Algoritmos

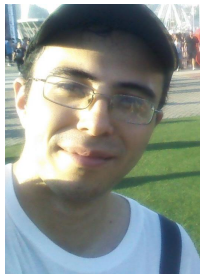
Pedro Hokama

Fontes

- [*clrs*] Algoritmos: Teoria e Prática (Terceira Edição) Thomas H. Cormen, Charles Eric Leiserson, Ronald Rivest e Clifford Stein.
- [*timr*] Algorithms Illuminated Series, Tim Roughgarden
- Desmistificando Algoritmos, Thomas H. Cormen.
- Algoritmos, Sanjoy Dasgupta, Christos Papadimitriou e Umesh Vazirani
- 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
- Conjunto de Slides do Professores Cid C. de Souza para a disciplina MO420

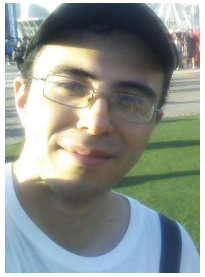
Qualquer erro é de minha responsabilidade.

Sobre o docente



- 2002 - 2004: Técnico em Programação e Desenvolvimento de Sistemas (CEFET-SP)
- 2006 - 2009: Bacharelado em Ciência da Computação (UNICAMP)
  - 2007-2008: IC - Algoritmos e Heurísticas para Empacotamento Tridimensional
  - 2008-2009: IC - Algoritmos e Heurísticas para o Problema de Roteamento Tridimensional
- 2009 - 2011: Mestrado em Ciência da Computação (UNICAMP) - O Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional
- 2011 - 2016: Doutorado em Ciência da Computação (UNICAMP) - Algoritmos para Problemas com Restrições de Empacotamento

Sobre o docente



- 2016 - 2018: Pós-doutorado (UFSCar)
- 2018 - atual: Professor Adjunto no Instituto de Matemática e Computação da Universidade Federal de Itajubá
  - Programa de Pós-Graduação em Ciência e Tecnologia da Computação.
  - Orientador de IC, TFG e Pós-Graduação: Algoritmos, Otimização, Teoria dos Jogos, Aprendizado de Máquina, etc..
  - Coordenador do Projeto de Extensão DevU - Desenvolvimento de Jogos
  - Coordenador de Mobilidade Acadêmica dos Cursos de Sistemas de Informação e Ciência da Computação.

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

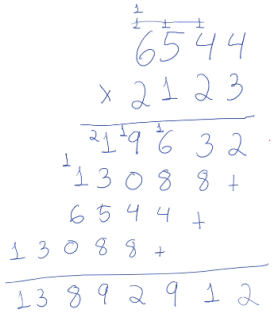
Vamos definir o problema da multiplicação de inteiros:

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 (!)

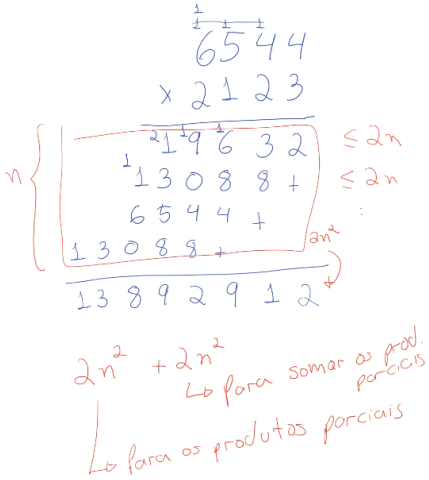


Resposta: 13892912

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 $a.c.10^n + e.10^{n/2} + b.d$

$$6544 \cdot 2123$$

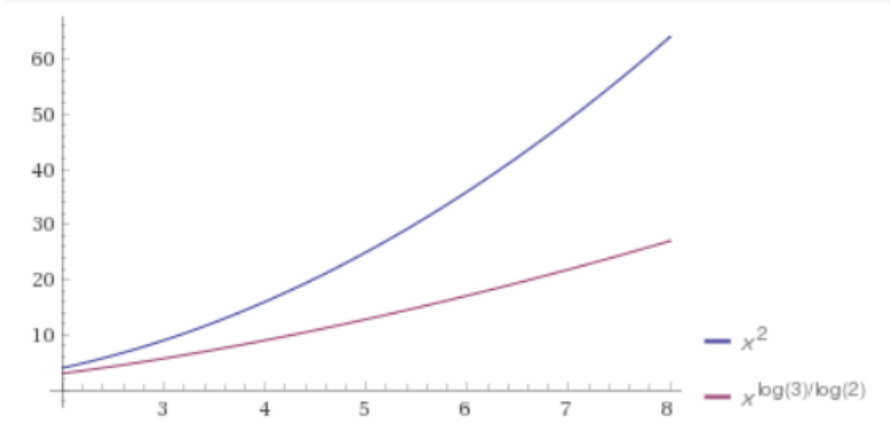
$$a = 65, b = 44, c = 21, d = 23$$

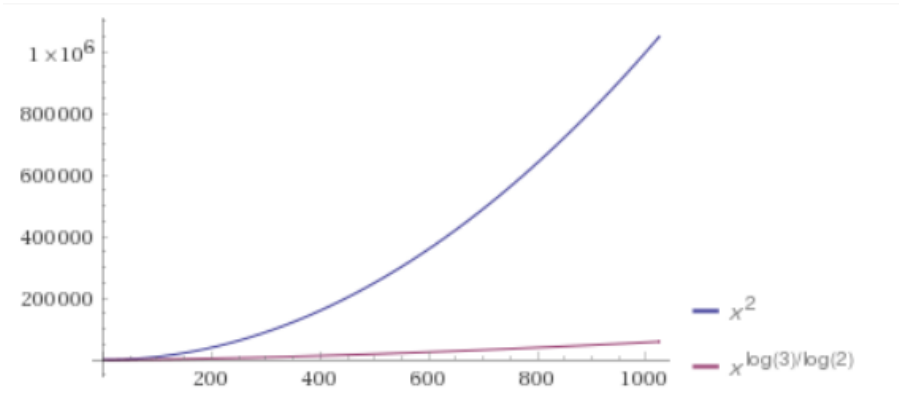
1. $ac = 1365$
2. $bd = 1012$
3. $(a+b)(c+d) = (109 \cdot 44) = 4796$
4. $e = 4796 - 1365 - 1012 = 2419$
5. $13650000 + 241900 + 1012$

$$13892912$$

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

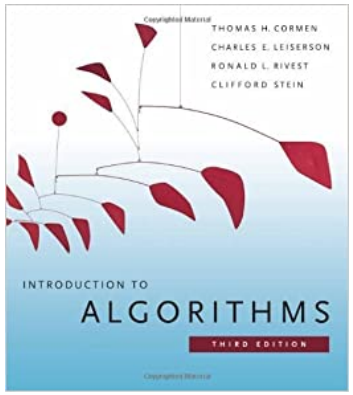
- Tornar-se um programador melhor
- Afiar a sua habilidade matemática e analítica
- Pensar de forma algorítmica
- Familiaridade com a literatura de ciência da computação, algoritmos clássicos, teorema fundamentais, personalidades importantes.
- Vai ajudar a passar em entrevistas de emprego!

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.

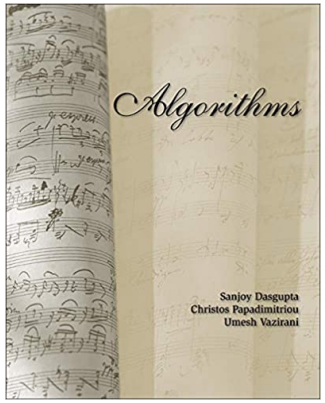
Referências

- 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`

Comunicação

- Site da disciplina https://hokama.com.br
- Email hokama@unifei.edu.br, melhor enviar pela sua conta @unifei.edu.br e usar a tag [CTCO04] no início do assunto.
- Nas aulas é obrigatório o uso de uma conta @unifei.edu.br

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)

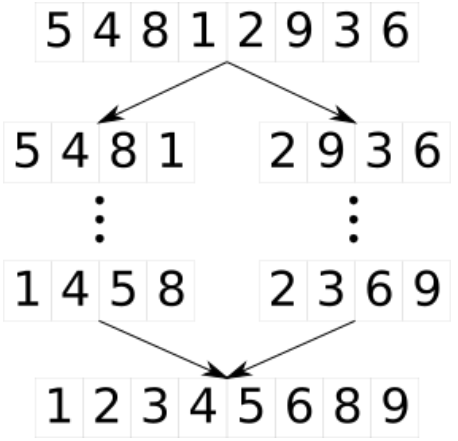
Por que veremos novamente o MergeSort?

- 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

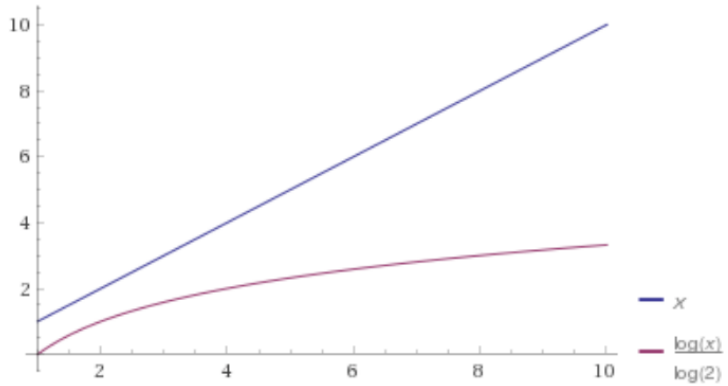
- Analisar o MergeSort é um pouco mais desafiador pois cada problema faz 2 chamadas recursivas, causando uma explosão de subproblemas. A boa notícia é que cada vez que dividimos um subproblema em dois, cada um deles tem metade do tamanho.
- De fato é esse equilíbrio entre a quantidade de subproblemas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

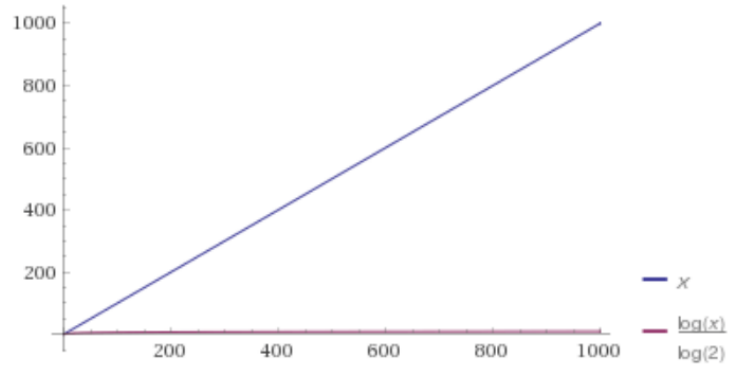
Complexidade do MergeSort

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.

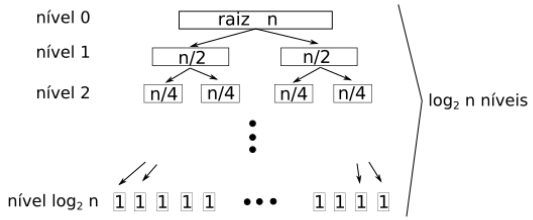




Complexidade do MergeSort

- 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

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

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

□

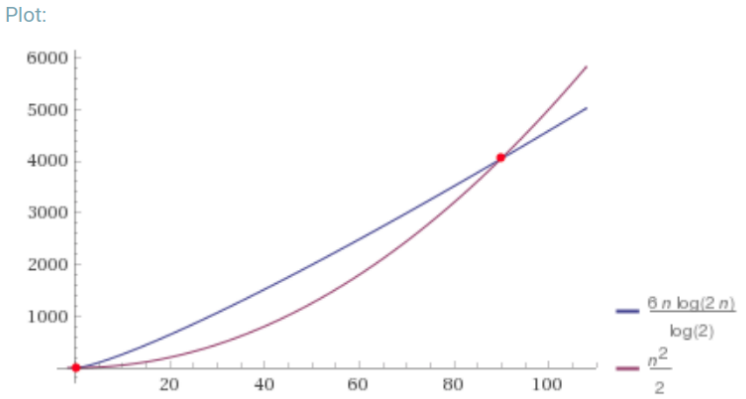
Princípios da Análise de Algoritmos

- 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)

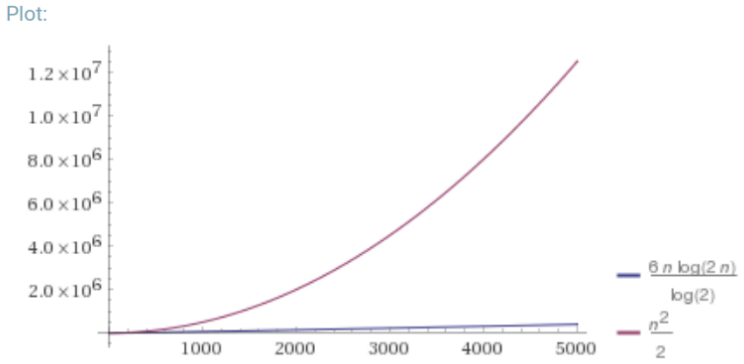
Princípios da Análise de Algoritmos

- 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



Princípios da Análise de Algoritmos



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

- 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 $f(n)$.

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.

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!

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!

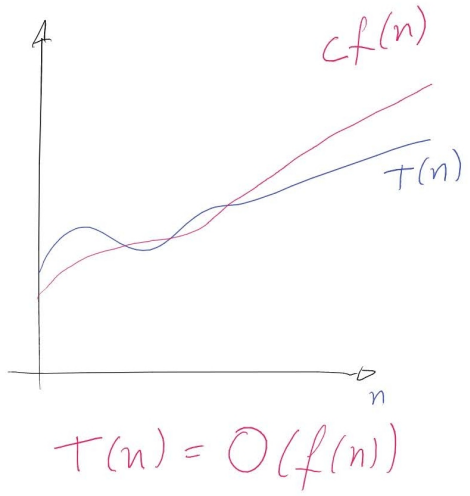
Qual o tempo de execução desse algoritmo?

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

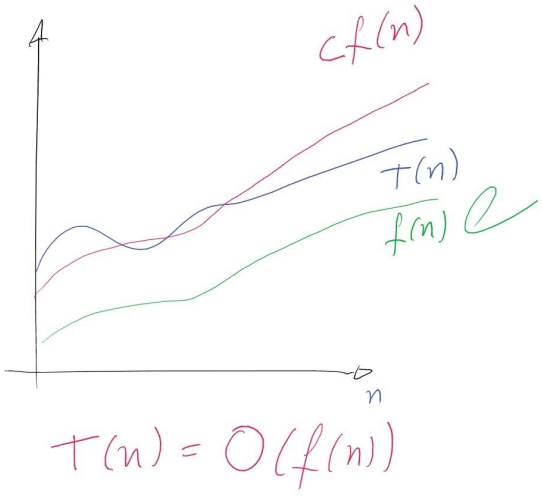
Exercício: É possível fazer esse algoritmo mais eficiente? Como?

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 $c = |a_k| + |a_{k-1}| + \ldots + |a_1| + |a_0|$.

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

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,

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

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

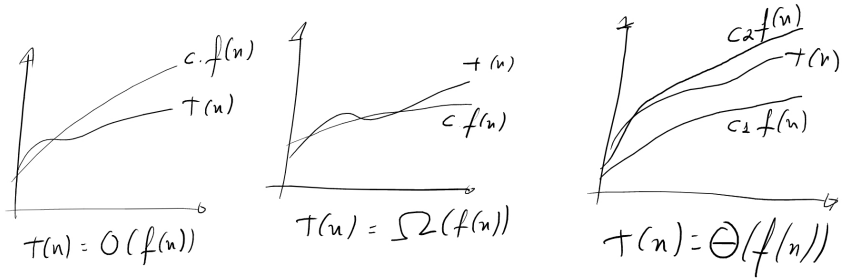
Analogia

- 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 $n \geq n_0$.

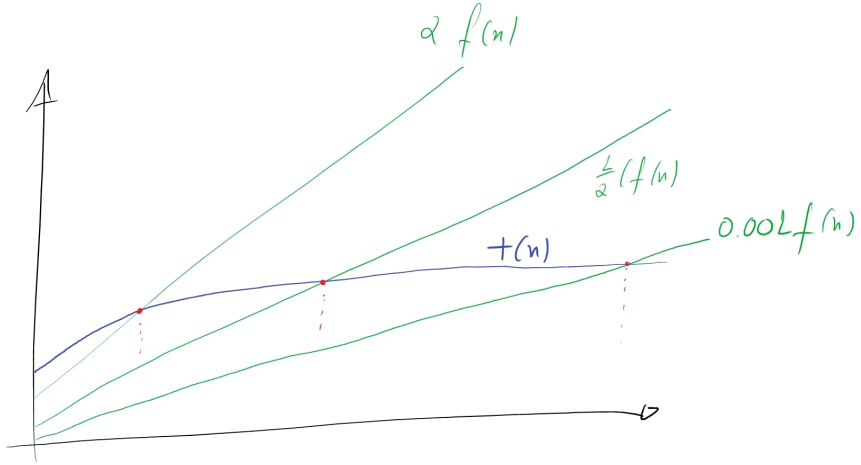
Definição

$T(n) = \Theta(f(n))$ 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 $n_0 = 1$ 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:

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

Usando limites

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

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

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

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

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

Exemplo

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

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

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

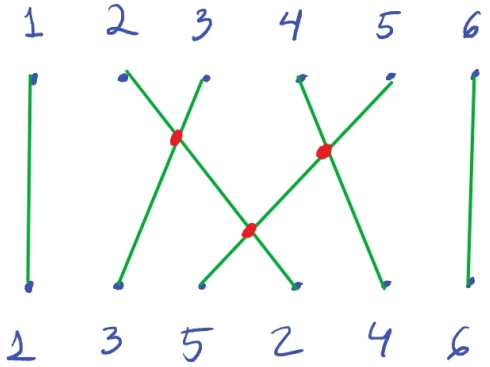
$$A = (1, 3, 5, 2, 4, 6)$$

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

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

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;
```

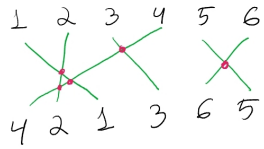
Qual a complexidade desse algoritmo?

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 D;
```


- Se não houver inversões do tipo split, o que 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.

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 D;
```

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 (D, 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);
```

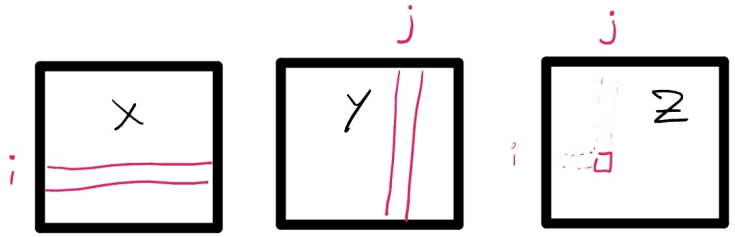

- Assim como no MergeSort, a árvore de recursão de SortCount tem $\log n$ níveis e cada nível executa $O(n)$ operações, então a complexidade total de SortCount é

$$O(n \log n)$$

- 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

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

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

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

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:

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

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

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

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

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

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:
- $T(n) = O(n^{\log_2 4}) = O(n^2)$

Algoritmo de Karatsuba

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

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

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

MergeSort:

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

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: $\frac{n}{b^j}$

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

$$= c \cdot n^d \cdot \left(\frac{a}{b^d}\right)^j$$

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

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

- 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:

  $T(n) \leq c \cdot n^d \cdot \frac{1}{1-r}$ 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

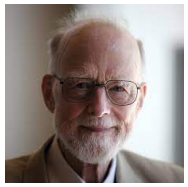
$$T(n) \leq c \cdot n^d \cdot c' \cdot r^{\log_b n} = c \cdot n^d \cdot c' \cdot \frac{a^{\log_b n}}{b^{\log_b n.d}} = c \cdot n^d \cdot c' \cdot \frac{a^{\log_b n}}{n^d} =$$

$$= c \cdot \cancel{n^d} \cdot c' \cdot \frac{a^{\log_b n}}{\cancel{n^d}}$$

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

$T(n) = O(n^{\log_b a})$ □

Sir Charles Antony Richard Hoare



- Cientista da Computação Britânico nascido em 1934
- Ganhador de diversos prêmios na Área da Computação
- I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.
- Inventou em 1959 (e publicou em 1961) o QuickSort.

QuickSort

- Porque ver e rever o QuickSort?
  - Um dos Greatest Hits da ciência da computação.
  - Muito usado na prática.
  - Memória extra constante.
  - Análise muito interessante.

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 $i, j \in \{1, \ldots, n\}$.

Partição

O QuickSort depende fortemente da operação de Partição cuja a ideia é particionar o arranjo em torno de um pivô:

- Escolha um pivô.
- Reorganize os elementos do arranjo de forma que:
  - A esquerda do pivô tenha os elementos menores que o pivô.
  - A direita do pivô tenha os elementos maiores que o pivô.

