-1} && \forall n \geq n_0 \\ \cancel{n n^{k-1}} &\leq \cancel{c n^{k-1}} && \forall n \geq n_0 \\ n &\leq c && \forall n \geq n_0 \end{aligned}$$

Como n pode ser todos os naturais maiores do que n_0, não pode existir tal constante c que seja maior do que qualquer natural. Portanto chegamos a um absurdo e provamos que todo $k \geq 1$, n^k não é $O(n^{k-1})$. □

Notação Ω e Θ

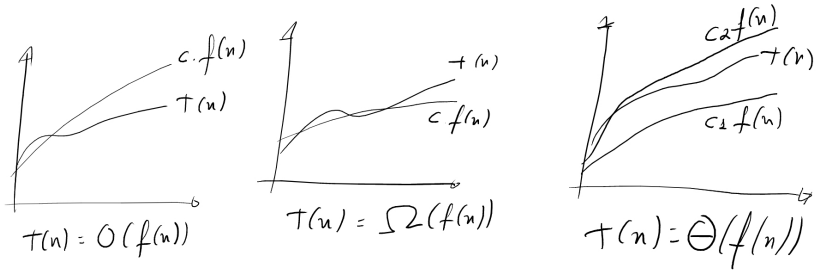
Analogia

- Intuitivamente, você pode comparar a notação O com uma relação de $\leq$. Já que dizer que $T(n)$ ser $O(f(n))$, quer dizer que: (existe constante c e n_0 tal que para todo n maior que n_0)

$$T(n) \leq cf(n)$$

- Nessa comparação a notação Ω seria como uma relação de $\geq$
- E a notação Θ seria como uma relação de $=$

Notação Ω e Θ



Notação Ω

A notação Ω que define um limitante inferior. Formalmente:

Definição

$T(n) = \Omega(f(n))$ se e somente se exitem constantes c, $n_0 > 0$ tais que

$$T(n) \geq c.f(n)$$

para todo $n \geq n_0$.

Notação Θ

A notação Θ indica que uma função $T(n)$ é limitada inferiormente e superiormente por uma função $f(n)$. Formalmente:

Definição

$T(n) = \Theta(f(n))$ se e somente se exitem constantes c_1, c_2 e $n_0 > 0$ tais que

$$c_1.f(n) \leq T(n) \leq c_2.f(n)$$

para todo $n \geq n_0$.

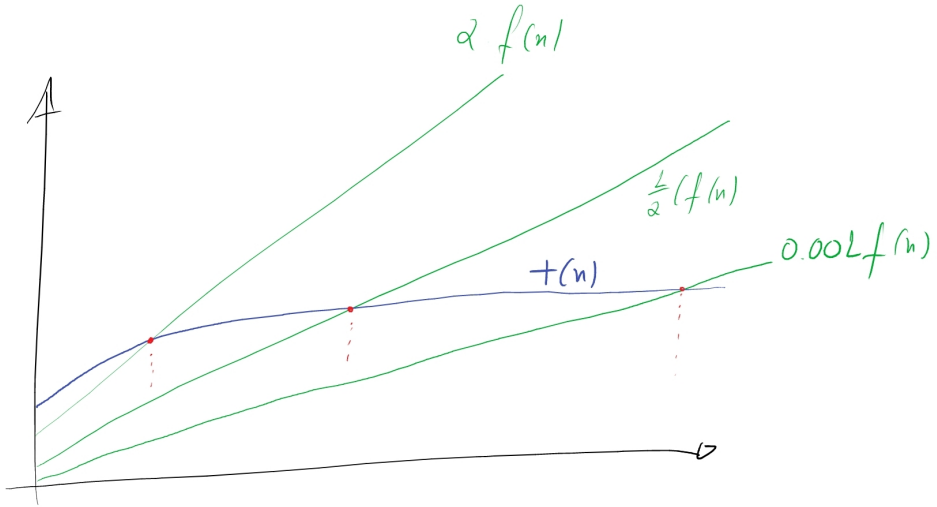
Definição

$T(n) = \Theta(f(n))$ se e somente se $T(n) = O(f(n))$ e também se $T(n) = \Omega(f(n))$.

o e ω

- A notação o (Little-Oh, ózinho, ó pequeno) é semelhante a notação O porém não assintoticamente justo.
- Informalmente pode-se pensar que se O está para uma relação do tipo $\leq$, enquanto o o é uma relação do tipo $<$.
- A notação ω (Little-omega, omeguinha, omega pequeno) é semelhante a notação Ω porém não assintoticamente justo.
- Informalmente pode-se pensar que Ω está para uma relação do tipo $\geq$, enquanto a ω é uma relação do tipo $>$.

o e ω



Notação o

Definição

$T(n) = o(f(n))$ se e somente se para toda constante $c > 0$, existe um $n_0 > 0$ tal que

$$T(n) < c.f(n)$$

para todo $n \geq n_0$.

Exercício: Provar que para todo $k \geq 1$, $n^{k-1} = o(n^k)$.

Notação ω

Definição

$T(n) = \omega(f(n))$ se e somente se para toda constante $c > 0$, existe um $n_0 > 0$ tal que

$$T(n) > c.f(n)$$

para todo $n \geq n_0$.

Mais exemplos de Notação Assintótica

Exemplo

Provar que $2^{n+10} = O(2^n)$.

Precisamos exibir c e $n_0 > 0$ tais que

$$2^{n+10} \leq c2^n \text{ para todo } n \geq n_0.$$

Como escolher c e n_0?

$$2^{n+10} \leq c2^n$$
$$2^{10}2^n \leq c2^n$$
$$1024 \cdot 2^n \leq c2^n$$

Quando isso é verdade?
Escolhemos então algum $c \geq 1024$, e $n_0 \geq 1$.

Exemplo

Provar que $2^{n+10} = O(2^n)$.

Precisamos exibir c e $n_0 > 0$ tais que

$$2^{n+10} \leq c2^n \text{ para todo } n \geq n_0.$$

Escolhemos então $c = 1024$, e $n_0 \geq 1$ e vamos mostrar que:

$$2^{n+10} \leq 1024 \cdot 2^n \text{ para todo } n \geq 1$$
$$2^{10}2^n \leq 1024 \cdot 2^n \text{ para todo } n \geq 1$$
$$1024 \cdot 2^n \leq 1024 \cdot 2^n \text{ para todo } n \geq 1$$

logo $2^{n+10} = O(2^n)$. □

Exemplo

Provar que 2^{10n} não é $O(2^n)$.

Suponha por absurdo (por contradição) que $2^{10n} = O(2^n)$ e portanto $2^{10n} \leq c2^n$ para alguma constante c e $n \geq n_0$. Então

$$2^{10n} \leq c2^n$$
$$2^{9n}2^n \leq c2^n$$
$$2^{9n}\cancel{2^n} \leq c\cancel{2^n}$$
$$2^{9n} \leq c$$

obviamente não existe uma constante que seja maior que 2^{9n} para todos os naturais n, portanto chegamos a uma contradição. Logo 2^{10n} não pode ser $O(2^n)$. □

Exemplo

Para quaisquer dois pares de funções positivas, $f(n)$ e $g(n)$, provar que $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

Para mostrar que a afirmação é verdadeira, podemos escolher $c_1 = 1/2$, $c_2 = 1$ e $n_0 = 1$ e verificamos que:

$$\frac{1}{2}(f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq f(n) + g(n) \qquad \forall n \geq n_0$$

Usando limites

$f(n) \in o(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$f(n) \in \omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$f(n) \in O(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

$f(n) \in \Omega(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$.

$f(n) \in \Theta(g(n))$ se $0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

Exemplo

$f(n) = \ln n$ e $g(n) = n^e$.

$$\lim_{n\to\infty} \frac{\ln n}{n^e} = \lim_{n\to\infty} \frac{1/n}{e.n^{e-1}} = 0.$$

portanto $\ln n = o(n^e)$. Obviamente $\ln n = O(n^e)$ também.

Divisão e Conquista: O paradigma

O paradigma de divisão e conquista tem três etapas:

1. **Dividir** o problema em subproblemas menores.
2. **Conquistar** os subproblemas (normalmente de forma recursiva).
3. **Combinar** a solução dos subproblema para encontrar uma solução para o problema original.

Diferentes algoritmos tem a complexidade diferente em cada uma das fases, por exemplo, no MergeSort a divisão é trivial mas a combinação exige esforço. Já no QuickSort a divisão é bastante elaborada mas a combinação é trivial.