Exemplo

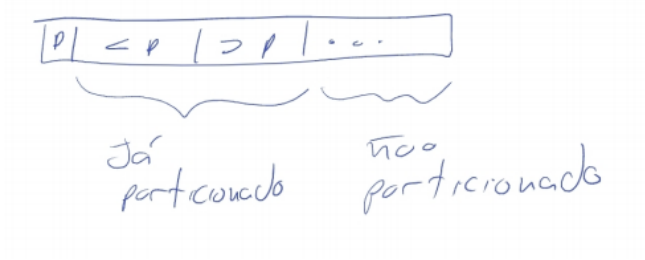
No vetor (3, 8, 2, 5, 1, 4, 7, 6), se escolhemos 3 como pivô.
Uma partição seria: (2, 1, 3, 6, 7, 4, 5, 8)

Observe que depois de uma partição o pivô estará na sua posição correta. Note também que não nos importamos com a posição relativa de cada uma das partes esquerda e direita.

- Como particionar em $O(n)$ usando memória extra?
- Podemos fazer a Partição em tempo $O(n)$ sem usar memória extra.

Ideia:

- Manter uma porção do arranjo já particionado, e o restante ainda não particionado



- Percorrer uma única vez o arranjo

Algoritmo 1: Partição

Entrada: Um arranjo A, índices l e r

Saída: O mesmo arranjo mas particionado

```
1 p = A[l];
2 i = l + 1;
3 para j = l + 1 até r faça
4     se A[j] < p então
5         Troca A[j] e A[i];
6         i = i + 1;
7 Troca A[l] e A[i − 1];
8 devolva A;
```


Provas por Indução - Revisão

Vamos revisar o que é uma prova por indução, que utiliza o seguinte axioma:

Principio da Indução Completa

Seja $P(n)$ uma sentença aberta sobre $\mathbb{N}$. Suponha que:

1. $P(0)$ é verdade, e
2. para todo $k \in \mathbb{N}$, $((\forall i \in \mathbb{N}) i \leq k \rightarrow P(i)) \rightarrow P(k+1)$

Então $P(n)$ é verdade para toda $n \in \mathbb{N}$.

Ou seja se $P(0), P(1), \ldots, P(k)$ é verdade então $P(k+1)$ também é verdade.

Para usar esse método podemos utilizar o seguinte roteiro:

- *Base da Indução*: Provar que $P(0)$ é verdade.
- *Hipótese de Indução*: Supor que $P(i)$ é verdade para todo $i \leq k \in \mathbb{N}$.
- *Passo da Indução*: Provar que $P(k+1)$ é verdade.

Exemplo: Provar que, para todo $n \geq 0$:

$$1 + 3 + 5 + \ldots + (2n + 1) = (n + 1)^2$$

- Base da Indução: $P(0)$ é verdade pois

$$1 = (0 + 1)^2 = 1$$

- Hipótese de Indução: Suponha que $P(i)$ é verdade para todo $i \leq k \in \mathbb{N}$.
- Passo da Indução: Provar que $P(k + 1)$ é verdade, ou seja que:

$$1 + 3 + 5 + \ldots (2k + 1) + (2(k + 1) + 1) = ((k + 1) + 1)^2$$

$$[1+3+5+\ldots+(2k+1)]+(2(k+1)+1)) = [(k+1)^2]+(2(k+1)+1))$$

$$= k^2+2k+1+2k+3 = k^2+4k+4 = (k+2)^2 = ((k+1)+1)^2 \quad \square$$

Corretude do algoritmo partição

Invariante de laço: Os elementos $A[l + 1], \ldots, A[i - 1]$ são todos menores que o pivô, e os elementos $A[i], \ldots, A[j - 1]$ são todos maiores que o pivô.

- **Inicialização:** Por vacuidade, antes da primeira iteração a invariante vale.
- **Manutenção:** Em cada iteração verificamos se $A[j]$ é menor que p e nesse caso passamos ele para a primeira porção, incrementando o i. E dessa forma a invariante de mantêm.
- **Termino:** Ao final do laço trocamos o pivô pelo ultimo elemento da primeira porção (ou ele mesmo caso a primeira porção seja vazia) e o arranjo estará particionado.

QuickSort

Algoritmo 2: QuickSort

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

1 **se** $n \leq 1$ **então** devolva A;
2 p = EscolhePivo(A, n);
3 Particionar A em torno de p;
4 Recursivamente ordenar a parte esquerda;
5 Recursivamente ordenar a parte direita;
6 devolva A;

QuickSort

Algoritmo 3: QuickSort

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

1 **se** $n \leq 1$ **então** devolva A;
2 p = EscolhePivo(A, n);
3 Particionar A em torno de p;
4 Recursivamente ordenar a parte esquerda;
5 Recursivamente ordenar a parte direita;
6 devolva A;

Qual o tempo de execução da fase de combinação?

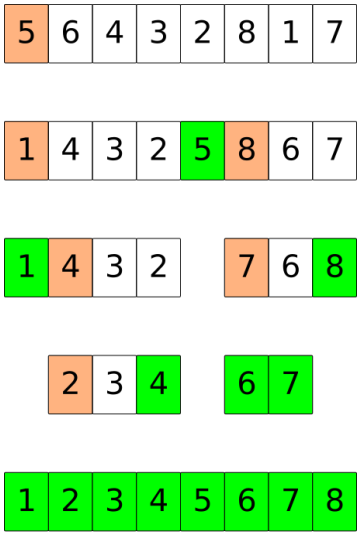
Teorema

O algoritmo QuickSort ordena corretamente um vetor.

Iremos provar a corretude do QuickSort por indução no comprimento n do arranjo. Considere a seguinte sentença aberta $P(n)$: "O QuickSort corretamente ordena arranjos de comprimento n.

- *Base da Indução*: Quando $n = 1$ o algoritmo não faz nada e o vetor está trivialmente ordenado. Portando $P(1)$ é verdade.
- *Hipótese de Indução*: $P(k)$ é verdade para $k < n$
- *Passo da Indução*: Agora queremos provar $P(k+1)$ é verdade.
  - O algoritmo escolhe algum pivô p e particiona o arranjo
  - O pivô termina já em seu lugar correto
  - O algoritmo recursivamente ordena a primeira e a segunda porção que necessariamente são menores que k, logo pela hipótese de indução, são ordenadas corretamente. □

QuickSort - revisão

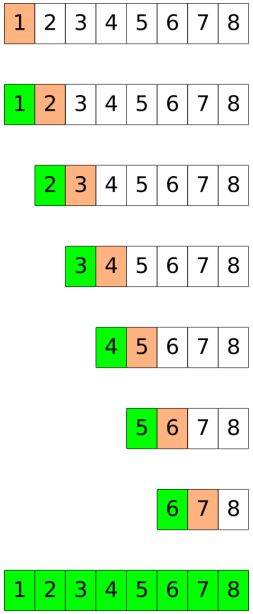


QuickSort

- Vamos analisar a complexidade do QuickSort.
- O que acontece com o QuickSort se a escolha de pivô for muito ruim?
- Como seria a árvore de recursão e complexidade no pior caso? Por exemplo se você tiver um arranjo já ordenado, e sempre escolhermos o primeiro elemento.

Suponha que implementamos o QuickSort, de forma que o pivô é sempre o primeiro elemento do arranjo. Qual é o tempo de execução desse algoritmo em um vetor de entrada que já está ordenado?

a. Não é possível estimar
b. $\Theta(n)$
c. $\Theta(n \log n)$
d. $\Theta(n^2)$



QuickSort

- Nesse caso iremos dividir em duas partes, uma vazia e uma com $n-1$ elementos. E o trabalho da partição será pelo menos:

$$n + (n-1) + (n-2) + \ldots + 1 = \Theta(n^2)$$

- O tempo de execução do QuickSort depende da qualidade do pivô escolhido.
- Um bom pivô é aquele que divide os problemas em partes de tamanho parecido,
- enquanto um pivô ruim é aquele que deixa duas partes muito desiguais.

QuickSort - o caso bom

- Se dividirmos o arranjo sempre no máximo, ou seja, na metade.
- Isso aconteceria se encontrássemos a mediana.
- Teremos duas partes com menos que metade dos elementos
- Podemos limitar superiormente pela mesma recorrência do MergeSort
- Sabemos então que no melhor caso QuickSort é $O(n \log n)$

Seja $T(n)$ o tempo de execução desse algoritmo em um vetor de tamanho n. Então

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Pelo teorema mestre:

$$T(n) = O(n \log n)$$

Suponha que implementamos o QuickSort, de forma que (magicamente) sempre escolhemos a mediana como pivô. Qual é o tempo de execução desse algoritmo?

a. Não é possível estimar
b. $\Theta(n)$
c. $\Theta(n \log n)$
d. $\Theta(n^2)$

- Infelizmente não é possível encontrar a mediana em $O(1)$ para garantir esse tempo de execução.
- Felizmente não é necessário!

Aleatorização do QuickSort

- A aleatorização de algoritmos é uma ferramenta importante da bagagem de qualquer profissional da computação.
- Veremos como aplicar essa técnica no QuickSort e as vantagens que ela pode trazer.
- A ideia é escolher é escolher cada pivô aleatoriamente com a **mesma probabilidade**. E assim escolher um pivô "razoavelmente bom", em uma frequência "razoavelmente alta".
- Digamos que um pivô razoavelmente bom seria um que divida o arranjo em 25-75%, que é suficiente para garantir o tempo de execução de $O(n \log n)$
- Como seria a árvore de recursão nesse caso?
- Note que metade dos elementos, se escolhidos como pivô, resultam numa divisão de 25-75% ou melhor.

Revisão(zinha) de Probabilidade

Para completar a análise do QuickSort Aleatorizado e outros algoritmos, precisamos relembrar alguns conceitos de probabilidade:

- Espaço Amostral
- Eventos
- Variáveis Aleatórias
- Esperança

Espaço Amostral

- **Espaço Amostral** Ω = todos os possíveis resultados de um evento aleatório.
- cada resultado $i \in \Omega$ tem probabilidade $p(i) \geq 0$.
- **Restrição**: $\sum_{i \in \Omega} p(i) = 1$, ou seja, com certeza um dos resultados de Ω vai acontecer.

Exemplo 1

Rolar dois dados de 6 lados.
$\Omega = \{(1,1),(2,1),(3,1),\ldots,(5,6),(6,6)\}$ (pares ordenados)
$p(i) = \frac{1}{36}$ para todos $i \in \Omega$

Exemplo 2

Escolher o índice do pivô aleatório da primeira iteração do QuickSort.
$\Omega = \{1,2,\ldots,n\}$
$p(i) = \frac{1}{n}$ para todo $i \in \Omega$

Evento

- Um **evento** é um subconjunto $S \subseteq \Omega$.
- A probabilidade de um evento S é

$$\sum_{i \in S} p(i).$$

- Considere o evento "a soma dos dados é 7". Qual é a probabilidade desse evento?
  a. $1/36$
  b. $1/12$
  c. $1/6$
  d. $1/2$
- $S = \{(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)\}$
- $Pr[S] = 6/36 = 1/6$

- Considere o evento "o pivô aleatório escolhido induz uma divisão de pelo menos $25 - 75\%$ ou melhor". Qual é a probabilidade desse evento?
  a. $1/n$
  b. $1/4$
  c. $1/2$
  d. $3/4$
- $S =$ qualquer escolha que não seja os $1/4$ menores ou os $1/4$ maiores.

$$Pr[S] = \frac{\frac{n}{2}}{n} = \frac{n}{2} \cdot \frac{1}{n} = \frac{1}{2}$$

Variáveis Aleatórias

- Uma **Variável Aleatória** X é uma função:

$$X : \Omega \to \mathbb{R}$$

- Exemplo 1: A soma dos dois dados, ex: $X((2,5)) = 7$
- Exemplo 2: O valor do maior dado, ex: $Y((2,5)) = 5$
- Exemplo 3: Tamanho do subproblema passado na primeira chamada recursiva do QuickSort

Esperança

- Seja $X : \Omega \to \mathbb{R}$ uma Variável Aleatória.
- A **Esperança** $E[X]$ de X é o valor médio de X ponderado pela probabilidade.

$$E[X] = \sum_{i \in \Omega} X(i) \cdot p(i)$$

- Qual a esperança da soma de dois dados?
  a. 6.5
  b. 7
  c. 7.5
  d. 8

x	2	3	4	5	6	7	8	9	10	11	12
S'											
$\lvert S' \rvert$	1	2	3	4	5	6	5	4	3	2	1

$$E[X] = \frac{1}{36} \cdot 2 + \frac{2}{36} \cdot 3 + \frac{3}{36} \cdot 4 + \frac{4}{36} \cdot 5 + \frac{5}{36} \cdot 6 + \frac{6}{36} \cdot 7 + \frac{5}{36} \cdot 8 + \frac{4}{36} \cdot 9 + \frac{3}{36} \cdot 10 + \frac{2}{36} \cdot 11 + \frac{1}{36} \cdot 12$$

$$= \frac{2 + 6 + 12 + 20 + 30 + 42 + 40 + 36 + 30 + 22 + 12}{36}$$

$$= \frac{252}{36} = 7$$

- Qual a esperança do tamanho do subproblema passado na primeira chamada recursiva do QuickSort?
  a. $n/4$
  b. $n/3$
  c. $(n-1)/2$
  d. $3n/4$
- Seja X o tamanho do subproblema.

$$\begin{aligned} E[X] &= \frac{1}{n} \cdot 0 + \frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 2 + \ldots + \frac{1}{n} \cdot (n-1) \\ &= \frac{1 + 2 + \ldots + (n-1)}{n} \\ &= \frac{(n-1)(1+n-1)/2}{n} \\ &= \frac{n^2 - n/2}{n} = \frac{n^2 - n}{2} \cdot \frac{1}{n} \\ &= \frac{n-1}{2} \end{aligned}$$

Linearidade da Esperança

Lema

Sejam $X_1, \ldots, X_n$ variáveis aleatórias definidas em Ω. Então:

$$E\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} E[X_j]$$

- Vale mesmo quando as variáveis não são independentes!
- Exemplo: X_1 e X_2 são variáveis aleatórias que dizem o valor do primeiro e do segundo dado.

$$E[X_j] = \frac{1}{6}(1 + 2 + 3 + 4 + 5 + 6) = 3.5$$

- Qual o valor esperado para a soma de dois dados?

$$E[X] = E[X_1 + X_2] = E[X_1] + E[X_2] = 3.5 + 3.5 = 7.$$

Análise - QuickSort

Teorema

Para qualquer entrada de comprimento n, o tempo esperado de execução do QuickSort (com pivôs aleatórios) é $O(n \log n)$.

- Primeiramente considere como entrada um arranjo A de comprimento n.
- Espaço amostral: Ω = todos as possíveis sequências de escolhas de pivôs no QuickSort.
- Variável aleatória: para qualquer $\sigma \in \Omega$, $C(\sigma)$ = número de comparações entre dois elementos do arranjo feito pelo algoritmo QuickSort dado as escolhas σ. Note que o tempo de execução total do QuickSort é dominado por esse número.
- O objetivo então é mostrar que $E[C] = O(n \log n)$

- Notação:
  - z_i = i-ésimo menor elemento
  - ou seja, z_i não é o elemento que está originalmente na i-ésima posição do vetor, mas sim o elemento que vai ocupar a i-ésima posição no vetor quando ordenado.

$$(\underbrace{6}_{z_2}, \underbrace{10}_{z_4}, \underbrace{8}_{z_3}, \underbrace{2}_{z_1})$$

  - para uma escolha σ, e $i < j$, seja $X_{ij}(\sigma)$ = número de vezes que z_i e z_j são comparados.
- Fixado dois elementos da entrada, quantas vezes eles podem ser comparados?
  a. 1
  b. 0 ou 1
  c. 0, 1 ou 2
  d. algo entre 0 e $n-1$

Algoritmo 13: QuickSort

Entrada: Um arranjo A, índices l e r
Saída: Um arranjo com os mesmos elementos de A porém ordenados

```
1 se r − l ≤ 0 então devolva A;
2 p = Partição(A, l, r);
3 QuickSort(A, l, p-1);
4 QuickSort(A, p+1, r);
5 devolva A;
```

Algoritmo 14: Partição

Entrada: Um arranjo A, índices l e r
Saída: O mesmo arranjo mas particionado

```
1 Coloca o pivot em A[l] ;
2 pivot = A[l] ;
3 i = l + 1;
4 para j = l + 1 até r faça
5     se A[j] < pivot então
6         Troca A[j] e A[i];
7         i = i + 1;
8 Troca A[l] e A[i − 1];
9 devolva i − 1;
```


- Podemos então facilmente escrever C em função de X

$$C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}(\sigma), \qquad \forall \sigma \in \Omega$$

- Pela linearidade da esperança (lembrando que ela se aplica mesmo se as variáveis não forem independentes)

$$E[C] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}].$$

- Como:

$$\begin{aligned} E[X_{ij}] &= 0 \cdot Pr[X_{ij} = 0] + 1 \cdot Pr[X_{ij} = 1] \\ &= \cancel{0 \cdot Pr[X_{ij} = 0]} + 1 \cdot Pr[X_{ij} = 1] \\ &= Pr[X_{ij} = 1] = Pr[z_i \text{ e } z_j \text{ serem comparados}] \end{aligned}$$

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ e } z_j \text{ serem comparados}]$$

Lema

Para todo $i < j$, $Pr[z_i \text{ e } z_j \text{ serem comparados}] = \frac{2}{(j-i+1)}$

- Fixe z_i, z_j com $i < j$.
- Considere o conjunto $\{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$.
- Desde que nenhum desses números seja escolhido como pivô, todos serão passados juntos para a mesma chamada recursiva.
- Considere então o primeiro entres $\{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$ que é escolhido como pivô:
  - se z_i ou z_j for escolhido primeiro, eles serão escolhidos.
  - se escolher um dos outros então z_i e z_j nunca serão comparados.
- como a escolha é aleatória qualquer um dos elementos do conjunto tem a mesma chance de ser escolhido primeiro (do conjunto) e portando a chance deles serem comparados é

$$2/(j-i+1)$$

□

Então

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr[z_i \text{ e } z_j \text{ serem comparados}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}$$

$$E[C] = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{(j-i+1)}$$

.

- Agora note que a soma interna

$$\sum_{j=i+1}^{n} \frac{1}{(j-i+1)} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n-i+1}$$

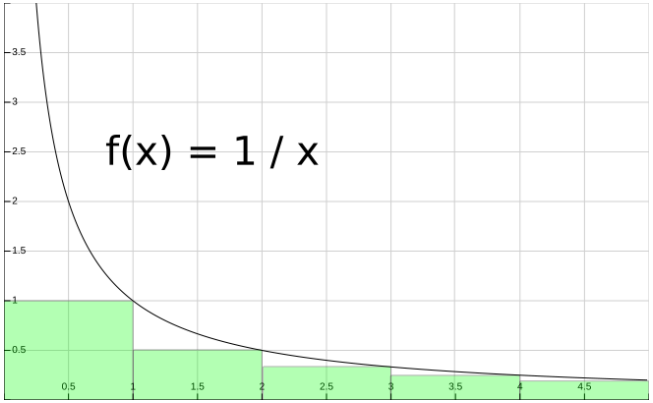
- e ela obtêm a maior quantidade de termos (e o maior valor) quando $i = 1$
- A maior soma então é

$$\sum_{j=2}^{n} \frac{1}{j} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$$

Limitamos então superiormente a esperança por:

$$\begin{aligned} E[C] &= 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{(j-i+1)} \\ &\leq 2 \cdot n \cdot \sum_{j=i+1}^{n} \frac{1}{(j-i+1)} \\ &\leq 2 \cdot n \cdot \sum_{k=2}^{n} \frac{1}{k} \end{aligned}$$

$$E[C] \leq 2 \cdot n \cdot \sum_{k=2}^{n} \frac{1}{k}$$



$$E[C] \leq 2 \cdot n \cdot \sum_{k=2}^{n} \frac{1}{k}$$

A soma agora pode ser limitada superiormente por:

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n} \frac{1}{x} dx = \ln x \Big|_{1}^{n} = \ln n - \cancel{\ln 1} = \ln n$$

.

Portanto:

$$E[C] \leq 2 \cdot n \cdot \ln n$$

e portanto:

O tempo de execução esperado do QuickSort é $O(n \log n)$ □

Problema da Seleção

Problema da Seleção

Dado um arranjo A com n números, e um inteiro $i \in \{1, 2, \ldots, n\}$, encontrar a i-ésima estatística de ordem, ou seja, o i-ésimo menor número de A.

- Comumente desejamos encontrar a mediana de um conjunto.
- A mediana normalmente é menos afetada por corrupção em um conjunto de dados.
- Como resolver esse problema em $O(n \log n)$?
  - Ordenar A e devolver $A[i]$.
  - Isso é uma chamada "redução", reduzir uma instância de um Problema A para uma instância do Problema B, usar um algoritmo para resolver a instância do Problema B, e usar o resultado para responder o problema A. Esse conceito será visto com detalhes no futuro.