O Problema

- Suponha que você e um amigo escolheram 10 filmes que ambos assistiram.
- Cada um ordenou esses 10 filmes em ordem de preferência.
- Queremos saber a compatibilidade entre essas duas listas, e verificar se essa amizade pode dar certo.
- Um serviço de streaming poderia usar essa comparação para verificar usuários que tem gostos parecidos para fazer recomendações.

Problema do Número de Inversões

Problema do Número de Inversões

Dado um arranjo A contendo n inteiros em uma ordem arbitrária, encontrar o número total de inversões, ou seja, o número de pares (i, j) de índices $1 < i, j < n$ tais que $i < j$ e $A[i] > A[j]$.

Considere o vetor seguinte vetor

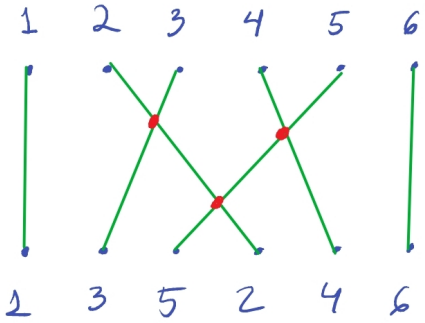
$$A = (1, 3, 5, 2, 4, 6)$$

qual o número de inversões?

- os elementos 3 e 2, então os índices $(2, 4)$ formam uma inversão
- os elementos 5 e 2, então os índices $(3, 4)$ formam uma inversão
- os elementos 5 e 4, então os índices $(3, 5)$ formam uma inversão

Problema do Número de Inversões

$$A = (1, 3, 5, 2, 4, 6)$$



Problema do Número de Inversões

Qual o número máximo de inversões em um vetor de tamanho 6?

a. 6
b. 15
c. 21
d. 36
e. 64

Problema do Número de Inversões

Qual o número máximo de inversões em um vetor de tamanho n?

- O máximo de inversões acontece se todos os pares de índice estiverem invertidos.
- Ou seja, dos n elementos, quaisquer dois que escolhermos estará invertido.

$$\binom{n}{2}$$

- Relembrando:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- Então

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n.(n-1).(n-2)!}{2!(n-2)!} = \frac{n.(n-1).\cancel{(n-2)!}}{2!\cancel{(n-2)!}} = \frac{n^2-n}{2}$$

Uma ideia ingenua

Como seria uma solução força bruta?

Algoritmo 7: ContaInversões

Entrada: Um vetor A de tamanho n
Saída: O número de inversões

```
1 t = 0;
2 para i de 1 até n − 1 faça
3     para j de i + 1 até n faça
4         se A[i] > A[j] então
5             t++ ;
6 devolva t;
```

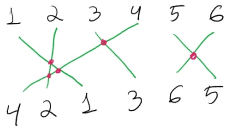
Qual a complexidade desse algoritmo?

a. $\log n$
b. n
c. $n \log n$
d. n^2
e. n^3

Podemos fazer melhor? Yep!

Uma algoritmo de divisão e conquista

$(4, 2, 1, 3, 6, 5)$



Valor verdadeiro: 5 inversões

Divisão
(4,2,1) (3,6,5)
Conquista
3 inversões 1 inversão
Combinação?
Como contar as inversões entre as metades?

Uma algoritmo de divisão e conquista

- Ideia: dividir o vetor em 2 metades, contar o número de inversões.
- Mas ainda faltará as inversões entre os elementos da primeira e da segunda metade.
- Podemos então contar essas inversões?
- Para isso vamos então classificar as inversões em três tipos:
  - **Esquerda:** se $i, j \leq n/2$
  - **Direita:** se $i, j > n/2$
  - **Split:** se $i \leq n/2 < j$
- Nova ideia: contar as inversões esquerda, direita e split.