Problema da Seleção

- Será que podemos fazer melhor? 🤔
- Certamente não vamos encontrar nada melhor que linear, pois pelo menos precisamos ler todos os valores uma vez. Mas será que conseguimos algo mais próximo de $O(n)$ do que $O(n \log n)$?

Problema da Seleção

- Queremos encontrar um algoritmo de tempo esperado linear 🤞
  - Ideia: usar o algoritmo Partição do QuickSort.

$$(5, 3, 8, 9, 10, 7, 1, 2)$$

partição

$$(\underbrace{2, 3, 1}_{\leq 5}, 5, \underbrace{10, 7, 8, 9}_{\geq 5})$$

- Suponha que procuramos o 6º menor elemento.
- Suponha que escolhemos um pivô qualquer
- Aplicamos a partição e descobrimos que ele é o 4º menor elemento
- Podemos então procurar o 2º menor elemento da segunda parte do arranjo.

Algoritmo 15: Partição

Entrada: Um arranjo A, índices l e r
Saída: O mesmo arranjo mas particionado

```
1 Coloca o pivot em A[l] ;
2 pivot = A[l] ;
3 i = l + 1;
4 para j = l + 1 até r faça
5     se A[j] < pivot então
6         Troca A[j] e A[i];
7         i = i + 1;
8 Troca A[l] e A[i − 1];
9 devolva i − 1;
```

Algoritmo 16: SelecaoR

Entrada: Um arranjo A, índices l e r, inteiro i
Saída: A i-ésima estatística de ordem

```
1 se l == r então devolva A[i];
2 p = Partição(A, l, r);
3 j = p − l + 1 //p é o j-ésimo
  menor valor do array atual;
4 se j = i então devolva A[p];
5 se j > i então devolva
  SelecaoR(A, l, p − 1, i);
6 se j < i então devolva
  SelecaoR(A, p + 1, r, i − j);
```

SeleçãoR - Análise

- Corretude = Análogo ao QuickSort.
- Complexidade:
  - Antes vamos pensar:
  - Qual o tempo de execução de pior caso? Ou seja, se na partição sempre dividirmos da forma mais desbalanceada possível. $O(n^2)$.
  - Qual o tempo de execução se na partição sempre dividirmos da forma mais balanceada possível.

$$T(n) \leq T(n/2) + O(n)$$

  - Teorema mestre: como $a < b^d$ caímos no caso 2. E portanto $T(n) = O(n)$.
- Assim como no QuickSort um pivô aleatório era suficiente para chegar perto do desempenho da mediana, será que nesse algoritmo essa estratégia vai funcionar?

SeleçãoR - Complexidade

Teorema

Para qualquer entrada de comprimento n, o tempo de execução esperado de SelecaoR é $O(n)$.

- Para facilitar a análise, vamos dividir a execução do SeleçãoR em **Fases**.
- Diremos que SelecaoR está na Fase j se o tamanho atual do arranjo está entre $\left(\frac{3}{4}\right)^{j+1} n$ e $\left(\frac{3}{4}\right)^{j} n$
- Por exemplo, se $n = 1000$,

Fase	Tamanho
0	(750, 1000]
1	(562, 750]
2	(421, 562]
3	(316, 421]
...	...

- O algoritmo Partição realiza $\leq cn$ operações para alguma constante $c > 0$.
- Seja X_j uma variável aleatória do número de chamadas recursivas durante a Fase j

$$\text{Tempo de execução do SeleçãoR} \leq \sum_{\text{Fases } j} X_j \cdot \overbrace{c \cdot \underbrace{\left(\frac{3}{4}\right)^j n}_{\text{operações por subproblema na Fase j}}}^{\text{tamanho máximo do vetor durante Fase j}}$$

$$\text{Tempo de execução do SeleçãoR} \leq \sum_{\text{Fases } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n$$

- Se SeleçãoR escolher um pivô que faz uma divisão de $25 - 75\%$ ou melhor,

$< p$	p	$> p$

- então a Fase atual termina!
- Já que por definição o subproblema (mesmo que for a maior porção) é no máximo 75%
- Lembre que metade dos elementos escolhidos vão dar uma divisão nessa proporção. E portanto a probabilidade de migrar para a próxima fase é de 50%
- Pense então que essa é a mesma probabilidade de jogar uma moeda e obter "cara".
- $E[X_j] \leq$ número esperado de vezes que você precisa jogar uma moeda para obter "cara".

- $E[X_j] \leq E[N]$, N = número esperado de vezes que você precisa jogar uma moeda para obter "cara".

$$
\begin{aligned}
E[N] &= 1 + \frac{1}{2}E[N] \\
E[N] - \frac{1}{2}E[N] &= 1 + \frac{1}{2}E[N] - \frac{1}{2}E[N] \\
\frac{1}{2}E[N] &= 1 \\
\frac{1}{2}E[N] \cdot 2 &= 1 \cdot 2 \\
E[N] &= 2
\end{aligned}
$$

Tempo de execução do SeleçãoR $\leq \sum_{\text{Fases } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n$

$$E\left[\sum_{\text{Fases } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right] = cn \cdot E\left[\sum_{\text{Fases } j} X_j \cdot \left(\frac{3}{4}\right)^j\right]$$

$$(L.E.) = cn \cdot \sum_{\text{Fases } j} E[X_j]\left(\frac{3}{4}\right)^j \leq 2cn \cdot \sum_{\text{Fases } j}\left(\frac{3}{4}\right)^j$$

$$= 2cn \cdot \overbrace{\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \left(\frac{3}{4}\right)^3 + \ldots\right)}^{\text{P.G. de razão } 3/4}$$

$$= 2cn \cdot \left(\frac{1}{1 - 3/4}\right) = 2cn \cdot 4 = 8cn$$

$E[\text{Tempo de execução do SeleçãoR}] = O(n)$ □

Um limitante inferior para Algoritmos de Ordenação

- Algoritmos de Ordenação como o InsertionSort, BubbleSort, SelectionSort, MergeSort, QuickSort e HeapSort baseiam seu funcionamento em comparação entre seus elementos.
- É o método usado quando não podemos assumir nenhuma propriedade sobre os elementos da entrada.
- Ou quando não podemos acessar diretamente os elementos mas apenas compará-los através de uma função.
- São algoritmos de propósitos gerais pois funcionam para qualquer tipo de entrada.
- Existem algoritmos de ordenação que se baseiam em outro tipo de interação com os dados. BucketSort, CountingSort, RadixSort.
- Mas esses algoritmos partem de alguma premissa sobre os dados. Ex: São inteiros com N dígitos, são valores de ponto flutuante uniformemente distribuídos entre 0 e 1;

Teorema

Todo algoritmo de ordenação baseado em comparações tem um tempo de execução de pior caso $\Omega(n \log n)$.

- Suponha uma entrada qualquer de comprimento n e um algoritmo baseado em comparações que ordena corretamente esse arranjo.
- Seja K o número de comparações feitas pelo algoritmo
- Para cada comparação o algoritmo tem um resultado digamos 0 ou 1
- Portanto o número total de resultados possíveis é 2^K
- O número possível de permutações do arranjo de entrada é $n!$

Se $2^K < n!$ pelo principio da casa dos pombos duas entradas idênticas irão obter os mesmos resultados, o que é um absurdo já que o algoritmo ordena corretamente.
Portanto:

$$\begin{aligned} 2^K &\geq n! \\ &= n(n-1)(n-2)\dots(n/2)\dots 1 \\ &\geq \underbrace{n(n-1)(n-2)\dots(n/2)}_{n/2 \text{ termos}} \\ &\geq \underbrace{(n/2)(n/2)(n/2)\dots(n/2)}_{n/2 \text{ termos}} \\ &= (n/2)^{(n/2)} \end{aligned}$$

$$\begin{aligned} 2^K &\geq (n/2)^{(n/2)} \\ \log 2^K &\geq \log(n/2)^{(n/2)} \\ K \log 2 &\geq (n/2)\log(n/2) \\ K &\geq (n/2)\log(n/2) \end{aligned}$$

Portanto K é $\Omega(n \log n)$. □

- Dessa forma nenhum algoritmo baseado em comparações pode ser melhor que $O(n \log n)$.
- Podemos afirmar que o problema da seleção é mais fácil que o problema da ordenação.

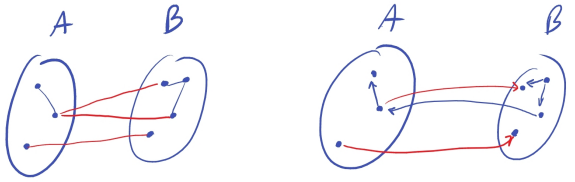
Corte Mínimo em Grafos

- Algoritmo Aleatorizado de Contração

Cortes em Grafos

Definição

Um corte de um Grafo $G = (V, E)$ é uma partição de V em dois conjuntos não vazios A e B.



Uma **aresta de corte** de um corte (A, B) são aquelas com

- um extremo em A e outro em B (em grafos não direcionados)
- com o inicio em A e o final em B (em grafos direcionados)

Cortes em Grafos

Quantos cortes (aproximadamente) um grafo com n vértices tem?

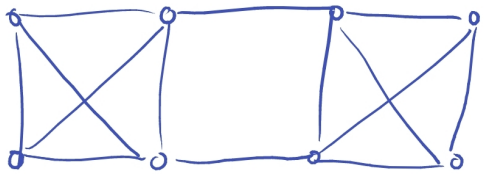
- n
- n^2
- 2^n
- n^n

O Problema do Corte Mínimo

Problema do Corte Mínimo

Dado um Grafo não direcionado $G = (V, E)$. Encontrar um corte com o menor número de arestas.

- O corte com o menor número de arestas em um grafo é o **corte mínimo**
- Arestas paralelas são permitidas.



Quantas arestas de corte tem um corte mínimo no grafo acima?

- 1
- 2
- 3
- 4

Cortes em Grafos

Aplicações:

- Identificar gargalos em redes (de computadores, de estradas, etc)
- Suponha que você queira redundância na sua rede para que a falha em uma (ou duas, ou mais) arestas principais não desconecte sua rede.
- Ou suponha que você quer sabotar um inimigo bombardeando uma estrada desconectando suas cidades.
- Detecção de comunidade em uma rede social. Grupos de usuários conectados entre si, mas com pouca conexão para o restante do grafo.
- Em reconhecimento de imagem, você pode entender cada pixel de uma imagem como um vértice com arestas para seus vizinhos, cada aresta com um peso de acordo com a semelhança dos dois pixels. Um corte minimo pode ser usado para encontrar objetos dentro da imagem.

Algoritmo Aleatorizado de Contração

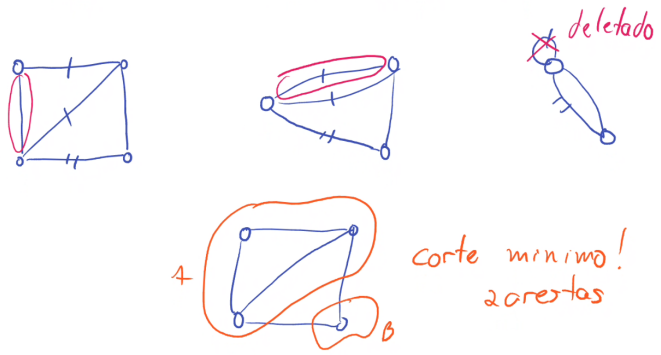
- Publicado por David Ron Karger, em 1993, quando aluno do Doutorado na Universidade de Stanford. (Hoje é professor no MIT)
- Ideia:

Algoritmo 17: Algoritmo Aleatorizado de Contração

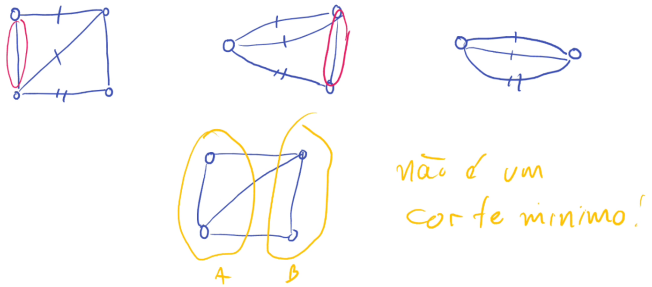
Entrada: Um Grafo G
Saída: Um Corte

```
1 enquanto houverem mais de 2 vértices faça
2     escolha uma aresta aleatória {u, v} com probabilidade uniforme;
3     fundir (contrair) u e v em um único vértice;
4     remover auto-laços ;
5 devolva o corte representado pelos 2 vértices finais;
```


Exemplo - Execução



Exemplo - Execução2



- Algoritmo Aleatorizado de Contração as vezes encontra o corte mínimo, as vezes não.
- Será que um algoritmo assim é útil?
- A probabilidade de encontrar a resposta certa, é maior que zero, mas menos que 1. Mas qual será? 🤔

Revisão(zinha) de Probabilidade - parte2

Probabilidade Condicional

Na primeira revisão de probabilidade vimos:

- Espaço Amostral
- Eventos
- Variáveis Aleatórias
- Esperança
- Linearidade da Esperança

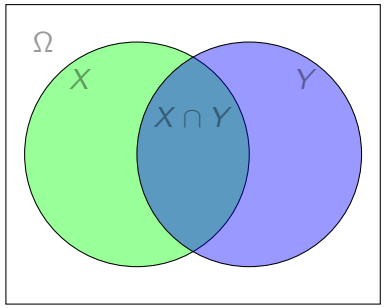
Hoje veremos:

- Probabilidade condicional
- Independência de Eventos e Variáveis Aleatórias

Probabilidade Condicional

Entender a probabilidade de um evento dado um segundo evento.

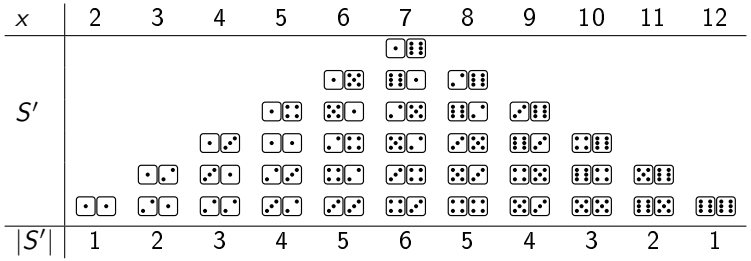
- Considere os eventos $X, Y \subseteq \Omega$.



- Então a probabilidade de X dado Y é:

$$Pr[X|Y] = \frac{Pr[X \cap Y]}{Pr[Y]}$$

- Suponha que você jogue 2 dados, qual a probabilidade de pelo menos um dado ser 1 dado que a soma deles é 7?
  - 1/36
  - 1/6
  - 1/3
  - 1/2
- X = pelo menos um dado é um 1
- Y = a soma de dois dados é 7
- $Y = \{(1,6),(2,5),(3,4),(4,3),(5,2),(6,1)\}$
- $X \cap Y = \{(1,6),(6,1)\}$



$$\begin{aligned} &Pr[\text{um dado ser 1}|\text{a soma é sete}] \\ &= \frac{Pr[\text{um dado ser 1 e a soma ser sete}]}{Pr[\text{soma ser sete}]} \\ &= \frac{2/36}{6/36} = \frac{2}{36}\frac{36}{6} = \frac{2}{6} = \frac{1}{3} \end{aligned}$$

Independência de Eventos

Definição

Eventos $X, Y \subseteq \Omega$ são independentes se e somente se

$$Pr[X \cap Y] = Pr[X].Pr[Y]$$

- Uma definição equivalente é $P[X|Y] = Pr[X]$

Independência de Variáveis Aleatórias

Definição

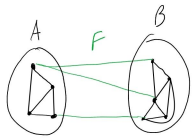
Variáveis aleatórias A e B definidas para um mesmo Ω são independentes se e somente os eventos $A = a$ e $B = b$ são independentes para todo a e b

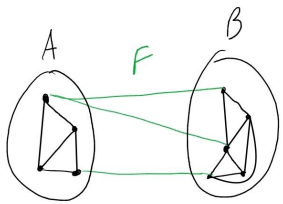
- Uma definição equivalente é $P[A = a \text{ e } B = b] = Pr[A = a] \cdot Pr[B = b]$
- Se as variáveis são independentes $E[A \cdot B] = E[A] \cdot E[B]$ (diferentemente da linearidade da esperança, essa propriedade só se aplica a variáveis independentes)

Algoritmo Aleatorizado de Contração

Qual a probabilidade de sucesso?

- Queremos encontrar um limitante inferior para essa probabilidade. Ou seja queremos mostrar que a probabilidade do algoritmo encontrar um Corte Mínimo não é menor do que um determinado valor.
- Considere um Grafo $G = (V, E)$ com n vértices e m arestas.
- Considere o corte mínimo (A, B), queremos encontrar esse corte! (Podem existir outros cortes mínimos, mas vamos considerar que estamos interessados somente nesse, já que queremos um limitante inferior)
- Seja F o conjunto de arestas de corte em (A, B) e $|F| = k$.





- Se uma aresta de F for escolhida durante o algoritmo, um vértice de A e um vértice de B serão fusionados, causando um corte diferente, e portanto o algoritmo falhará.
- Já se nas $n - 2$ iterações apenas arestas com os dois extremos em A ou os dois extremos em B forem selecionadas, o algoritmo vai ser bem sucedido.
- $Pr[\text{devolver } (A, B)] = Pr[\text{não contrair uma aresta de } F]$

- Queremos saber a probabilidade de nenhuma aresta em F ser contraída no algoritmo.
- Seja S_i o evento de que uma aresta de F foi contraída na iteração i
- Então $\neg S_i$ é o evento de nenhuma aresta de F ser contraída na iteração i
- A probabilidade do nosso algoritmo funcionar será:

$$Pr[\neg S_1 \cap \neg S_2 \cap \neg S_3 \cap \neg S_4 \cap \ldots \cap \neg S_{n-3} \cap \neg S_{n-2}]$$

- Qual é a probabilidade de uma aresta do corte (A, B) ser escolhida na primeira iteração? Sendo n o número de vértices, m o número de arestas e k o número de arestas do corte.
  - k/n
  - k/m
  - k/n^2
  - n/m
- Para as próximas iterações será complicado encontrar essa probabilidade em termos do número de arestas, pois o número de aresta varia de maneira imprevisível.
- Então será útil encontrar um limite para essa probabilidade em termos do número de vértices. Já que esse número se comporta bem: A cada iteração diminuímos em 1 o número de vértices.

Definição

O **grau** de um vértice v é o número de arestas incidentes em v. Normalmente denotado por $\delta(v)$.

- O grau de qualquer vértice em V é pelo menos k. (Do contrário $(\{v\}, V - \{v\})$ seria um corte melhor do que (A, B)).
- Note que

$$\sum_{v \in V} \delta(v) = 2m$$

já que cada aresta contribui em 2 para a soma total dos graus.

$$m = \frac{\sum_{v \in V} \delta(v)}{2} \geq \frac{\sum_{v \in V} k}{2}$$

$$m \geq \frac{kn}{2}$$

Como $m \geq \frac{kn}{2}$, então:

$$Pr[S_1] = \frac{k}{m} \leq \frac{k}{kn/2} = k \cdot \frac{2}{kn} = \frac{2}{n}$$

$$Pr[S_1] \leq \frac{2}{n}$$

- Então a probabilidade do algoritmo falhar na primeira iteração é menor que $2/n$
- Note também que a probabilidade de não falhar é $Pr[\neg S_1] \geq \left(1 - \frac{2}{n}\right) = \frac{n-2}{n}$

- Queremos então saber a probabilidade do algoritmo não contrair uma aresta de F na segunda iteração dado que não contraiu na primeira, ou seja:
- Vejamos o complemento, a evento de contrair uma aresta de F na segunda iteração dado que não contraiu na primeira, ou seja:

$$Pr[S_2|\neg S_1] = \frac{k}{\text{num. arestas restantes}}$$

- Como cada nó restante é um corte, o grau de cada nó é $\geq k$. E portanto o número total de arestas é

$$\text{num. arestas restantes} \geq k(n-1)/2$$

- então

$$Pr[S_2|\neg S_1] \leq \frac{k}{k(n-1)/2} = \frac{2}{n-1}$$

- Para qualquer iteração i a probabilidade de eu contrair uma aresta de F dado que eu não o fiz nas iterações anteriores é análogo.

$$Pr\left[S_i|\bigcap_{j<i}\neg S_j\right] = \frac{k}{\text{num. arestas restantes}}$$

- Como cada nó restante também é um corte, o grau de cada nó é $\geq k$, o número total de nós é $n-i+1$. E portanto o número total de arestas é

$$\text{num. arestas restantes} \geq k(n-i+1)/2$$

- então

$$Pr\left[S_i|\bigcap_{j<i}\neg S_j\right] \leq \frac{k}{k(n-i+1)/2} = \frac{2}{n-i+1}$$

$$Pr\left[S_i|\bigcap_{j<i}\neg S_j\right] \leq \frac{2}{n-i+1}$$

- Dessa forma a probabilidade de **Não** contrair uma aresta de F na iteração i dado que também não contraiu nas anteriores é

$$Pr\left[\neg S_i|\bigcap_{j<i}\neg S_j\right] \geq 1 - \frac{2}{n-i+1} = \frac{n-i+1-2}{n-i+1} = \frac{n-i-1}{n-i+1}$$

$$Pr[\neg S_1 \cap \neg S_2 \cap \neg S_3 \cap \neg S_4 \cap \ldots \cap \neg S_{n-3} \cap \neg S_{n-2}]$$

$$Pr[\neg S_1] \cdot Pr[\neg S_2|\neg S_1] \cdot Pr[\neg S_3|\neg S_1 \cap \neg S_2] \cdot Pr[\neg S_4|\neg S_1 \cap \neg S_2 \cap \neg S_3] \ldots$$
$$\ldots Pr[\neg S_{n-3}|\neg S_1 \cap \ldots \cap \neg S_{n-2}] \cdot Pr[\neg S_{n-2}|\neg S_1 \cap \ldots \cap \neg S_{n-3}]$$

$$\geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \frac{4}{6} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}$$

$$= \frac{\cancel{n-2}}{n} \cdot \frac{\cancel{n-3}}{n-1} \cdot \frac{\cancel{n-4}}{\cancel{n-2}} \cdot \frac{\cancel{n-5}}{\cancel{n-3}} \cdots \frac{\cancel{4}}{6} \cdot \frac{\cancel{3}}{5} \cdot \frac{\cancel{2}}{\cancel{4}} \cdot \frac{1}{\cancel{3}} = \frac{2}{n(n-1)} = \frac{2}{n^2-n}$$
$$= \frac{2}{n^2-n} \geq \frac{2}{2n^2-n} \geq \frac{2}{2n^2} = \frac{1}{n^2}$$

$$Pr[\neg S_1 \cap \neg S_2 \cap \neg S_3 \cap \neg S_4 \cap \ldots \cap \neg S_{n-3} \cap \neg S_{n-2}] \geq \frac{1}{n^2}$$

$$Pr[\neg S_1 \cap \neg S_2 \cap \neg S_3 \cap \neg S_4 \cap \ldots \cap \neg S_{n-3} \cap \neg S_{n-2}] \geq \frac{1}{n^2}$$

- Então a probabilidade do algoritmo funcionar é... muito baixa!
- PORÉM!!! Veja bem. Se você pegasse um corte aleatório a probabilidade dele ser o (A, B) é $\frac{1}{2^n}$

n	$\frac{1}{2^n}$	$\frac{1}{n^2}$
5	$\frac{1}{32}$	$\frac{1}{25}$
10	$\frac{1}{1024}$	$\frac{1}{100}$
100	$\frac{1}{1267650600228229401496703205376}$	$\frac{1}{10000}$

- Podemos fazer um truque para melhorar essa probabilidade. Basta executar o algoritmo várias vezes!

Múltiplas Execuções

- Iremos executar o algoritmo N vezes, e devolver o menor corte encontrado.
- Quantas execuções serão necessárias?
- Seja T_j o evento de que o corte (A, B) seja encontrado na j-ésima tentativa. T_j são independentes.

$$Pr[\text{falhar nas } N \text{ tentativas}] = Pr[\neg T_1 \cap \ldots \cap \neg T_N]$$

$$(ind.) = \prod_{j=1}^{N} Pr[\neg T_j] \leq \left(1 - \frac{1}{n^2}\right)^N$$

$$Pr[\text{falhar nas } N \text{ tentativas}] \leq \left(1 - \frac{1}{n^2}\right)^N$$

- Para qualquer numero real x, $1 + x \leq e^x$

$$\begin{aligned} Pr[\text{falhar nas } N \text{ tentativas}] &\leq \left(1 - \frac{1}{n^2}\right)^N \\ &\leq \left(e^{-1/n^2}\right)^N \\ &= e^{-N/n^2} \\ &= \frac{1}{e^{N/n^2}} \end{aligned}$$

- Para $N = n^2$

$$Pr[\text{falhar nas } n^2 \text{ tentativas}] \leq \frac{1}{e} \approx 37\%$$

- Para $N = n^2 \ln n$

$$Pr[\text{falhar nas } n^2 \ln n \text{ tentativas}] \leq \frac{1}{e^{\ln n}} = \frac{1}{n}$$

Paradigmas de Projeto de Algoritmos

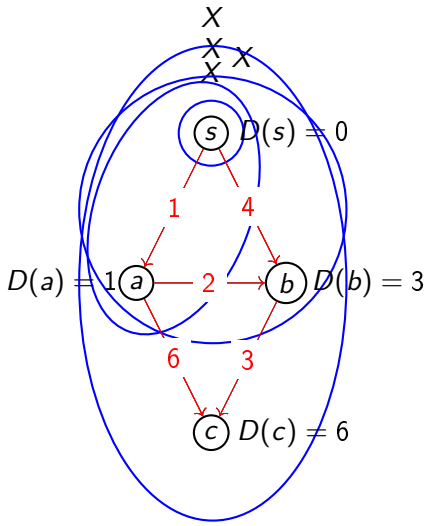
- Divisão e Conquista
- Aleatorização
- **Algoritmos Gulosos**
- Programação Dinâmica
- etc, etc...

Algoritmos Gulosos (Ganânciosos)

- Informalmente, um algoritmo guloso toma uma decisão que parece ser a melhor possível naquele momento (Escolha Gulosa).
- Não faz uma analise global para tomar essa decisão.
- Não se "arrepende" dessa decisão.
- Já vimos pelo menos um Algoritmo Guloso: O Algoritmo de Dijkstra.

Algoritmo de Dijkstra

- Manter um conjunto X com os vértices já processados.
- Considerar os arcos que começam em X e terminam em $V \setminus X$
- Escolher o arco (u, v) que minimiza $D(u) + c_{(u,v)}$, ou seja, o arco que minimiza o caminho para um vértice de $V \setminus X$.
- Note que não existe um caminho menor para chegar em v.
- Computa $D(v)$ e inclui v em X.
- Repita até que todos os vértices estejam em X, ou não tenha nenhum arco.



Algoritmos Gulosos - Vantagens

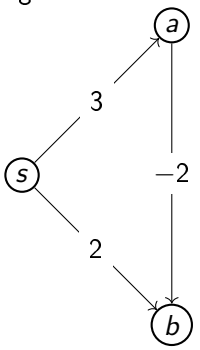
- Vantagens:
  - Normalmente é fácil criar um critério guloso.
  - Normalmente é fácil de implementar.
  - Normalmente a complexidade é baixa e fácil de analisar.
- Desvantagens:
  - A maioria dos critérios gulosos não vai levar a uma solução correta.
  - Provar que um critério guloso leva a solução pode ser trabalhoso.

Algoritmos Gulosos - Contexto

- Estamos interessados em algoritmos que encontrem a solução correta para um dado problema.
- Algoritmos Gulosos também podem ser usados como heurísticas para encontrar boas soluções para problemas de otimização. Não necessariamente encontra a melhor solução, nem por isso estão incorretos.

Algoritmos Gulosos - Não Exemplo

- Suponha que temos o mesmo problema do caminho mínimo, mas agora com arestas de pesos negativos.



- Vamos tentar o mesmo critério guloso de Dijkstra.

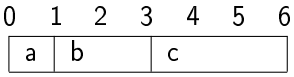
Problema do Escalonamento Ponderado

- Suponha que você tenha uma máquina que faz alguma atividade.
- Trabalhos diferentes, cada trabalho i tem um tempo de execução l_i, e uma prioridade w_i, prioridade maior indica uma tarefa mais importante.
- Seja I o conjunto de todos os trabalhos, e c_i o tempo de conclusão do trabalho i.
- Em que ordem devo executar os trabalhos para minimizar

$$\sum_{i \in I} c_i \cdot w_i$$

Problema do Escalonamento Ponderado - Exemplo

- Três trabalhos $l_a = 1$, $l_b = 2$ e $l_c = 3$



- Qual o tempo de término de cada atividade? $c_a = 1$, $c_b = 3$ e $c_c = 6$.
- Se as prioridades forem, $w_a = 3$, $w_b = 2$ e $w_c = 1$. Quanto é:

$$\sum_{i \in I} c_i \cdot w_i = 1 \cdot 3 + 3 \cdot 2 + 6 \cdot 1 = 15$$

Problema do Escalonamento Ponderado - Critério Guloso

- Vamos tentar criar alguns critérios gulosos.
- Caso 1: Todos os trabalhos tem comprimento igual, mas prioridades diferentes. Qual seria um bom critério guloso? Maior ou menor prioridade na frente?
- Caso 2: Todas as prioridades são iguais, mas o tempo de execução é diferente. Qual seria um bom critério guloso? Curtas ou Longas primeiro?
- Caso 3: Se você tiver um trabalho curto com alta prioridade, e um longo com baixa prioridade? Qual executar primeiro?
- Caso 4: Trabalho longo com alta prioridade e outro curto com baixa prioridade?

Problema do Escalonamento Ponderado - Critério Guloso

- Assim como no Dijkstra vamos tentar criar uma pontuação para guiar a nossa escolha.
- Essa pontuação tem que ser crescer se a prioridade for maior.
- E tem que diminuir quanto mais longa.
- Podemos pensar em pelo menos 2 tentativas claras:
- Tentativa 1: $w_i - l_i$
- Tentativa 2: $\frac{w_i}{l_i}$

Problema do Escalonamento Ponderado - Critério Guloso

a	$l_a = 5$	$w_a = 3$
b	$l_b = 2$	$w_b = 1$

- Tentativa 1: $w_i - l_i$
- Tentativa 2: $\frac{w_i}{l_i}$

Tentativa 1:
$w_a - l_a = 3 - 5 = -2$
$w_b - l_b = 1 - 2 = -1$
Vai colocar b primeiro
$c_a = 7$ e $c_b = 2$
F.O. $= 7 \cdot 3 + 2 \cdot 1 = 23$

Tentativa 2:
$\frac{w_a}{l_a} = \frac{3}{5}$
$\frac{w_b}{l_b} = \frac{1}{2}$
Vai colocar a primeiro
$c_a = 5$ e $c_b = 7$
F.O. $= 5 \cdot 3 + 7 \cdot 1 = 22$

Problema do Escalonamento Ponderado - Corretude

- Suponha que renomeamos os trabalhos de forma que

$$\frac{w_1}{l_1} > \frac{w_2}{l_2} > \frac{w_3}{l_3} > \ldots > \frac{w_n}{l_n}$$

- O critério da Tentativa 2 irá escalonar os trabalhos nessa ordem $\sigma = 1, 2, 3, \ldots, n$.
- Suponha que exista uma outra sequencia $\sigma^* \neq \sigma$ tal que F.O.(σ^*) < F.O.(σ)

Problema do Escalonamento Ponderado - Corretude

- Em σ^* vai existir duas atividades em sequencia, i, j tal que i é executado antes de j mas $i > j$.
  F.O. $(\sigma^*) = A + c_i \cdot w_i + c_j \cdot w_j + B$
- Suponha que trocamos essas duas atividades.

$$\begin{aligned} \text{novo custo} &= A + (c_i + l_j) \cdot w_i + (c_j - l_i) \cdot w_j + B \\ &= A + c_i \cdot w_i + l_j \cdot w_i + c_j \cdot w_j - l_i \cdot w_j + B \end{aligned}$$

- A diferença então é

$$l_j \cdot w_i - l_i \cdot w_j$$

Problema do Escalonamento Ponderado - Corretude

$$l_j \cdot w_i - l_i \cdot w_j$$

- Sabemos que se $i > j$, então

$$\frac{w_i}{l_i} < \frac{w_j}{l_j}$$
$$\frac{(l_i \cdot l_j)w_i}{l_i} < \frac{(l_i \cdot l_j)w_j}{l_j}$$
$$l_j \cdot w_i < l_i \cdot w_j$$

- portanto o que é economizado é maior que o gasto. Portanto a F.O. desse novo escalonamento é MENOR. Mas como σ^* era ótimo isso é um absurdo.

Problema do Escalonamento Ponderado - Complexidade

- Podemos calcular todas as razões em $O(n)$
- Ordenar em tempo $O(n \log n)$
- Portanto a complexidade total do algoritmo é $O(n \log n)$

Representação de Caracteres

- Tradicionalmente caracteres são representados em binários através de uma codificação de tamanho fixo, ou seja, todo caracteres tem o mesmo número de bits, tradicionalmente 8 bits.
- Para simplificar, considere um alfabeto bem pequeno, de 6 letras { a, b, c, d, e, f}.
- Para representar esse alfabeto poderíamos usar uma representação de 3 bits.

a	b	c	d	e	f
000	001	010	011	100	101

- Para representar a palavra "babaca" precisaríamos de 18 bits:

001000001000010000

- Para representar um texto de 100.000 caracteres, precisaríamos de 300.000 bits.

Compressão de Dados

- Suponha agora que queremos comprimir o texto, de forma a ocupar uma quantidade de bits menor.
- Uma ideia é utilizar uma codificação de tamanho variável, ao invés da de tamanho fixo.
- Considere a seguinte codificação (Spoiler: Não vai funcionar)

a	b	c	d	e	f
0	1	00	01	10	11

- Para representar a palavra "babaca" precisaríamos agora de apenas de 7 bits:

1010000

- Para representar um texto de 100.000 caracteres, precisaríamos de menos de 200.000 bits.

Porém não funcionaria!

a	b	c	d	e	f
0	1	00	01	10	11

- Não é possível decodificar uma palavra. Por exemplo:

1010000

- Poderia ser “babaca”. Mas também poderia ser “eeca” ou “bdaaaa”
- O Problema é que o código de um caractere é prefixo de outro caractere.

Solução

- Codificação de tamanho variável livre de prefixo.

a	b	c	d	e	f
0	101	100	111	1101	1100

- Dessa forma uma codificação não será ambígua.

101010101000

- Com essa codificação a palavra “babaca” precisa de 12 caracteres.

Solução

- Utilizando essa codificação em um texto de 100.000 caracteres, em que cada um aparece com a seguinte frequência:

a	b	c	d	e	f
0	101	100	111	1101	1100
45%	13%	12%	16%	9%	5%

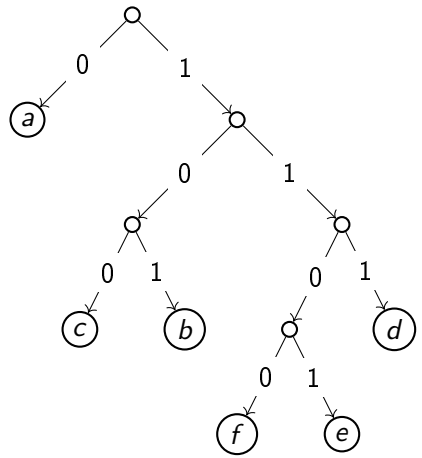
- Seriam necessários:

$$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = 224.000$$

- Compressões de até 90%
- Como escolher uma boa codificação?

Código de Huffman

- Criaremos uma árvore binária que representa uma codificação.
- A cada ramificação para a esquerda temos um bit 0 e cada ramificação a direita temos um bit 1.
- As folhas representam os caracteres.



Código de Huffman

- Construindo a árvore.

a	b	c	d	e	f
45%	13%	12%	16%	9%	5%

Código de Huffman

Algoritmo 18: Algoritmo de Huffman

Entrada: Um alfabeto C e suas frequências
Saída: Um código de tamanho variável livre de prefixo

```
1  Seja Q uma fila de prioridade, inicialmente vazia.;
2  para cada c em C faça
3      Q.insere(c);
4  enquanto |Q| > 1 faça
5      Criar novo Nó V;
6      V.esq = ExtraiMin(Q);
7      V.dir = ExtraiMin(Q);
8      V.freq = V.esq.freq + V.dir.freq;
9      Q.inserir(V);
10 devolva Árvore construída;
```


Complexidade

- Criar a fila de prioridade usando um Heap, pode ser feito em $O(n \log n)$ (na verdade dá para fazer até em $O(n)$.
- No laço da linha 3, cada iteração reduz o tamanho da fila em 1, então serão executadas $O(n)$ iterações.
- Dentro do laço, 2 extrações e uma inserção, que podem todas ser feitas em $O(\log n)$.
- Portanto a complexidade total do algoritmo é $O(n \log n)$

Corretude

- Exercício!
- Dicas:
  - Suponha por absurdo que uma árvore ótima T^* tem os nós mais profundos a e b, e que esses nós não são os com a menor frequência. Mostre que trocando a e b pelos nós com a menor frequência, você obtêm uma codificação melhor.
  - Aplique indução para mostrar que isso vale para os meta-nós também.

Conjunto Independente de Peso Máximo em Grafo Caminho

- Dado um grafo, um conjunto independente é um subconjunto S dos vértices tal que dois vértices adjacentes não podem pertencer a S.

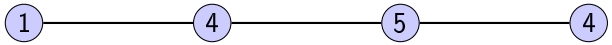
Conjunto Independente de Peso Máximo em Grafo Caminho

Dado um grafo caminho $G = (V, E)$, em que cada vértice $v \in V$ tem um peso não negativo w_v. Desejamos encontrar um subconjunto S de vértices não adjacentes (um conjunto independente - CI) de peso máximo.

Abordagens

- Uma tentativa força bruta: podemos testar todos os conjuntos independentes, lembrando daquele que tem o peso máximo.
- Porém existe $O(2^n)$ conjuntos independentes e portanto seria inviável para qualquer instância de tamanha razoável.
- Uma tentativa gulosa: Escolher o vértice de maior peso, que não seja adjacente a nenhum vértice já escolhido.
- Qual é a solução ótima e qual o resultado desse algoritmo nesse grafo?

a) 14 e 10
b) 8 e 6
c) 8 e 8
d) 9 e 8



Ideias - Subestrutura Ótima

- Vamos pensar sobre a estrutura de uma solução ótima.
- Em particular vamos tentar enxergar na solução ótima, soluções ótimas de subproblemas menores
- A ideia é que talvez a solução ótima possa ser obtida analisando um conjunto pequeno de subproblemas, e dessa forma uma busca direta nessas soluções é suficiente para encontrar a solução ótima.

- **Notação:** Seja $S \subseteq V$ ser um conjunto independente de peso máximo. Seja v_n o último vértice do caminho.
- Note que temos 2 opções aqui. Ou v_n está em S ou não está em S.
  - Caso 1: Suponha que $v_n \notin S$. Seja $G' = G$ com v_n removido.
  - Note que, S também é um conjunto independente em G' e também é o de peso máximo (suponha por contradição que não seja e você terá um CI mais pesado também para G, o que seria um absurdo)
  - Caso 2: Suponha que $v_n \in S$. Então v_{n-1} não pode mais estar na solução ótima. Seja $G'' = G$ com v_n e v_{n-1} removidos.
  - Note então que, $S - \{v_n\}$ é um conjunto independente em G'' e também é o de peso máximo (suponha por contradição que não seja e você terá um CI mais pesado também para G, o que seria um absurdo)

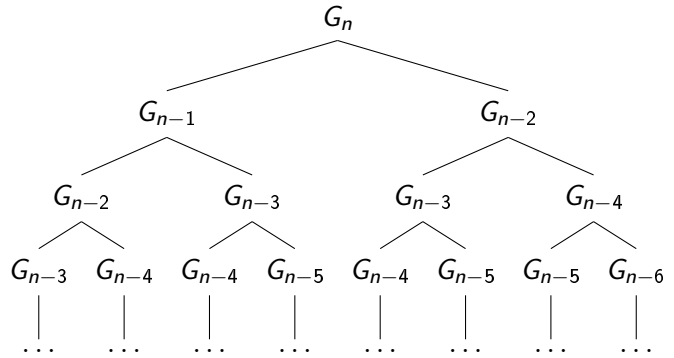
- Resumindo: Um CI de peso máximo em G deve ser
  1. um CI de peso máximo em G' ou
  2. v_n com um CI de peso máximo em G''
- Vamos tentar ambas em um algoritmo recursivo

Algoritmo 19: CIPM(G)

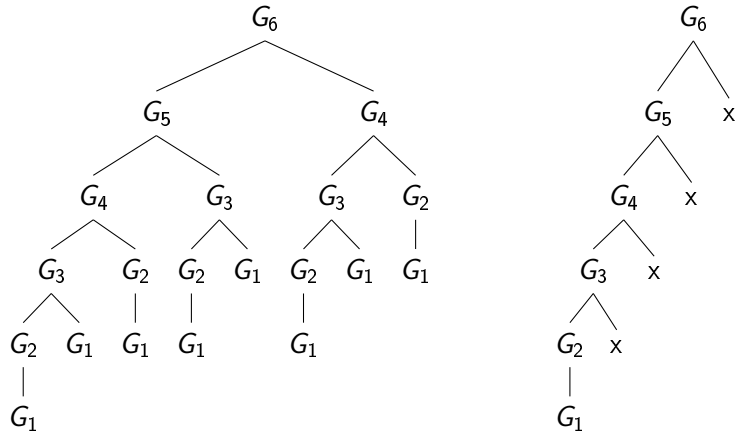
Entrada: Um grafo caminho G
Saída: Um conjunto independente de peso máximo de G

```
1 G' = G removendo o último vértice;
2 G" = G removendo os dois últimos vértices;
3 X = CIPM(G');
4 Y = CIPM(G") + ultimo vértice;
5 devolva o melhor entre X e Y;
```


- E qual o problema disso? Seja G_i o grafo só com os i primeiros vértices. Vejamos a árvore de recursão:



- Pode-se notar um grande número de repetições. Certo?
- E se memorizarmos numa tabela a primeira vez que encontrássemos um problema, e na próxima vez apenas consultamos ela? "Memoização"(Memoization)



Só temos n subproblemas diferentes!