Uma algoritmo de divisão e conquista

Uma ideia (ainda incompleta) seria:

Algoritmo 8: Count

Entrada: Um arranjo A de comprimento n
Saída: O número de inversões

```
1 se n ≤ 1 então devolva 0;
2 senão
3     x = Count(Primeira metade de A, n/2);
4     y = Count(Segunda metade de A, n/2);
5     z = ContaSplit(A,n);
6 devolva x + y + z;
```

- Se conseguirmos contar o número de inversões split em tempo linear $O(n)$ a árvore de recursão fica idêntica ao MergeSort.
- A complexidade total ficaria $O(n \log n)$
- O número de inversões de tipo split é $O(n^2)$. Será que podemos contar um número quadrático de coisas em tempo linear? 🤔
- Yep!

- Considere que Count conta o número de inversões, mas também ordena o vetor.
- E vamos dar uma olhada na função Merge do MergeSort

Algoritmo 9: Merge

Entrada: B e C arranjos ordenados com $m/2$
Saída: Arranjo D de tamanho m com os mesmos elementos de B e C mas ordenados

```
1 i = 1; j = 1;
2 para k de 1 até m faça
3     se B[i] < C[j] então
4         D[k] = B[i];
5         i++;
6     senão
7         D[k] = C[j];
8         j++;
9 devolva C;
```

- Se não houver inversões do tipo split, o vai acontecer na hora de copiar B e C?
- Nesse caso B seria copiado inteiramente antes de C.
- O que acontece significa, em número de inversões, quando copiamos um elemento do vetor C?
- Isso significa que o elemento $C[j]$ copiado está em inversão do tipo split com todos os valores que ainda não foram copiados de B.
- Isso significa $|B| - i + 1$ elementos. (Se indexar em 0 não precisa do "+1")

Modificando o Merge para Contar Inversões Splits

Algoritmo 10: Merge

Entrada: B e C arranjos ordenados com $m/2$
Saída: Arranjo D de tamanho m com os mesmos elementos de B e C mas ordenados

```
1 i = 1; j = 1;
2 para k de 1 até m faça
3     se B[i] < C[j] então
4         D[k] = B[i];
5         i++;
6     senão
7         D[k] = C[j];
8         j++;
9 devolva C;
```

Algoritmo 11: MergeCountSplit

Entrada: B e C arranjos ordenados com $m/2$
Saída: Arranjo D de tamanho m com os elementos de B e C mas ordenados, e o número de inversões Splits

```
1 i = 1; j = 1; t = 0;
2 para k de 1 até m faça
3     se B[i] < C[j] então
4         D[k] = B[i];
5         i++;
6     senão
7         D[k] = C[j];
8         j++; t = t + (m/2 − i + 1);
9 devolva (C, t);
```


Uma algoritmo de divisão e conquista (versão completa)

Algoritmo 12: SortCount

Entrada: Um arranjo A, o comprimento do arranjo n
Saída: Um arranjo com os mesmos elementos de A porém ordenados, O número de inversões

```
1 se n ≤ 1 então devolva (A, 0);
2 senão
3     (B, x) = SortCount(Primeira metade de A, n/2);
4     (C, y) = SortCount(Segunda metade de A, n/2);
5     (D, z) = MergeCountSplit(B, C, n);
6 devolva (D, x + y + z);
```

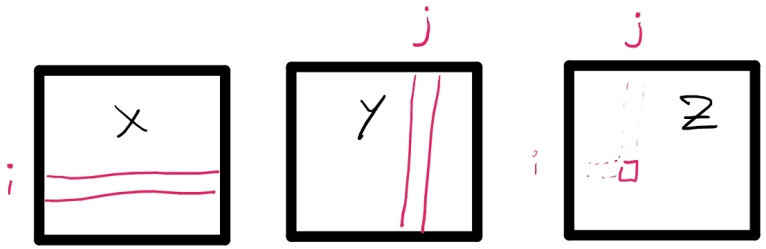

- Assim como no MergeSort, a árvore de recursão de SortCount tem $\log n$ níveis e cada nível executa $O(n)$ operações, então a complexidade total de SortCount é

$$O(n \log n)$$

- Portanto muito melhor que a versão força bruta!

O Problema da Multiplicação de matrizes é extremamente importante pois é a essencial para aplicações em diversas áreas:

- Computação Gráfica
- Machine Learning
- Biologia Computacional
- Algoritmos Matemáticos e Físicos.
- e muitas outras.