- A memoização é conhecido como um método top-down, começa revolvendo do problema original e vai diminuindo o problema.
- Uma abordagem ainda mais interessante é a bottom-up, começando pelos problemas menores e usando a solução deles para construir os próximos, até chegar no problema original.
- Iremos manter um vetor $A[0 \ldots n]$ e preenche-lo da esquerda para a direita, sendo $A[i]$ o valor do CI de peso máximo de G_i.

Algoritmo 20: CIPM(G)

Entrada: Um grafo caminho G
Saída: O peso do conjunto independente de peso máximo de G

```
1 A[0] = 0; A[1] = w_1;
2 para i = 2, 3, ... n faça
3     A[i] = max{  A[i − 1]  ;  A[i − 2] + w_i  };
4 devolva A[n];
```

- Ainda precisamos reconstruir a solução (se precisarmos)
- Podemos fazer isso armazenando um vetor extra que vai contar qual decisão foi tomada em cada passo do algoritmo.
- Mas usualmente (e para economizar memória) o que fazemos é reconstruir a solução a partir do vetor que foi preenchido.
- O tempo de execução é $O(n)$ (também para reconstruir)
- A prova de corretude pode ser feita por indução.

Programação dinâmica

Princípios

Ingredientes para a Programação Dinâmica

- Identificar um conjunto pequeno de subproblemas
- Poder resolver subproblemas maiores usando a soluções dos subproblemas menores (já calculados). Usualmente escrevemos uma recorrência para demonstrar esse fato.
- Depois de resolver todos os subproblemas, poder computar (rapidamente) a solução final.

Problema da Mochila

Dado uma coleção I de n itens. Cada item $i \in I$ tem:

- Um valor v_i (não negativo)
- Um peso w_i (não negativo e inteiro)

Além disso também é dado uma capacidade W não negativa e inteira. Queremos encontrar um subconjunto $S \subseteq I$ cujo peso não ultrapasse W, ou seja,

$$\sum_{i \in S} w_i \leq W$$

e que maximiza

$$\sum_{i \in S} v_i$$

Esse problema aparece em vários contextos, basicamente qualquer problema em que temos uma quantidade limitada de recursos e queremos maximizar a eficiência.

Desenvolvendo um Algoritmo de Programação Dinâmica

- Vamos tentar pensar em como uma solução ótima pode ser formada usando a solução de subproblemas menores.
- O objetivo é encontrar recorrência que descreva esse fato.
- A ideia é partir de uma solução ótima e tentar disseca-la.
- Seja S uma solução para o problema da mochila com itens I e capacidade W.
- Assuma alguma ordenação dos itens
- Caso 1: o item n (último) não está em S. Logo S também é uma solução para o problema que considera só os $n-1$ primeiros itens.
- Caso 2: Suponha que $n \in S$. Então o que eu posso dizer sobre $S - \{n\}$?
  1. É solução para os $n-1$ primeiros itens e mochila com capacidade W
  2. É solução para os $n-1$ primeiros itens e mochila com capacidade $W - v_n$
  3. É solução para os $n-1$ primeiros itens e mochila com capacidade $W - w_n$
  4. Pode não ser viável

- **Notação:** Seja $V_{i,x}$ o valor da melhor solução que:
  1. só usa os i primeiros itens
  2. tem tamanho total $\leq x$
  3. Passo 1: encontra uma recorrência

Para $i \in I$ e para qualquer x,

$$V_{i,x} = \max \begin{cases} V_{(i-1),x} \\ v_i + V_{(i-1),(x-w_i)} \end{cases} \qquad \text{(somente se } w_i \leq x\text{)}$$

- Passo 2: Agora precisamos identificar os subproblemas.
- Variamos por todos os prefixos de itens $\{1, 2, \ldots, i\}$
- Variamos todas as capacidades residuais possíveis $x \in \{0, 1, 2, \ldots, W\}$
- Passo 3: Usar a recorrência para revolver todos os subproblemas (preencher tabela)
- Seja A um vetor bidimensional tal que $A[i, x]$ vai guardar o valor de $V_{i,x}$

Algoritmo 21: Knap(I, W)

Entrada: Um conjunto de itens I e uma capacidade W
Saída: O valor da solução ótima

```
1 A[0, x] = 0 para todo x;
2 para i = 1, 2, ..., n faça
3     para x = 0, 1, 2, ..., W faça
4         A[i, x] = max{A[i − 1, x]; A[i − 1, x − w_i] + v_i};
5 devolva A[n, W];
```

- O tempo de execução é $O(nW)$

Exemplo

- Um exemplo com 4 itens e Capacidade 6
- $v_1 = 3,\ w_1 = 4$
- $v_2 = 2,\ w_2 = 3$
- $v_3 = 4,\ w_3 = 2$
- $v_4 = 4,\ w_4 = 3$

Algoritmo 22: Knap(I, W)

1 $A[0, x] = 0$ para todo x;
2 **para** $i = 1, 2, \ldots, n$ **faça**
3 **para** $x = 0, 1, 2, \ldots, W$ **faça**
4 $A[i, x] =$
5 $\max \begin{cases} A[i-1, x], \\ A[i-1, x-w_i] + v_i \end{cases}$
6 devolva $A[n, W]$;

6					
5					
4					
3					
2					
1					
x = 0					
i =	0	1	2	3	4

6	0	3	3	7	8
5	0	3	3	6	8
4	0	3	3	4	4
3	0	0	2	4	4
2	0	0	0	4	4
1	0	0	0	0	0
x = 0	0	0	0	0	0
i =	0	1	2	3	4

Classes de Complexidade

- Formalmente as classes de complexidade P, NP, NP-completo e outras, são melhor definidas sobre os problemas de decisão, para os quais a resposta é simplesmente **SIM** ou **NÃO**. Mas os problemas tem uma forte relação entre si.
  - Caminho mínimo: Dado G, um vértice s e um número k existe um caminho de s para qualquer outro vértice com custo no máximo k?
  - MST: Dado G e um inteiro k, existe uma Árvore geradora mínima de custo no máximo k?
  - Compressão de Texto: Dado um texto T e um número k, existe uma compressão desse texto com no máximo k bits?

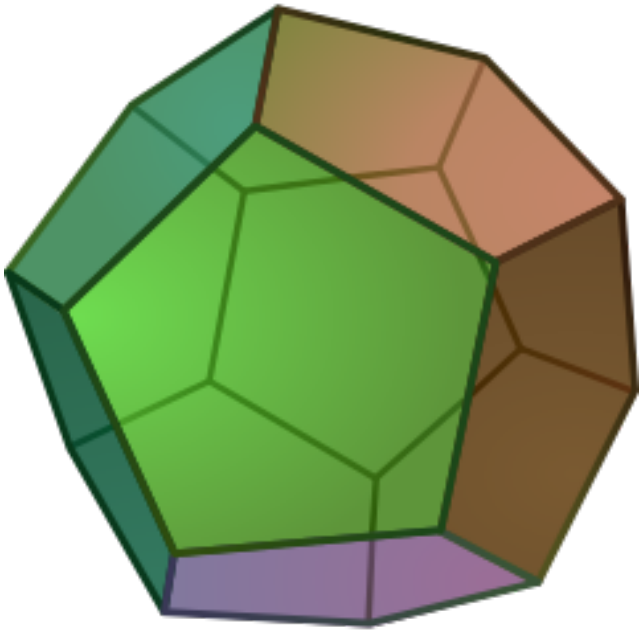
- O problemas que podem ser decididos em tempo polinomial formam a classe de complexidade

$$P$$

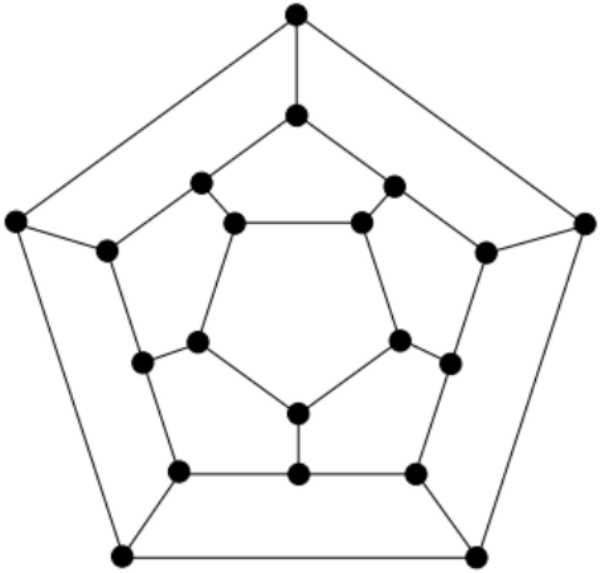
- Vimos vários problemas e algoritmos eficientes para resolve-los
  - Ordenação - $O(n \log n)$
  - Multiplicação - $O(n^{1.586})$
  - Multiplicação de Matrizes - $O(n^{2.8})$
  - Busca em Grafos - $O(m + n)$
  - Encontrar Componentes fortemente conexas - $O(m + n)$
  - Caminhos Mínimos - $O(m \log n)$
  - Escalonamento Ponderado - $O(n \log n)$
  - Compressão de Texto - $O(n \log n)$
  - MST - $O(m \log n)$

- Mas será que todos os problemas podem ser resolvidos em tempo polinomial?
- Quando falamos de Caminhos entre todos os pares de vértices, em grafos com ciclos de pesos negativos, se proibíssemos os ciclos o problema era bem definido, mas não sabíamos como resolver de maneira eficiente.
- De fato o problema da mochila que resolvemos em $O(nW)$ também não foi resolvido em tempo polinomial no tamanho da entrada. Uma vez que para escrever W precisamos $m = \log W$ bits. Portanto o tempo de execução é $O(n2^m)$, exponencial no tamanho da entrada!

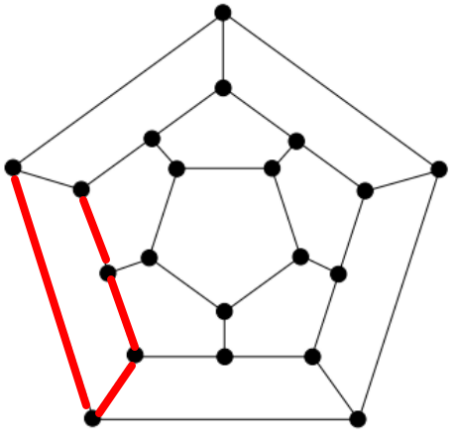
Sir William Rowan Hamilton (1805-1865)



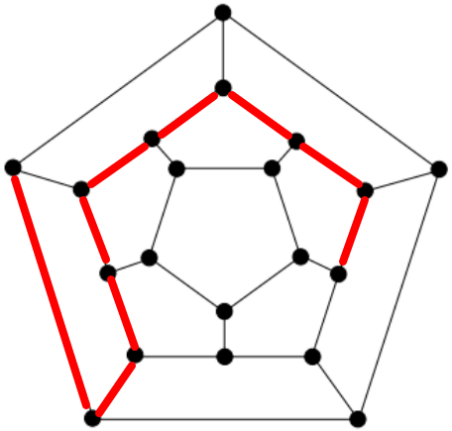
257 / 460



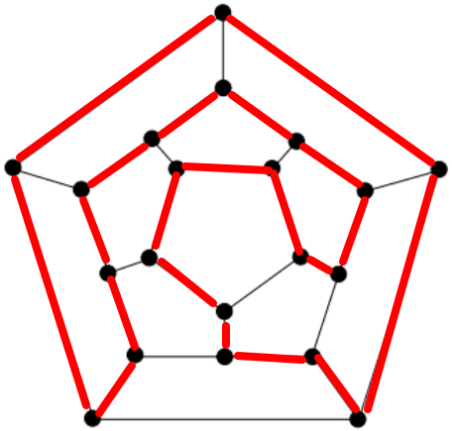
258 / 460

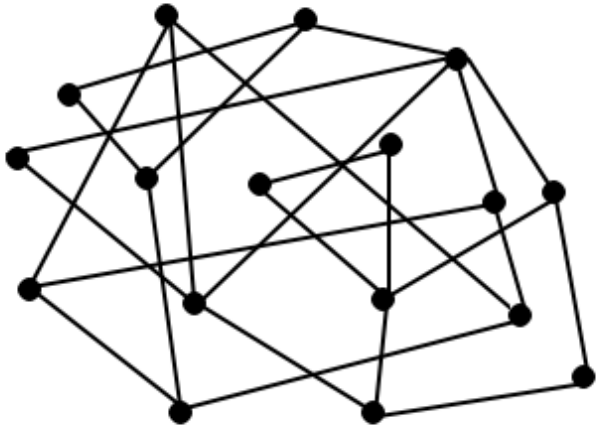


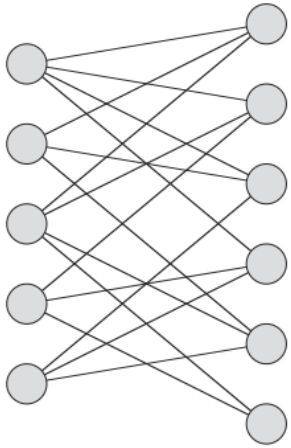
259 / 460



260 / 460







- Um **ciclo hamiltoniano** de um grafo G é um ciclo que passa por todos os vértices exatamente uma vez.

Problema do Ciclo Hamiltoniano

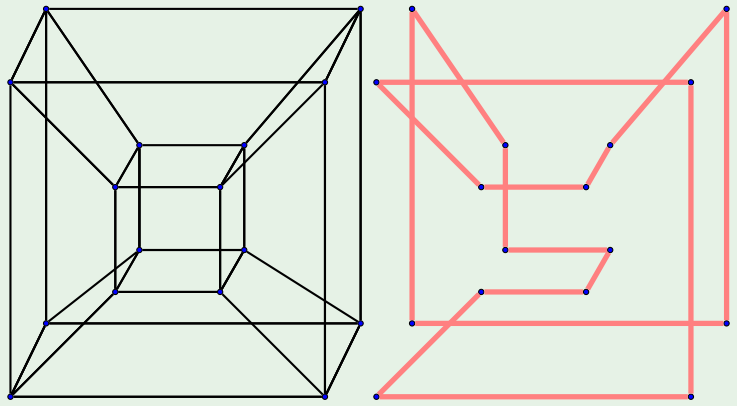
Dado um grafo não orientado $G = (V, E)$. Decidir se G possui um ciclo hamiltoniano.

- Não se conhece nenhum algoritmo de tempo polinomial para resolve-lo!

A classe *NP*

- **Definição:** Um problema A está em NP se para qualquer instância x de A para o qual a resposta seja "sim", existe um certificado y de tamanho polinomial, que pode ser verificado em tempo polinomial, provando que a resposta de x de fato é "sim".

Ciclo Hamiltoniano



- Dado um grafo G, existe uma árvore geradora de custo $\leq W$
  - Um certificado, é a própria árvore
- Dado um texto T, existe uma compressão que usa $\leq B$ bits
  - Um certificado é o próprio texto comprimido
- Todo problema em P está em NP
- Dado um conjunto de itens I e uma mochila de capacidade W, decidir se cabe na mochila um subconjunto de itens de valor $\geq K$.
  - Um certificado é a própria coleção de itens.

- Todo problema em NP pode ser resolvido por força-bruta em tempo exponencial.
- Gerar um número exponencial de soluções e verificar cada uma em tempo polinomial.
- A maioria dos problemas (computáveis) que encontramos está em NP

- **Definição:** Dada uma classe de problemas C um problema é C-**difícil** se ele é tão difícil quanto qualquer outro problema em C. Ou seja, existe uma redução de tempo polinomial de qualquer problema em C para ele.
- Um problema é C-Completo se é C-difícil e pertence a C

Reduções

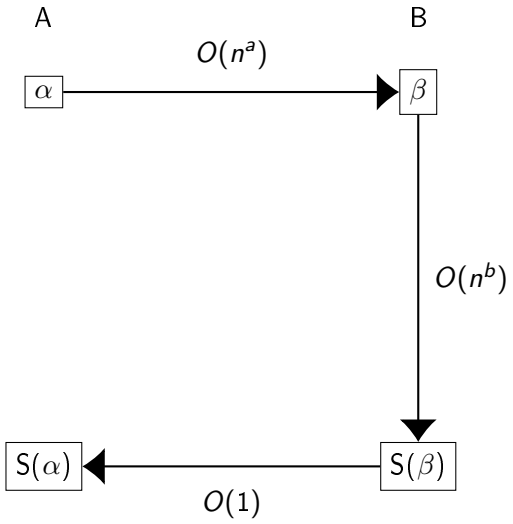
Suponha que temos um problema de decisão A que queremos resolver em tempo polinomial.

Agora suponha que temos outro problema de decisão B que já sabemos que pode ser resolvido em tempo polinomial.

Finalmente, suponha que conhecemos um procedimento que transforma uma instância α do problema A, em uma instância β no problema B. Tal que:

- A transformação leva tempo polinomial
- A resposta de α para A é a mesma da instância β para B.

Reduções

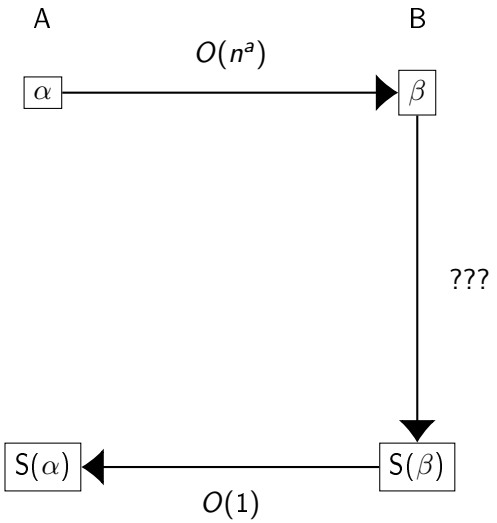


Reduções

- Temos então um algoritmo polinomial para resolver o problema A.

- Agora suponha que temos um problema A que sabemos que é difícil de resolver.
- e suponha que conseguimos uma redução de A para B que seja muito rápida.
- O que podemos afirmar sobre B?

Problema muito difícil



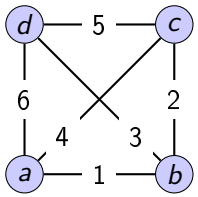
A classe NP-Completo

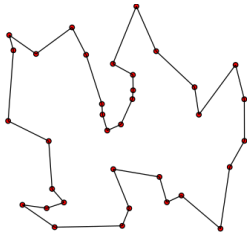
- Será que existem problemas *NP*-Completos?
- E se existirem será que existe um algoritmo polinomial para resolve-los?
- A maioria dos Ciências da Computação acredita que tal algoritmo não existe.

O Problema do Caixeiro Viajante

Travelling Salesman Problem - TSP

- Dado um grafo com custos nas arestas, encontrar um ciclo que passa por todos os vértices exatamente uma vez de custo mínimo.





- Versão de decisão: Dado um grafo com custos nas arestas e um valor W, decidir se existe um ciclo que passa por todos os vértices exatamente uma vez de custo $\leq W$.

História

- Jack Edmonds é um Matemático e Cientista da Computação Estadunidense
- Graduado em 1957 na George Washington University e Pós-graduado na University of Maryland em 1959.

História

- Trabalhou no National Institute of Standards and Technology de 1959 até 1969
- Em 1965 em um artigo chamado "Paths, Trees and Flowers" conjecturou que não existe um algoritmo em tempo polinomial para o TSP.

História

- Stephen Arthur Cook é um cientista da computação e matemático Estadunidense-Canadense
  - Obteve em 1962 e 1966 seu mestrado e doutorado pela Universidade de Harvard
  - Em 1970 mostrou que existem Problemas *NP*-completos

História

- Leonid Levin é um matemático e cientista da computação Soviético
  - Obteve seu mestrado e doutorado em 1970 e 1972 na universidade de Moscou
  - Em 1972 mostrou a existência dos problemas *NP*-completos.
- O teorema de Cook-Levin afirma que o problema da satisfabilidade boleana é NP-Completo.

História

- Richard Karp é um cientista da computação Estadunidense.
- Obteve seu mestrado e doutorado em matemática aplicada em 1956 e 1959 em Harvard.

História

- Trabalhou na IBM e foi professor de ciência da computação e Pesquisa Operacional na Universidade da Califórnia, Berkeley.
- Mostrou 21 problemas que eram NP-completos. E a partir desses milhares foram provados.‘

Problemas *NP*-Completos

- Suponha então que você se deparou com um problema π, e tentou todas as técnicas que você aprendeu, mas não obteve um algoritmo polinomial.
- Mais uma ferramenta essencial para a nossa caixa: talvez seu problema seja *NP*-Completo. Receita para provar:
  1. Provar que ele pertence a *NP*
  2. Provar que ele é *NP*-difícil,
     - ⋆ Escolha um problema π' que sabidamente é *NP*-completo
     - ⋆ Faça uma redução (polinomial) de π' para π.