Problema de Multiplicação de Matrizes

Sejam X e Y duas matrizes quadrada $n \times n$, desejamos obter a matriz $Z = X.Y$ também $n \times n$, tal que

$$Z_{ij} = \sum_{k=1}^{n} X_{ik}.Y_{kj}$$

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Para obter cada número Z_{ij} precisamos multiplicar os n pares de elementos formados por X_{ik} e Y_{kj} para $k = 1 \ldots n$. Qual a complexidade para se obter cada Z_{ij}?

a. $\Theta(n)$
b. $\Theta(n \log n)$
c. $\Theta(n^2)$
d. $\Theta(n^3)$

E qual o tempo de execução total do algoritmo para obter todos os $O(n^2)$ valores de Z em função da largura das matrizes n?

a. $\Theta(n \log n)$
b. $\Theta(n^2)$
c. $\Theta(n^3)$
d. $\Theta(n^4)$

- Será possível fazer em tempo menor que cubico?
- Vamos tentar o paradigma de Divisão e Conquista!
- Identificar os passos de Divisão, Conquista e Combinação.
- Será que conseguimos dividir a matriz de alguma forma como no algoritmo de contagem de inversões.

Primeira Ideia

Dividir cada matriz em quadrantes:

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \text{ e } Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

A multiplicação de matrizes nesse caso se comporta como se multiplicássemos elementos isolados.

$$Z = X.Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

$$XY = \begin{pmatrix} a_{11} & a_{12} & b_{11} & b_{12} \\ a_{21} & a_{22} & b_{21} & b_{22} \\ c_{11} & c_{12} & d_{11} & d_{12} \\ c_{21} & c_{22} & d_{21} & d_{22} \end{pmatrix} \begin{pmatrix} e_{11} & e_{12} & f_{11} & f_{12} \\ e_{21} & e_{22} & f_{21} & f_{22} \\ g_{11} & g_{12} & h_{11} & h_{12} \\ g_{21} & g_{22} & h_{21} & h_{22} \end{pmatrix}$$

$$= \begin{pmatrix} a_{11}e_{11} + a_{12}e_{21} + b_{11}g_{11} + b_{12}g_{21} & a_{11}e_{12} + a_{12}e_{22} + b_{11}g_{12} + b_{12}g_{22} & z_{13} & z_{14} \\ a_{21}e_{11} + a_{22}e_{21} + b_{21}g_{11} + b_{22}g_{21} & a_{21}e_{12} + a_{22}e_{22} + b_{21}g_{12} + b_{22}g_{22} & z_{23} & z_{24} \\ z_{31} & z_{32} & z_{33} & z_{34} \\ z_{41} & z_{42} & z_{43} & z_{44} \end{pmatrix}$$

De fato o primeiro quadrante do resultado é:

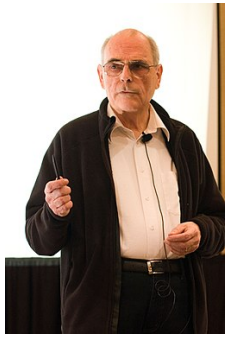
$$AE + BG$$

$$Z = XY = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

- Podemos criar um algoritmo recursivo que calcula os 8 subproblemas de maneira recursiva, cada um com matrizes de dimensão $n/2$ e depois combina com as somas que podem ser feitas em tempo $O(n^2)$.
- A notícia ruim é que esse algoritmo também é $O(n^3)$ igual ao direto. Pfft! 😒
- Talvez aplicar algum *truque* como o de Gauss no algoritmo de Karatsuba?

Volker Strassen

- Volker Strassen, é um matemático alemão nascido em 1936.
- Professor emérito do Departamento de Matemática e Estatística da University of Konstanz.
- Recebeu diversos prêmios por suas contribuições em projeto e análise de algoritmos
- Em 1969 apresentou o primeiro algoritmo para fazer multiplicações de matrizes em tempo de execução inferior a $O(n^3)$.
- Causou um grande *frisson* na época, pois não acreditavam que tal multiplicação pudesse ter tempo subcúbico.



Algoritmo de Strassen - 1969

- Ideia: Reduzir o número de chamadas recursivas!
- Iremos computar apenas 7 chamadas recursivas.
- Faremos as adições e subtrações necessárias. O que ainda vai requisitar $O(n^2)$.

Os 7 produtos a serem computados são:

- $P_1 = A(F - H)$,
- $P_2 = (A + B)H$,
- $P_3 = (C + D)E$,
- $P_4 = D(G - E)$,
- $P_5 = (A + D)(E + H)$,
- $P_6 = (B - D)(G + H)$ e
- $P_7 = (A - C)(E + F)$.

$$X.Y = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix} = \begin{pmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{pmatrix}$$

$$\begin{aligned} & P_5 + P_4 - P_2 + P_6 \\ &= (A + D)(E + H) + D(G - E) - (A + B)H + (B - D)(G + H) \\ &= AE + AH + DE + DH + DG - DE - AH - BH + BG + BH - DG - DH \\ &= AE + \cancel{AH} + \cancel{DE} + \cancel{DH} + \cancel{DG} - \cancel{DE} - \cancel{AH} - \cancel{BH} + BG + \cancel{BH} - \cancel{DG} - \cancel{DH} \\ &= AE + BG \end{aligned}$$

- Pelo teorema mestre a complexidade desse algoritmo é $O(n^{\log_2 7})$ (quando está no expoente a base do logaritmo importa sim) que é $\approx O(n^{2.8})$.
- Portando subcubico!
- As constantes no algoritmo de Strassen são bem maiores que as da multiplicação tradicional. Por isso só valem a pena para matrizes a partir de um certo tamanho. Para matrizes pequenas vale mais a pena a multiplicação tradicional.
- Minha sugestão para implementar o Algoritmo de Strassen é que na recursão, a partir de matrizes menores de 32 você use a multiplicação tradicional.
- Existem algoritmos teóricos mais eficientes como o de Coppersmith–Winogra, que é $O(n^{2.375})$ mas não é usado na prática porque só seria melhor que o Strassen para matrizes tão grandes que não são possíveis de serem processadas.

Teorema Mestre

- O teorema mestre é uma ferramenta útil para avaliar algoritmos de divisão e conquista, que normalmente precisam de uma análise matemática mais complexa.
- Por exemplo os algoritmos de Karatsuba, de Contagem de Inversões e o Algoritmo de Strassen.

Problema da Multiplicação de Inteiros

Multiplicação de Inteiros

Dado dois inteiros x e y de n dígitos cada. Encontrar o produto $x \cdot y$.

Dividir $x = 10^{n/2}a + b$ e $y = 10^{n/2}c + d$, dessa forma:

$$xy = 10^n ac + 10^{n/2}(ad + bc) + bd$$

- Estratégia 1:
  - calcular recursivamente ac, ad, bc e bd
  - seja $T(n)$ o tempo de execução máximo para resolver um problema de tamanho n

$$\begin{aligned} \text{Para } n > 1: &\quad T(n) \leq 4T\left(\frac{n}{2}\right) + O(n) \\ \text{Caso base}: &\quad T(1) \leq O(1) \end{aligned}$$

- Estratégia 2 (Algoritmo de Karatsuba)
  - $xy = 10^n ac + 10^{n/2}(ad + bc) + bd$
  - calcular recursivamente ac, bd e $(a + b)(c + d)$
  - obter $ad + bc = (a + b)(c + d) - ac - bd$

$$\begin{aligned} \text{Para } n > 1: &\quad T(n) \leq 3T\left(\frac{n}{2}\right) + O(n) \\ \text{Caso base}: &\quad T(1) \leq O(1) \end{aligned}$$

O Teorema Mestre

- Pode ser usado como uma caixa preta para resolver recorrências.
- Só funciona quando todos os subproblemas tem o mesmo tamanho.

Suponha uma recorrência da seguinte forma:

$$\begin{aligned} \text{Caso base}: &\quad T(n) \leq O(1) \text{ para } n \text{ suficientemente pequeno} \\ &\quad T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d) \end{aligned}$$

$$T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d \log n) & \text{se } a = b^d \\ O(n^d) & \text{se } a < b^d \\ O(n^{\log_b a}) & \text{se } a > b^d \end{cases}$$

Multiplicação de Inteiros com 4 chamadas recursivas:

- $T(n) \leq 4T\left(\frac{n}{2}\right) + O(n)$
- Como $4 > 2$, caímos no caso 3. Portanto:
- $T(n) = O(n^{\log_2 4}) = O(n^2)$

$$T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d \log n) & \text{se } a = b^d \\ O(n^d) & \text{se } a < b^d \\ O(n^{\log_b a}) & \text{se } a > b^d \end{cases}$$