Problemas *NP*-Completos

- Qual problema π' escolher?
- 21 problemas de Karp
- Cap. 34 do CLRS
- Garey e Johnson, Computers and Intractability, 1979

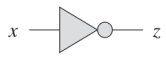
Um primeiro problema *NP*-completo

Satisfabilidade de Circuito

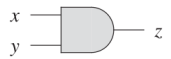
- O teorema de Cook-Levin é um pouco complicado (para mim, pelo menos).
- Ao invés disso vamos argumentar (um pouco informalmente) que de fato existe um problema *NP*-completo.
- O nosso primeiro problema será o problema da satisfabilidade de circuitos.

Satisfabilidade de Circuito

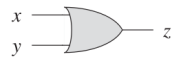
- Um circuito é formado por portas lógicas ligadas por fios.
- Em um extremo temos os fios de entrada e no outro temos os fios de saída.
- Existem três portas básicas. a porta NOT, AND e OR



x	$\neg x$
0	1
1	0



x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

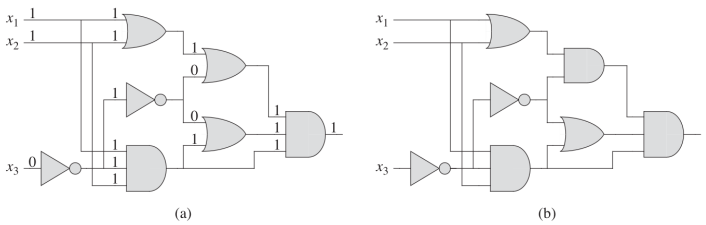


x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Satisfabilidade de Circuito

Problema da Satisfabilidade de Circuito

Dado um circuito C com fios de entrada e uma saída. Existe alguma atribuição dos valores de entrada que tornam a saída verdadeira?



- O primeiro circuito tem uma atribuição que a torna verdadeira.
- Enquanto o segundo é inviável.

Satisfabilidade de Circuito

Teorema

O Problema da Satisfabilidade de Circuitos é NP-completo.

- Primeiro provar que o Problema da Satisfabilidade de Circuitos é *NP*
- Depois provar que o Problema da Satisfabilidade de Circuitos é *NP*-Difícil.

Lema

O Problema da Satisfabilidade de Circuitos $\in$ NP.

Prova: Dado um circuito C e uma atribuição dos valores de cada fio de C. Um algoritmo A pode verificar cada porta lógica e verificar se os fios de entrada estão correspondendo corretamente ao fio de saída daquela porta. E verifica se o fio de saída do circuito é 1. Se C é viável ele tem um certificado, se C não é viável nenhum certificado pode enganar A. O algoritmo A roda em tempo polinomial. Então verificamos que o Problema da Satisfabilidade de Circuitos $\in$ *NP*.

Satisfabilidade de Circuito

Lema

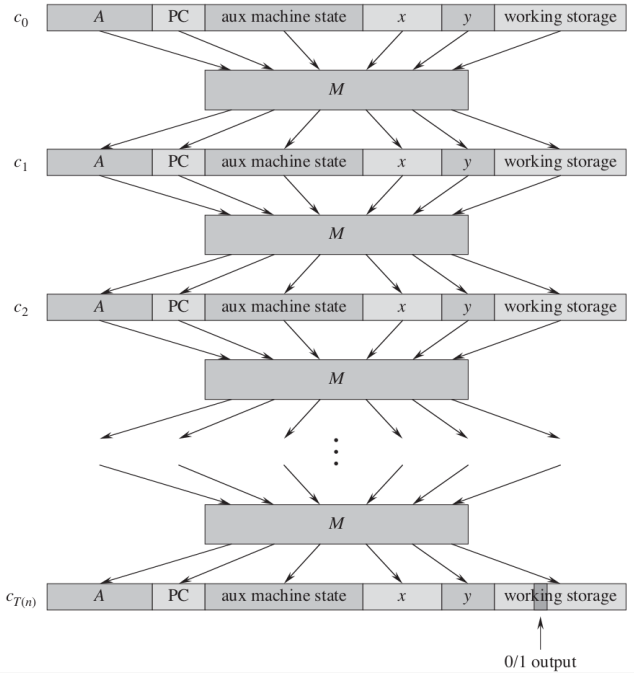
O Problema da Satisfabilidade de Circuitos é NP-Difícil.

- Primeiramente vamos relembrar de como funciona um computador.
- Um programa de computador é uma sequencia de instruções guardadas na memória do computador.
- Uma instrução tipicamente é composta da operação a ser executada, do endereço da memória dos operadores e do endereço onde vai ser armazenado o resultado.

- Um pedaço especial da memória é o **contador de programa** que sabe qual é a próxima instrução a ser executada.
- Sempre que uma instrução é executada, o contador é incrementado ou sobrescrito (no caso de um desvio)
- A qualquer momento, a memoria do computador (RAM, contadores, registradores, etc) guarda uma **configuração** completa de um passo na execução do programa.

Satisfabilidade de Circuito

- Agora considere um problema *NP* qualquer.
- Por definição existe um algoritmo A que recebe uma instância x e um certificado y e devolve SIM (1) se aquele certificado comprova que a solução de x é SIM.
- Então vamos simular a execução desse algoritmo!



- Mas o computador M nada mais é do que um circuito lógico.
- O número de réplicas é polinomial no tamanho de x
- Então se removermos o y, e deixar entradas livres. Temos um circuito que só é satisfeito com a entrada adequada.

3-CNF-SAT

- Uma formula booleana é composta
  - de variáveis booleanas $x_1, x_2, \ldots, x_n$,
  - de conectivos $\neg$ (NOT), $\vee$ (OR), $\wedge$ (AND),
  - Parenteses.
- Uma variável ou sua negação são chamadas **literais**.
- Uma fórmula booleana está na forma normal 3-conjuntiva se
  - está dividido em clausuras
  - cada clausura tem 3 literais conectas por ORs
  - as clausuras estão conectadas por ANDs.

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

3-CNF-SAT

3-CNF-SAT

Dado uma fórmula booleana na forma normal 3-conjuntiva, decidir se existe alguma atribuição das variáveis que torna a formula verdadeira.

Teorema

O 3-CNF-SAT é NP-Completo

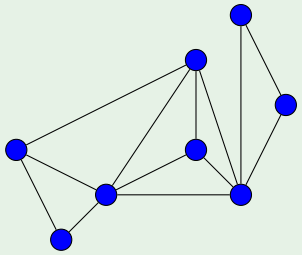
Prova: Veremos depois.

Problema do CLICK

Problema do CLICK

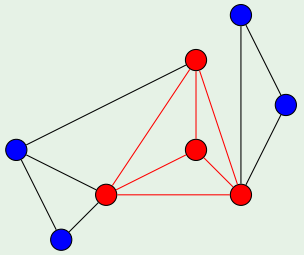
Dado um grafo não orientado $G = (V, E)$, e um inteiro k decidir se existe um subgrafo G' induzido de G que tenha k vértices e seja completo.

Exemplo



Existe uma click de tamanho 4?

Exemplo



Existe uma click de tamanho 4?

O Problema do CLICK

Será que CLICK é NP-Completo?

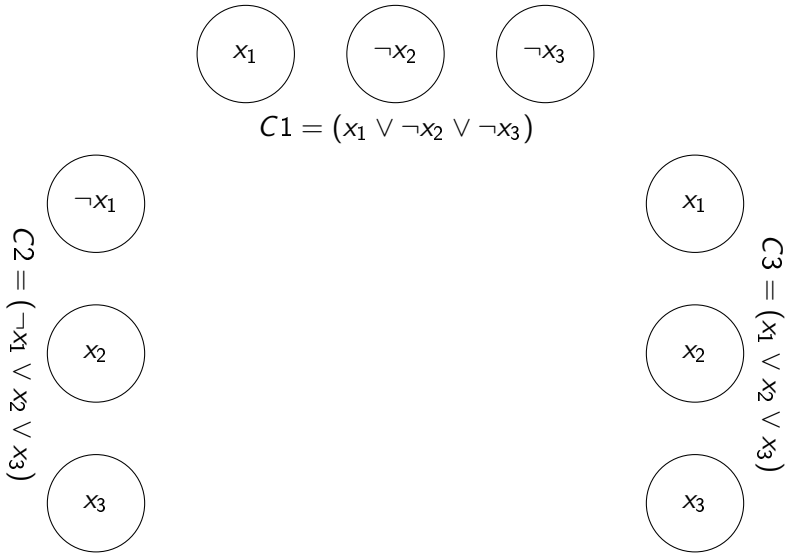
- CLICK $\in$ NP, basta apresentar o conjunto de vértices
- Vamos reduzir o 3-CNF-SAT para CLICK
  - Devemos converter qualquer instancia do 3-CNF-SAT para o problema do CLICK
  - Essa redução deve ter tempo polinomial
  - Aqui vamos mostrar o algoritmo da redução em um exemplo, mas é necessário garantir que funciona para qualquer instância.

3-CNF-SAT $\leq_p$ CLICK

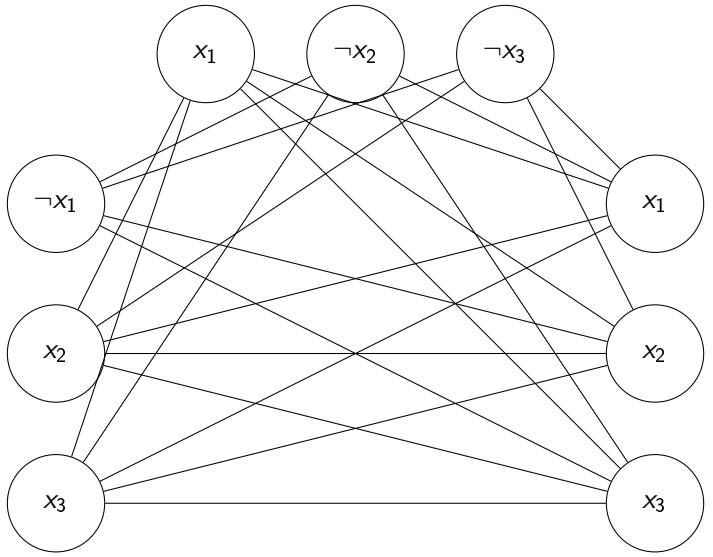
Considere uma formula booleana com k clausulas na forma normal 3-conjuntiva.

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

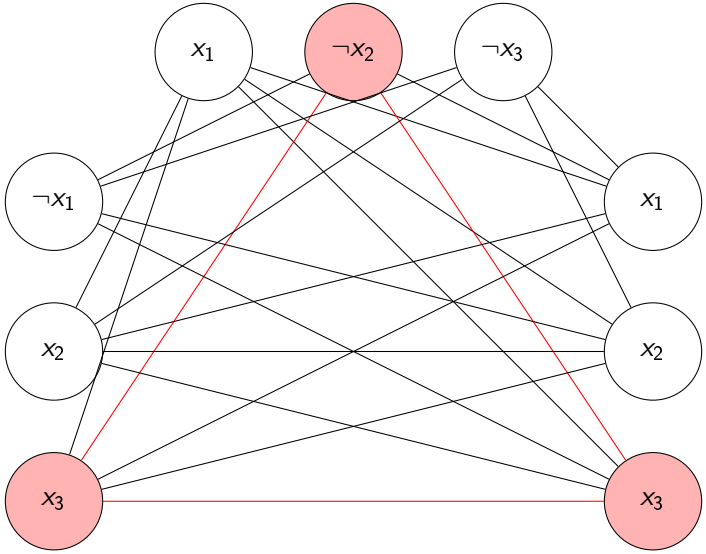
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



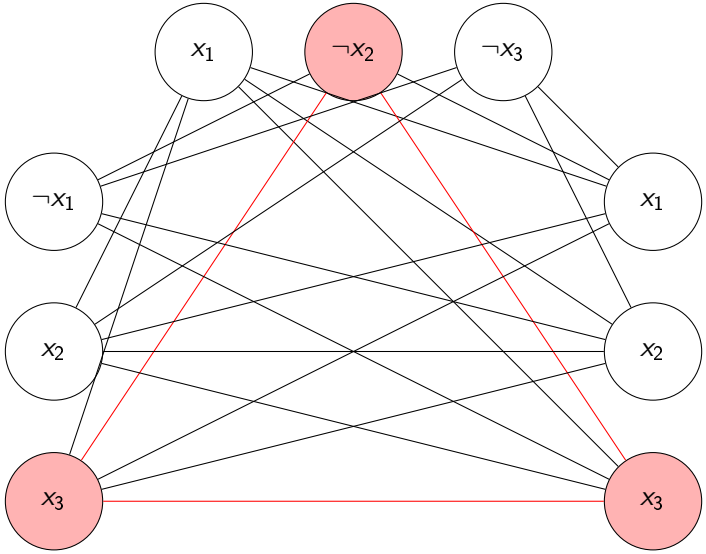
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



A questão P vs NP

- Seria $P = NP$?
- Bastaria mostrar 1 algoritmo com tempo de execução polinomial para 1 problema NP-completo.
- Conjectura-se (fortemente) que $P \neq NP$.
- Mas também não foi provado.

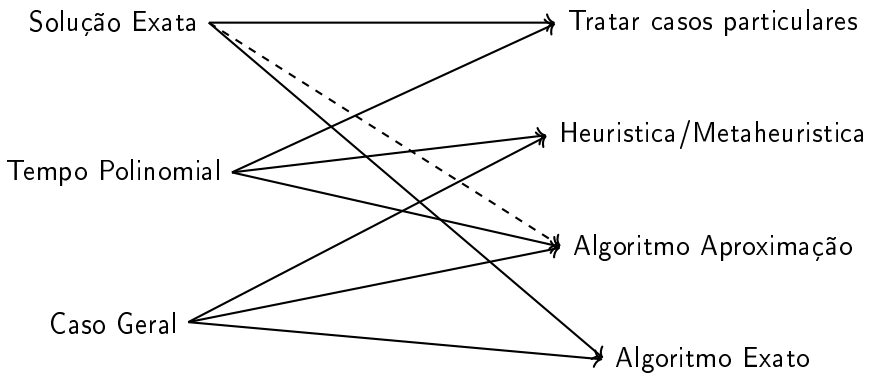
O nome NP

- O nome *NP* deriva de Non-deterministic Polynomial Time
- São problemas que podem ser resolvidos em tempo polinomial em uma máquina de Turing não determinística.
- Informalmente, considere uma máquina que conseguisse testar todas as soluções simultaneamente.

Problemas Intratáveis

- O que podemos fazer se um problema é *NP*-completo? (sem ser desistir)
- Na verdade muita coisa pode ser feita, e é disso que trataremos nessa disciplina.

Se $P \neq NP$ não conseguiremos para um problema NP-Difícil:



Casos Especiais

- Muitas vezes um problema geral pode ser *NP*-completo mas alguns casos especiais podem ser mais fáceis de resolver
  1. Encontrar um conjunto independente máximo é *NP*-completo. Mas no grafo caminho é fácil.
  2. O problema da Mochila binária é *NP*-completo, mas o da mochila fracionária é fácil.
  3. Na mochila binária, se a capacidade for polinomial no número de itens o problema também é fácil
  4. Satisfabilidade de fórmulas booleanas é *NP*-completo, mas o 2-CNF-SAT é fácil

Heurísticas

- Podemos abrir mão de encontrar a solução ótima, e tentar encontrar uma solução boa o bastante.
- Normalmente heurísticas são rápidas.
- Geralmente não possuem garantias do quão próximas estão da solução ótima

Algoritmos Aproximados

- Podemos abrir mão de encontrar a solução ótima, e tentar encontrar uma solução boa o bastante.
- Executam em tempo polinomial.
- Possuem garantias do quão próximas estão da solução ótima.
- Por exemplo: A solução de um algoritmo aproximado vai estar no máximo a um fator α da solução ótima.

Algoritmos Exatos

- Busca a solução ótima.
- Desiste do tempo polinomial.
- A ideia é reduzir ao máximo a complexidade, e aumentar ao máximo o tamanho das instâncias que conseguimos resolver.
  - Branch-and-Bound
  - Programação Linear Inteira
  - Programação por restrições

Satisfazibilidade de Fórmulas

- Uma fórmula booleana ϕ é composta de:
  - n variáveis booleanas: $x_1, x_2, \ldots, x_n$;
  - m conectivos booleanos: $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$
  - parênteses
- Ex. $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

Problema da Satisfazibilidade de Fórmulas - SAT

Dada uma fórmula booleana, decidir se existe uma atribuição das variáveis booleanas que faz com que ela seja avaliada como 1 (verdadeira).

- No exemplo ϕ é satisfazível com a atribuição $< x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 >$
- Um algoritmo ingênuo que testa todas as possibilidades é inviável pois existem 2^n atribuições diferentes.

Teorema

SAT é NP-completo

Lema

$SAT \in NP$

Lema

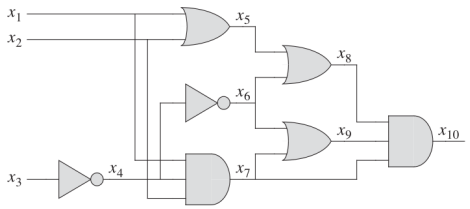
$SAT \in$ *NP-Difícil*

Lema

$SAT \in NP$

- Para mostrar que SAT $\in$ NP, mostramos que um certificado consiste em uma atribuição satisfatória das variáveis. Esse certificado pode ser verificado substituindo cada variável pelo valor dessa atribuição e avaliando a expressão, que pode ser feito em tempo polinomial. Se o resultado for 1 o algoritmo verificou que a formula é satisfazível.

- Para mostrar que SAT $\in$ NP-Difícil mostraremos que CIRCUIT-SAT $\leq_p$ SAT.
- Considere um circuito C qualquer. Para cada fio x_i no circuito, a fórmula ϕ vai ter uma variável x_i, expressamos cada porta lógica como uma cláusula na formula booleana que representa o seu comportamento. No fim fazemos a conjunção das cláusulas.



$$\begin{aligned}\phi \;=\; & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \\ & \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4) \\ & \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) \,.\end{aligned}$$

Lema

C é satisfazível se e somente se ϕ é satisfazível.

- Esse algoritmo de redução executa em tempo polinomial.
- ($\rightarrow$) Se C tem uma atribuição que satisfaz, cada fio tem um valor bem definido e a saída do circuito é 1.
- Portanto se atribuirmos os valores de cada fio para as respectivas variáveis, a formula também terá valor 1.
- ($\leftarrow$) Se uma atribuição faz ϕ se verdadeira. Podemos atribuir o valor de cada fio com o valor das suas respectivas variáveis. E C terá saída igual a 1.
- Portanto mostramos que CIRCUIT-SAT $\leq_p$ SAT. $\square$

Satisfazibilidade 3-CNF

- Uma formula está na forma normal conjuntiva (*conjunctive normal form* - CNF) se é expressa como uma conjunção (ANDs) de clausulas, e cada clausula como disjunções (ORs) de uma ou mais literais.
- Uma fórmula booleana está na forma normal 3-conjuntiva (3-CNF) se está na forma normal conjuntiva e cada cláusula tem exatamente três literais distintas.
- Por exemplo:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

3-CNF-SAT

Dada uma fórmula booleana na forma normal 3-conjuntiva, decidir se existe uma atribuição das variáveis booleanas que faz com que ela seja avaliada como 1 (verdadeira).

Teorema

3-CNF-SAT é NP-completo

Lema

3-CNF-SAT $\in$ NP

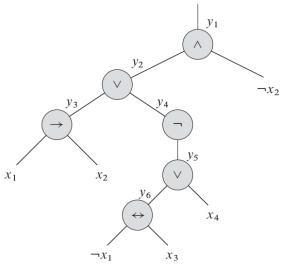
Prova igual à prova que SAT $\in$ NP.

Lema

3-CNF-SAT ∈ NP-Difícil

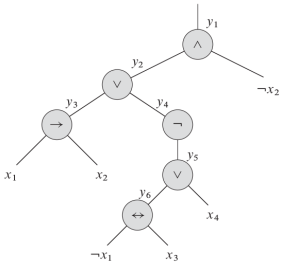
- Para mostrar que 3-CNF-SAT ∈ NP-Difícil mostraremos que SAT $\leq_p$ 3-CNF-SAT.
- Considere ϕ uma formula booleana qualquer.
- Primeiramente construímos uma árvore de análise para ϕ em que cada literal é uma folha e os conectivos são nós internos. Essa construção sempre é possível (usando a associatividade podemos colocar parenteses para explicitar uma ordenação)

- Por exemplo $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.



- Depois dessa construção, introduzimos uma variável para cada aresta que sobe dos nós internos.
- A seguir escrevemos cláusulas para cada nó interno. E fazemos a conjunção dessas cláusulas e criamos uma formula ϕ'.

- Por exemplo $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.



- No exemplo ao lado a formula obtida seria:

$$\begin{aligned}\phi' \;=\; & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge\ (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge\ (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge\ (y_4 \leftrightarrow \neg y_5) \\ & \wedge\ (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge\ (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\ .\end{aligned}$$

- Dessa forma obtemos clausulas que tem no máximo 3 literais. Mas ainda não está na forma normal 3-conjuntiva. Então podemos escrever a tabela verdade de cada clausula, e encontrar os valores que tornam ela FALSA. Por exemplo a cláusula $\phi'_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2))$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

- Analisando as linhas da tabela que tornam a cláusula FALSA obtemos:

$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

- Agora aplicamos a lei de DeMorgan

$\phi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$

- Caso a cláusula resultante só tenha 2 literais $(l_1 \vee l_2)$, então incluímos uma literal auxiliar e a seguinte formula $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$
- No caso de uma cláusula com uma única literal l, incluímos 2 literais auxiliares $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$.
- Seja ϕ''' a fórmula em 3-CNF-SAT resultante.

Lema

ϕ''' é satisfazível se e somente se ϕ é satisfazível.

- Como todas as transformações preservam o valor algébrico da formula, tanto ϕ quanto ϕ''' são equivalentes.
- A redução é calculada em tempo polinomial. Construir ϕ' a partir de ϕ insere no máximo uma variável e uma cláusula por conectivo.
- Construir ϕ'' a partir de ϕ' introduz no máximo oito cláusulas para cada cláusula. E a construção de ϕ''' no máximo multiplica por 4 o número de cláusulas.
- Portanto mostramos uma redução de tempo polinomial de SAT para 3-CNF-SAT e concluímos a prova. □

CLICK

Problema do CLICK

Dado um grafo não orientado $G = (V, E)$, e um inteiro k decidir se existe um subgrafo G' induzido de G que tenha k vértices e seja completo.

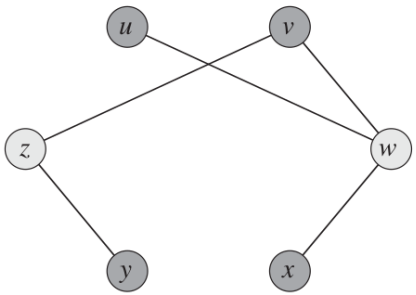
Teorema

CLICK é NP-completo

- Provado com 3-CNF-SAT $\leq_p$ CLICK

Cobertura por Vértices

- Dado um grafo não direcionado $G = (V, E)$ uma cobertura por vértices de G é um subconjunto $V' \subseteq V$ tal que para toda aresta $(u, v) \in E$, pelo menos um entre u e v deve estar em V'.
- Dizemos que um vértice em V' cobre todas as arestas adjacentes a ele. E em uma cobertura por vértices todas as arestas devem ser cobertas.
- O **tamanho** de uma cobertura por vértices é o número de vértices que contêm.



Problema da Cobertura por Vértices - VERTEX-COVER

Dado um grafo não orientado $G = (V, E)$, e um inteiro k decidir se existe uma cobertura por vértices $V' \subseteq V$ de tamanho k.

Teorema

VERTEX-COVER é NP-completo

Lema

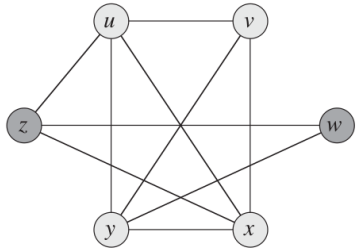
VERTEX-COVER $\in$ NP

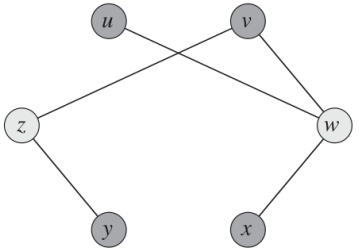
- Utilizando a própria cobertura V' como certificado, podemos verificar facilmente, em tempo polinomial, se $|V'| = k$ e se toda aresta (u, v), u ou v está em V'.

Lema

VERTEX-COVER $\in$ *NP-Difícil*

- Vamos mostrar que CLICK $\leq_p$ VERTEX-COVER.
- Definição: O **Complemento** de um grafo $G = (V, E)$ denotado por $\overline{G} = (V, \overline{E})$, em que, $\overline{E} = \{(u, v) : u, v \in V, u \neq v \text{ e } (u, v) \notin E\}$. Ou seja, o complemento de G é um grafo com os mesmos vértices e exatamente as arestas que não estão em G.
- A redução consiste em dada uma entrada $< G, k >$ do problema da CLICK. Calcular o complemento $\overline{G}$, o que pode ser feito em tempo polinomial.
- E então usar $\overline{G}$ como entrada para o problema do VERTEX-COVER, procurando uma cobertura de tamanho $|V| - k$.





Lema

G tem uma CLICK de tamanho k, se e somente se, $\overline{G}$ *tem uma cobertura por vértices de tamanho* $|V| - k$

- ($\rightarrow$) Se G tem uma CLICK C com k vértices, então $V \setminus C$ é uma cobertura em $\overline{G}$.
- Seja (u, v) qualquer aresta em $\overline{E}$, então (u, v) não está em G e portanto u e v não podem estar em C simultaneamente.
- Logo, pelo menos um deles está em $V \setminus C$ e portanto (u, v) está coberta.
- ($\leftarrow$) Se $\overline{G}$ tem uma cobertura por vértices $V' \subseteq V$ em que $|V'| = |V| - k$, então para todo $u, v \in V$ se $(u, v) \in \overline{E}$ então ou $u \in V'$ ou $v \in V'$ ou ambos. Pela contrapositiva se nenhum dos dois está em V' significa que (u, v) está em E e portanto $V - V'$ é uma CLICK, e tem tamanho $|V| - |V'| = k$
- Portanto mostramos um redução CLICK $\leq_p$ VERTEX-COVER. $\square$

O Problema do Ciclo Hamiltoniano

- Definição: Um **ciclo hamiltoniano** de um grafo é um circuito que passa exatamente uma vez por todos os vértices.

Problema do Ciclo Hamiltoniano - HAM-CYCLE

Dado um grafo não orientado $G = (V, E)$, decidir se G tem um ciclo hamiltoniano.

Teorema

HAM-CYCLE é NP-completo

Lema

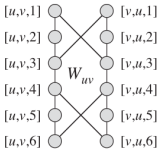
HAM-CYCLE $\in$ *NP*

Prova: Exercício

Lema

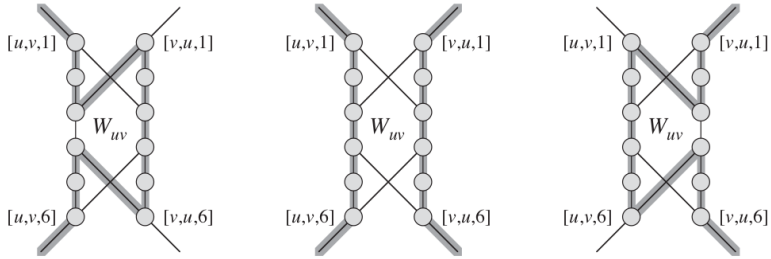
HAM-CYCLE ∈ NP-Difícil

- Mostraremos que VERTEX-COVER $\leq_p$ HAM-CYCLE.
- Dado um grafo não dirigido $G = (V, E)$ e um inteiro k, construiremos um grafo não dirigido $G' = (V', E')$ que tem um ciclo hamiltoniano, se e somente se, G tem uma cobertura por vértices de tamanho k.
- A construção de G' se baseia em uma **engenhoca** (*widget*) que é um pedaço de um grafo que impõe certas propriedades.



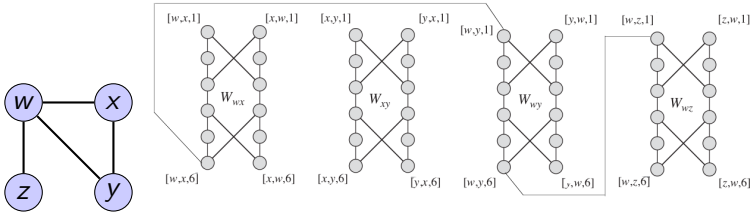
- Para cada aresta $(u, v) \in E$ o nosso grafo G' conterá uma cópia da engenhoca. Que denotaremos por W_{uv}.
- Cada vértice W_{uv} tem um nome e ao total ele tem 14 arestas.
- Para a engenhoca funcionar como queremos, ela vai se conectar ao resto do grafo apenas pelos vértices $[u, v, 1], [u, v, 6], [v, u, 1]$ e $[v, u, 6]$

- Só existem três formas de um caminho entrar na engenhoca, passar por todos os vértices e sair.



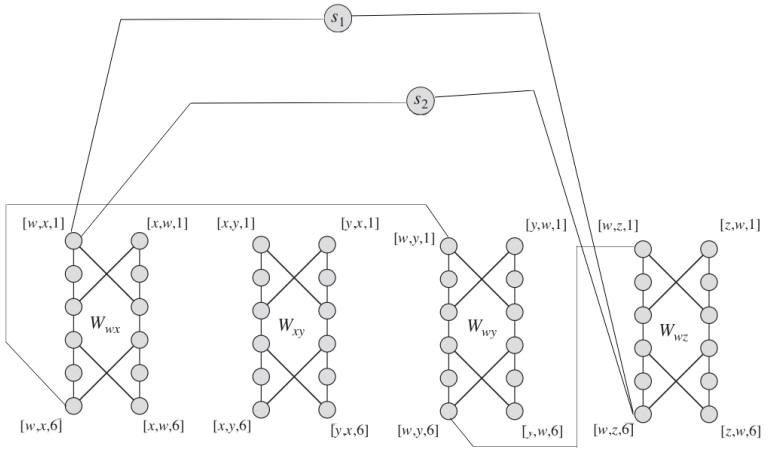
- Em particular é impossível construir dois caminhos disjuntos nos vértices, um que ligue $[u, v, 1]$ a $[v, u, 6]$ e outro que ligue $[v, u, 1]$ a $[u, v, 6]$.
- Além das engenhocas serão adicionados k vértices seletores $s_1, s_2, \ldots, s_k$. Usaremos as arestas que incidem nesses vértices para selecionar os k vértices que formarão a cobertura por vértices em G.

- Posteriormente para cada vértice $u \in V$ adicionamos arestas que criam um caminho em G' que passam por todas as engenhocas que correspondem a arestas incidentes a u.
- Ordenamos arbitrariamente os vértices adjacentes a u. E dada a ordenação $(v_1, \ldots, v_{grau(u)})$, conectamos $[u, v_i, 6]$ com $[u, v_{i+1}, 1]$.
- No exemplo a seguir, w é vizinho de x, y e z (considerando essa ordem).

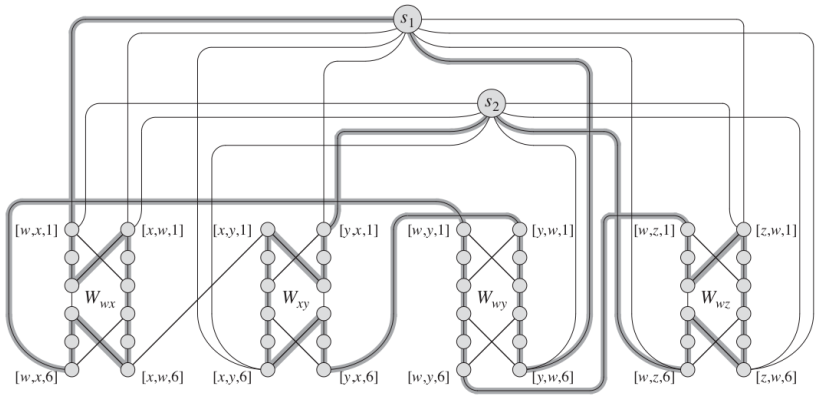


- A intuição é que se escolhermos um vértice u para a nossa cobertura, podemos fazer um caminho que passa por todas as engenhocas que correspondem as arestas adjacentes a u.

- O último tipo de aresta em E' une os vértices $[u, v_1, 1]$ e $[u, v_{grau(u)}, 6]$ a cada um dos seletores.



- Fazendo esse mesmo procedimento para todos os vértices obtemos o seguinte grafo, que é grande mas ainda é polinomial.



- Com 12 vértices por engenhoca, mais $k \leq |V|$ vértices seletores, em um total de

$$\begin{aligned} |V'| &= 12|E| + k \\ &\leq 12|E| + |V| \end{aligned}$$

- Para cada vértice $u \in V$ temos $grau(u) - 1$ arestas entre as engenhocas, em um total de

$$\sum_{u \in V} (grau(u) - 1) = 2|E| - |V|$$

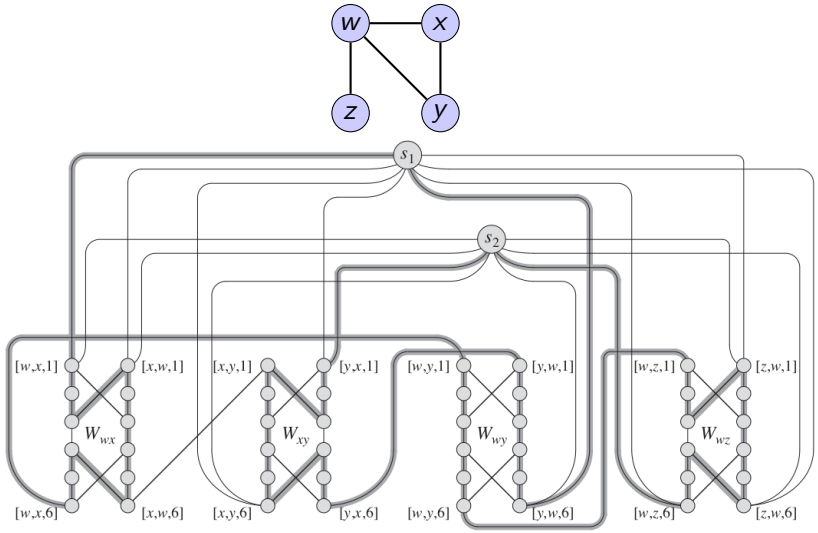
- Cada engenhoca tem 14 arestas, além das arestas entre os vértices seletores, no total

$$\begin{aligned} |E'| &= (14|E|) + (2|E| - |V|) + (2k|V|) \\ &= 16|E| + (2k - 1)|V| \\ &\leq 16|E| + (2|V| - 1)|V| \end{aligned}$$

Lema

G tem uma cobertura por vértices de tamanho k, se e somente se, G' tem um caminho hamiltoniano.

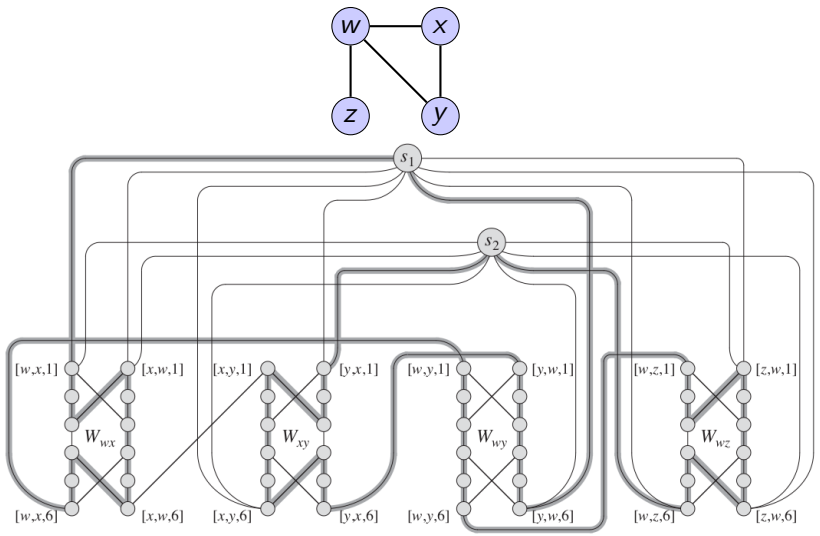
- ($\rightarrow$) Suponha que $G = (V, E)$ tem uma cobertura por vértices $V' \subseteq V$ de tamanho k.
- Para cada vértice $u \in V*$ com os vértices $(v_1, \ldots, v_k)$, adicionamos no caminho as arestas que ligam a primeira engenhocas que representa v_i ao seletor s_i.
- Depois ligamos a saída da primeira engenhoca, com a próxima do vértice v_i.
- Por fim ligamos a ultima engenhoca ao próximo seletor $s_{i+1 mod k}$.
- Além disso ligamos os vértices internos da engenhoca dependendo se a aresta é coberta por um ou por dois vértices.
- No exemplo se $k = 2$, temos uma cobertura formada por w e y.



- Como a cobertura por vértices incide em todas as aresta, todas as engenhocas serão visitadas (uma ou duas vezes), assim como os vértices seletores. E portanto formamos um ciclo hamiltoniano.
- ($\leftarrow$) Suponha que $G' = (V', E')$ tem um ciclo hamiltoniano $C \subseteq E'$. Afirmamos que o conjunto

$$V' = \{u \in V : (s_j, [u, v, 1]) \in C \text{ para algum } 1 \leq j \leq k\}$$

- é uma cobertura por vértices para G.
- Como o caminho que sai de um seletor passa pelas engenhocas até chegar em algum seletor (já que é um ciclo hamiltoniano).
- Pela forma como G' foi construído, os vértices internos de cada engenhoca W_{uv} só podem ser visitados se o caminho teve inicio em u ou v, e portanto este estará na cobertura por vértices.



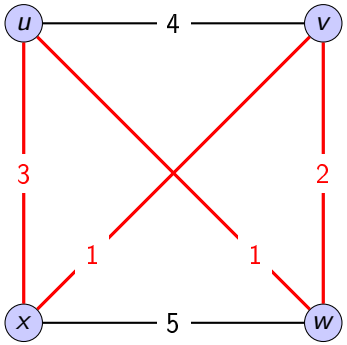
Resumindo:

- Mostramos que HAM-CYCLE $\in$ NP
- Mostramos que HAM-CYCLE $\in$ NP-Difícil
  - Mostrando uma redução de qualquer instância do VERTEX-COVER para HAM-CYCLE
  - Essa redução é de tempo polinomial
  - Essa redução é correta, ou seja a instância do VERTEX-COVER decide sim se e somente se a instância do HAM-CYCLE decide sim.
- Portanto demonstramos que HAM-CYCLE $\in$ NP-Completo. $\square$

Caixeiro Viajante

Problema do Caixeiro Viajante - TSP

Dado um grafo completo não direcionado $G = (V, E)$, custos inteiros $c(i, j)$ para $i, j \in V$ e um inteiro k decidir se existe um ciclo que passa exatamente uma vez por cada vértice (hamiltoniano) com custo menor ou igual a k.



Existe uma solução com custo menor ou igual a 7? Sim.

Teorema

TSP é NP-completo

Lema

TSP $\in$ *NP*

Lema

TSP $\in$ *NP-Difícil*

Lema

TSP $\in$ *NP*

- Dado uma instância do problema, podemos usar como certificado a sequência de n vértices do percurso.
- O algoritmo de verificação confirma se a sequência contém cada vértice uma vez, e se a soma dos custos das arestas é menor ou igual a k.
- Esse algoritmo pode ser feito em tempo polinomial.

Lema

TSP $\in$ *NP-Difícil*

- Mostraremos que HAM-CYCLE $\leq_p$ TSP.
- Seja $G = (V, E)$ uma instância do HAM-CYCLE.
- Construímos uma instância do TSP da seguinte maneira: Formamos o grafo completo $G' = (V, E')$, e custos:

$$c(i,j) = \begin{cases} 0 & \text{se } (i,j) \in E, \\ 1 & \text{se } (i,j) \notin E. \end{cases}$$

- Essa instância pode ser criada em tempo polinomial.

- Agora mostramos que G tem um ciclo hamiltoniano, se e somente se, G' tem um percurso cujo custo é zero.
- ($\rightarrow$) Suponha que G tenha um ciclo hamiltoniano h. Cada aresta $h \in E$ e portanto tem custo 0 em G'. Dessa forma h também é um percurso em G' e tem custo 0.
- ($\leftarrow$) Suponha que G' tenha um percurso h' de custo menor ou igual a 0. Como só exitem arestas de custo 0 ou 1, o custo de h' é exatamente 0. Portanto as arestas de h' exitem em G e formam um ciclo hamiltoniano.

Problema da Soma de Subconjuntos

Problema da Soma de Subconjuntos - SUBSET-SUM

Dado um conjunto finito S de inteiros positivos e um inteiro $t > 0$. Decidir se existe um subconjunto $S' \subseteq S$ cuja a soma de seus elementos é t.

- Por exemplo: seja $S = \{1, 2, 7, 14, 49, 98, 343\}$ e $t = 108$.
- O conjunto $S' = \{1, 2, 7, 98\}$ é uma solução.

Teorema

SUBSET-SUM é NP-completo

Lema

SUBSET-SUM $\in$ *NP*

- Exercício.