Algoritmo de Karatsuba

- $T(n) \leq 3T\left(\frac{n}{2}\right) + O(n)$
- Como $3 > 2$, também caímos no caso 3. Portanto:
- $T(n) = O(n^{\log_2 3}) \approx O(n^{1.58496})$

MergeSort:

- $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$
- Como $2 = 2$, caímos no caso 1. Portanto:
- $T(n) = O(n^1 \log n) = O(n \log n)$

$$T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = \begin{cases} O(n^d \log n) & \text{se } a = b^d \\ O(n^d) & \text{se } a < b^d \\ O(n^{\log_b a}) & \text{se } a > b^d \end{cases}$$

Algoritmo de Strassen

- $T(n) \leq 7T\left(\frac{n}{2}\right) + O(n^2)$
- Como $7 > 4$, também caímos no caso 3. Portanto:
- $T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$

Prova do Teorema Mestre

- Vamos considerar que n é uma potência de b.
- Ressalva: Essa prova não está 100% rigorosa, mas funciona para entender porque e como o teorema mestre funciona
- Usaremos a árvore de Recursão

$$T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$$

- Considere um nível j da árvore de recursão. Quanto trabalho é executado nesse nível?
- Número de subproblemas: a^j
- Tamanho de cada subproblema: $\frac{n}{b^j}$

Trabalho no nível $j \leq a^j O\left(\left(\frac{n}{b^j}\right)^d\right) \leq a^j \cdot c \cdot \frac{n^d}{b^{jd}} = c \cdot n^d \cdot \frac{a^j}{b^{jd}} = c \cdot n^d \cdot \left(\frac{a}{b^d}\right)^j$

Somando todos os níveis:

$$T(n) \leq c \cdot n^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

$$T(n) \leq c \cdot n^d \cdot \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

- a é a taxa de proliferação de subproblema
- b^d é a taxa de encolhimento do trabalho no subproblema
- Intuitivamente:
  - se $a = b^d$ o trabalho é igualmente distribuído por toda a árvore e portanto $O(n^d \log n)$
  - se $a < b^d$ o trabalho mais bruto está na raiz, portanto $O(n^d)$
  - se $a > b^d$ temos muitas folhas e portanto o trabalho principal está lá, Portanto O(número de folhas)

$$T(n) \leq c \cdot n^d \cdot \underbrace{\sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j}_{S}$$

- $r = \frac{a}{b^d}$
- se $r = 1$, $S = \sum_{j=0}^{\log_b n} 1 = \log_b n$ logo

$$T(n) \leq c \cdot n^d \cdot \log_b n \text{ e ai é fácil mostra que } T(n) = O(n \log n)$$

- se $r < 1$, $S = \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \leq 1 + r + r^2 + \ldots$ isso é uma Progressão geométrica de razão $r < 1$. Portanto:

$$T(n) \leq c \cdot n^d \cdot \frac{1}{1-r} \text{ e ai é fácil mostra que } T(n) = O(n^d)$$

$$T(n) \leq c \cdot n^d \cdot \underbrace{\sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j}_{S}$$

- Se r > 1:

$$S = \sum_{j=0}^{\log_b n} r^j = 1 + r + r^2 + \ldots + r^{\log_b n}$$

Isso é uma PG de razão r.

$$S_x = \frac{a_1(1-r^x)}{1-r} \qquad S = \frac{1 - r^{\log_b n}}{1-r} = \frac{r^{\log_b n} - 1}{r-1} \leq c' r^{\log_b n}$$

logo

$$T(n) \leq c \cdot n^d \cdot c' \cdot r^{\log_b n} = c \cdot n^d \cdot c' \cdot \frac{a^{\log_b n}}{b^{\log_b n \cdot d}} = c \cdot n^d \cdot c' \cdot \frac{a^{\log_b n}}{n^d} = c \cdot \cancel{n^d} \cdot c' \cdot \frac{a^{\log_b n}}{\cancel{n^d}}$$

$$T(n) \leq cc' a^{\log_b n} = cc' n^{\log_b a} \text{ e ai é fácil mostra que } T(n) = O(n^{\log_b a}) \quad \square$$