Lema

SUBSET-SUM $\in$ *NP-Difícil*

Lema

SUBSET-SUM $\in$ *NP-Difícil*

- Mostraremos que 3-CNF-SAT $\leq_p$ SUBSET-SUM.
- Considere uma fórmula f para as variáveis $x_1, x_2, \ldots, x_n$ com cláusulas $C_1, C_2, \ldots, C_k$.
- Construiremos uma instância $< S, t >$ para o SUBSET-SUM da seguinte forma:
- Dois números para cada variável x_i (v_i e v'_i),
- e dois números para cada cláusula C_j (s_j e s'_j)
- cada número terá $n + k$ dígitos.

- Rotulamos os n primeiros dígitos para cada uma das variáveis,
- e os k últimos dígitos para cada uma das cláusulas.

- O alvo t tem os n primeiros dígitos iguais a 1 e o restante iguais a 4.
- v_i e v_i tem 1 nos dígitos rotulados por x_i.
- v_i tem 1 nos dígitos rotulados pelas cláusulas em que x_i aparece (não negado).
- v_i' tem 1 nos dígitos rotulados pelas cláusulas em que $\neg x_i$ aparece.
- v_i e v_i' tem 0 nos demais dígitos.
- s_j tem 1, e s_j' tem 2 nos dígitos rotulados por C_j e 0 em todos os outros.

$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge$
$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge$
$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge$
$(x_1 \vee x_2 \vee x_3)$

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
$v_1 =$	1	0	0	1	0	0	1
$v_1' =$	1	0	0	0	1	1	0
$v_2 =$	0	1	0	0	0	0	1
$v_2' =$	0	1	0	1	1	1	0
$v_3 =$	0	0	1	0	0	1	1
$v_3' =$	0	0	1	1	1	0	0
$s_1 =$	0	0	0	1	0	0	0
$s_1' =$	0	0	0	2	0	0	0
$s_2 =$	0	0	0	0	1	0	0
$s_2' =$	0	0	0	0	2	0	0
$s_3 =$	0	0	0	0	0	1	0
$s_3' =$	0	0	0	0	0	2	0
$s_4 =$	0	0	0	0	0	0	1
$s_4' =$	0	0	0	0	0	0	2
$t =$	1	1	1	4	4	4	4

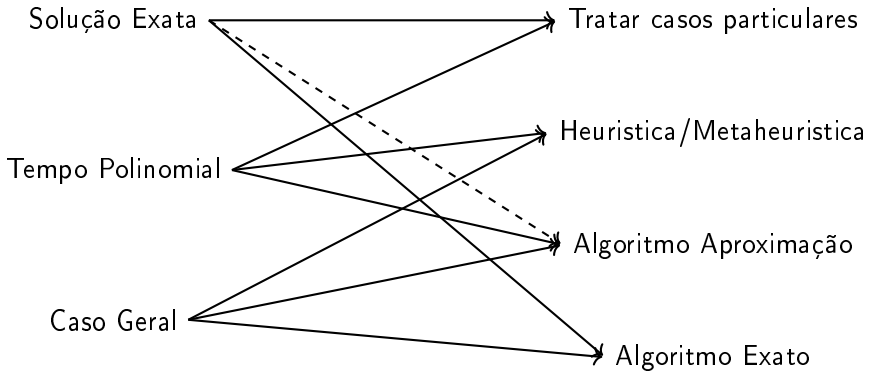
- Falta mostrar que f é satisfazível, se e somente se, a instância $< S, t >$ do SUBSET-SUM tem uma solução.
- ($\rightarrow$) Suponha que f tem uma atribuição que a torna verdadeira. Para $i = 1, 2, \ldots, n$ se $x_i = 1$ incluímos v_i em S', se $x_i = 0$ incluímos v_i' em S'.
- Como cada cláusula é satisfeita, a soma em cada dígito rotulado como C_j é no mínimo 1 e no máximo 3, dessa forma basta colocar em S' os valores s_j e s_j' que completam 4 naquele dígito.
- ($\leftarrow$) Exercício.

NP-Difícil

- Quando nos deparamos com um problema NP-Completo ou NP-Difícil é provável que não consigamos encontrar uma solução exata em tempo polinomial.

O que fazer então?

Se $P \neq NP$ não conseguiremos para um problema NP-Difícil:

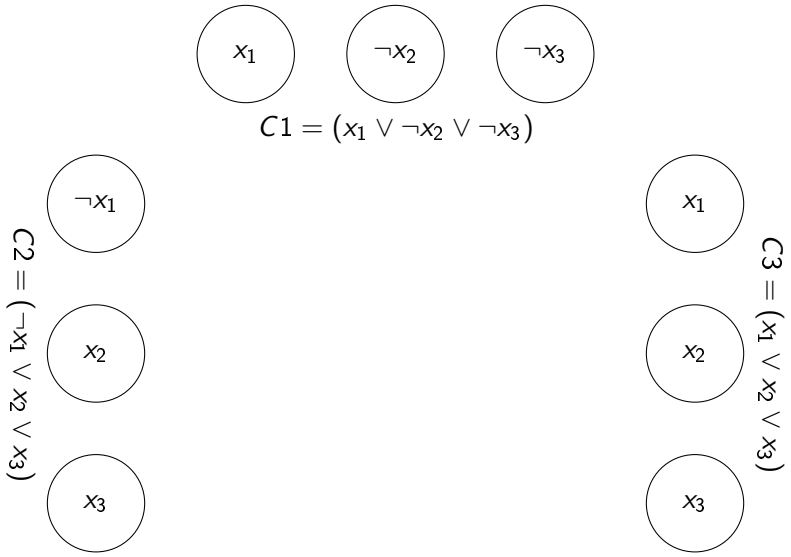


3-CNF-SAT $\leq_p$ CLICK

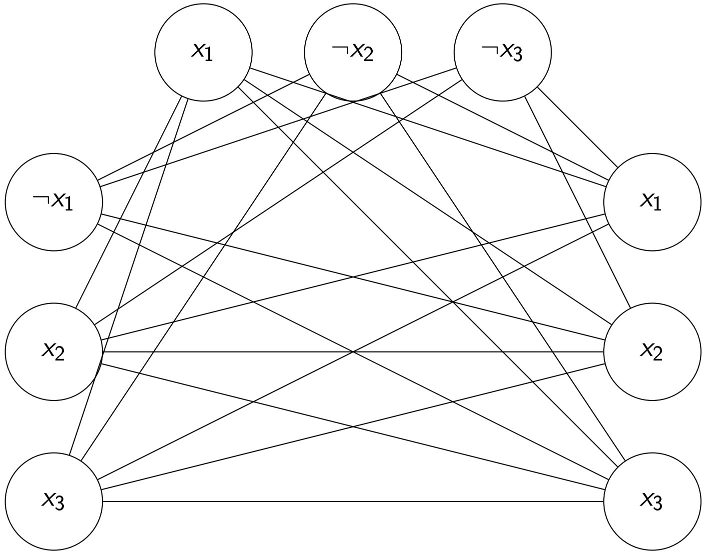
Considere uma formula booleana com k clausulas na forma normal 3-conjuntiva.

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

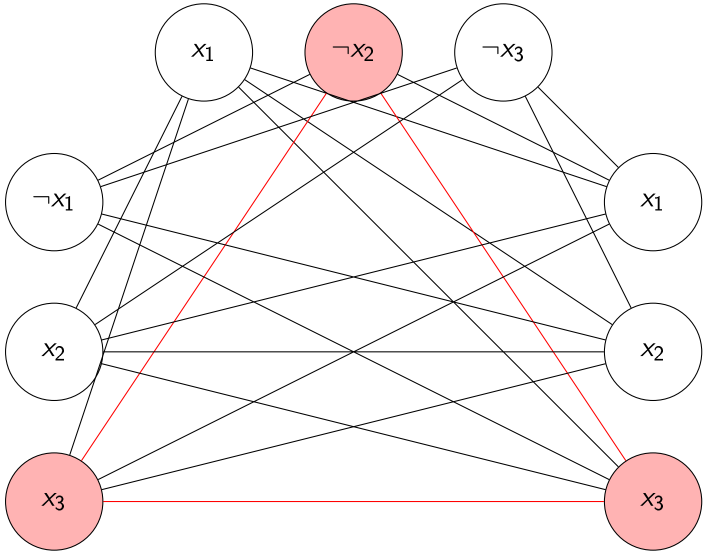
$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



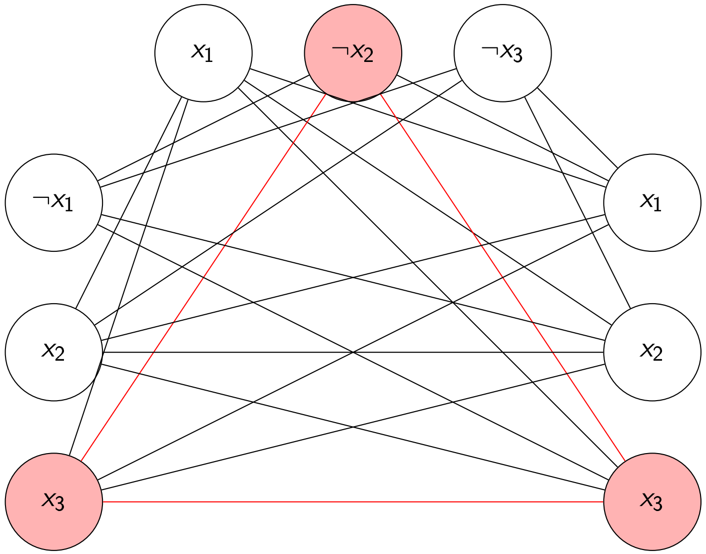
$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



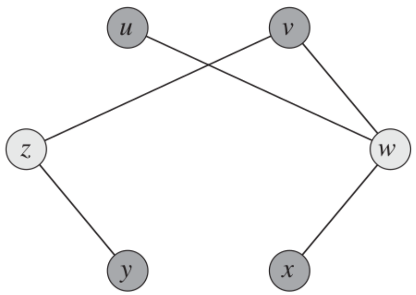
$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



Cobertura por Vértices

VERTEX-COVER

Dado um grafo não orientado $G = (V, E)$ com n vértice e m arestas, e um inteiro k decidir se existe uma cobertura por vértices $V' \subseteq V$ de tamanho k.



Qual é o tamanho de uma cobertura por vértices de tamanho mínimo em um grafo estrela com n vértices e em um grafo clique de tamanho n.

a) 1 e $n-1$
b) 1 e n
c) 2 e $n-1$
d) $n-1$ e n

- Uma solução força bruta:
- Considerando que k seja pequeno em relação a n podemos tentar todos os conjuntos com k vértices.
- São $\binom{n}{k}$ conjuntos.
- Se $k <<< n$ então a complexidade do algoritmo é $\approx \Theta(n^k)$.
- Podemos fazer melhor?

Teorema

Dada uma aresta (u, v),
$G_u = G$ deletando u e todas as suas arestas adjacentes, e
$G_v = G$ deletando v e todas as suas arestas adjacentes.

G tem uma cobertura de tamanho k, se e somente se
G_u ou G_v tem uma cobertura de tamanho $k-1$.

- ($\rightarrow$) Suponha que G tem uma cobertura V' de tamanho k. Como a aresta (u, v) precisa estar coberta, pelo menos um dos seus extremos tem que estar em V'.
- Sem perda de generalidade, suponha que $u \in V'$.
- O vértice u cobre apenas as arestas adjacentes a u, e portanto todas as demais arestas devem estar cobertas pelos $k-1$ vértices restantes de V', e portanto
- $V \backslash \{u\}$ é uma cobertura para G_u e tem $k-1$ vértices
- ($\leftarrow$) Exercício.

Algoritmo 23: BuscaCobertura

Entrada: Um grafo $G(V, E)$ e um inteiro k
Saída: Verdadeiro se G contêm uma cobertura de tamanho k

```
1 se |E| > 0 e k == 0 então devolve Falso
2 se |E| == 0 então devolve Verdadeiro
3 Escolhe uma aresta qualquer (u, v) ∈ E
4 Cria G_u = G sem u e suas arestas adjacentes
5 Cria G_v = G sem v e suas arestas adjacentes
6 se BuscaCobertura(G_u, k − 1) então devolve Verdadeiro
7 se BuscaCobertura(G_v, k − 1) então devolve Verdadeiro
8 devolve Falso
```

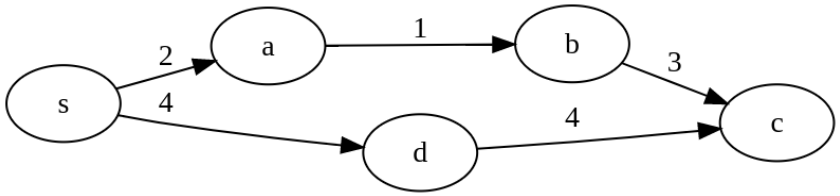

Complexidade de BuscaCobertura:

- A cada chamada recursiva, fazemos outras 2.
- A profundidade da recursão é no máximo k.
- Portanto o número de chamadas recursivas é no máximo 2^n.
- Cada chamada recursiva leva tempo $O(m)$ para criar G_u e G_v.
- Portanto a complexidade total é $O(2^n m)$, portanto exponencial. (Claro que é).
- Ainda muito melhor que o $\Theta(n^k)$ da força bruta.

Caminhos mais Curtos de Única Fonte

Problema do Caminho Mínimo de Única fonte

Dado um grafo direcionado $G = (V, A)$ com m arcos e n vértices, em que cada arco $a \in A$ tem um custo c_a, e um vértice fonte $s \in V$. Desejamos calcular para cada vértice $v \in V$ o valor $D(v)$ do $s - v$ caminho mais curto.



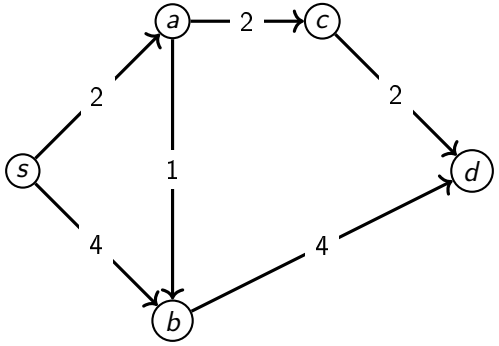
- O Algoritmo de Dijkstra executa em tempo $O(m \log n)$.
- O Dijkstra só funciona corretamente se os comprimentos forem **positivos**, ou seja, $c_a \geq 0, \forall a \in A$.
- Quando os custos podem ser negativos precisamos de outro algoritmo.

Encontrando uma subestrutura ótima

- Note que como não existem ciclos negativos, qualquer caminho mais curto entre a fonte e qualquer outro vértice passa por no máximo $n-1$ arcos.

- Qual o caminho mínimo de $s - v$ com no máximo k arcos?
- Seja $A[i, v]$ o custo do caminho mínimo se s até v com no máximo k arcos:
- $A[0, s] = 0$, $A[0, v] = +\infty$ se $v \neq s$:

$$A[i, v] = \min \begin{cases} A[i-1, v] \\ \min\limits_{w:(w,v)\in A} \{A[i-1, w] + c_{wv}\} \end{cases}$$



Algoritmo 24: Bellman-Ford(G, s)

Entrada: Um conjunto Grafo, e um vértice fonte s
Saída: O valor do menor caminho de s para todos os outros vértices

1 $A[0, s] = 0$; $A[0, v] = +\infty$ para todo $v \neq s$;
2 **para** $i = 1, 2, \ldots, n-1$ **faça**
3 **para** *todo* $v \in V$ **faça**
4 $A[i, v] = \min\{A[i-1, v]; \min\limits_{w:(w,v)\in A} \{A[i-1, w] + c_{wv}\}$;
5 devolva $A[n-1, *]$;

Algoritmo Bellman-Ford

- O tempo de execução do Bellman-Ford é $O(mn)$
- Você pode parar o algoritmo se em uma iteração a distância para nenhum vértice mudar.
- Podemos detectar ciclos negativos rodando uma iteração a mais $i = n$, se encontrarmos algum caminho mais curto significa que existe um ciclo de custo negativo.

Algoritmo Exato para o TSP

- Dado um grafo não-direcionado $G = (V, E)$, encontrar o custo mínimo de ciclo que visita exatamente uma vez todos os vértice de V.
- Uma solução força bruta:
- Testar todos os $n!$ percursos possíveis.
- Resolve 12, 13, talvez 14 vértices.
- Podemos fazer melhor? Vamos tentar fazer um algoritmo de Programação Dinâmica!

Tentativa 1

- Vamos entender o ciclo como um caminho que começa no vértice 1 vai até algum vértice j e depois viaja direto de j para 1, fechando o ciclo.
- Podemos usar a ideia do algoritmo de Bellman-Ford, para encontrar o caminho de 1 até j que usa no máximo i arestas.
- Para todo $i \in \{0, 1, \ldots, n\}$ e todo destino $j \in \{1, 2, ..., n\}$, temos L_{ij} o custo de um caminho mínimo de 1 até j que usa no máximo i arestas.

- Podemos então pegar todos os $L_{n-1,j}$ e somar com o custo de c_{j1}.
- Isso vai formar um ciclo de caixeiro viajante? Infelizmente não!
- O Bellman-Ford apenas indica o máximo de arestas que podem ser usadas.

Tentativa 2

- Podemos modificar um pouco a ideia do algoritmo de Bellman-Ford, para encontrar o caminho de 1 até j que usa **EXATAMENTE** i arestas.
- Para todo $i \in \{0, 1, \ldots, n\}$ e todo destino $j \in \{1, 2, ..., n\}$, temos L_{ij} o custo de um caminho mínimo de 1 até j que usa **exatamente** i arestas.

- Novamente, podemos então pegar todos os $L_{n-1,j}$ e somar com o custo de c_{j1}. Certo?
- Infelizmente também não. Note que o caminho mínimo de $n-2$ arestas até um vértice k pode passar por j,
- Dessa forma estaríamos repetindo vértices (e deixando alguns de fora).

Tentativa 3

- Podemos tentar modificar mais um pouco a ideia para encontrar o caminho de 1 até j que usa no **exatamente** i arestas e **NÃO REPETE** vértices.
- Para todo $i \in \{0, 1, \ldots, n\}$ e todo destino $j \in \{1, 2, ..., n\}$, temos L_{ij} o custo de um caminho mínimo de 1 até j que usa **exatamente** i arestas e **não repete** vértices.

- Note que ainda o caminho mínimo de $n-2$ arestas até um vértice k pode passar por j.
- Como garantir que não vai haver repetições de vértices?
- O único jeito é resolver mais subproblemas, para cada possível conjunto de vértices.

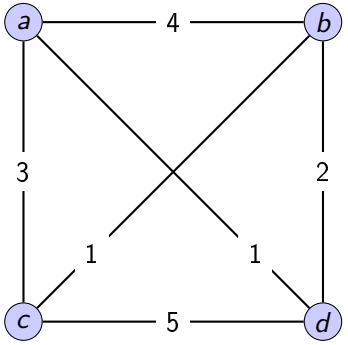
Subestrutura Ótima

- Para todo destino $j \in \{1, 2, ..., n\}$ e todo subconjunto $S \subseteq V$ que contêm 1 e j.
- L_{Sj} é o custo de um caminho mínimo de 1 até j que visita todos os vértices de S (e somente eles).

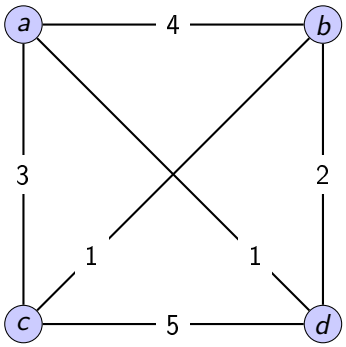
- Considere o seguinte exemplo:
- Queremos descobrir $L_{\{1,3,7,8,9\},7}$ ou seja queremos o caminho de 1 até 7 que visita exatamente uma vez os vértices 1, 3, 7, 8 e 9. Quais opções temos?
- Podemos ir de 1 até 3 visitando $\{1, 3, 8, 9\}$ depois para 7.
- Podemos ir de 1 até 8 visitando $\{1, 3, 8, 9\}$ depois para 7.
- Podemos ir de 1 até 9 visitando $\{1, 3, 8, 9\}$ depois para 7.

Dessa forma podemos escrever a seguinte recorrência:

$$L_{S,j} = \begin{cases} c_{1j}, \text{ se } S = \{1, j\} \\ \min_{k \in S, k \neq j} \{L_{S \setminus \{j\}, k} + c_{kj}\} \end{cases}$$

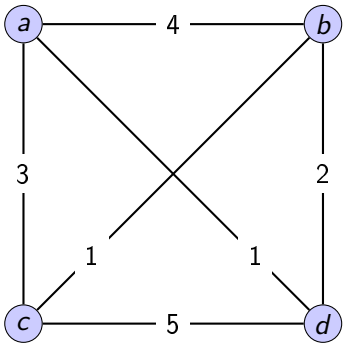


S	b	c	d
$\{a, b\}$	4	-	-
$\{a, c\}$	-	3	-
$\{a, d\}$	-	-	1



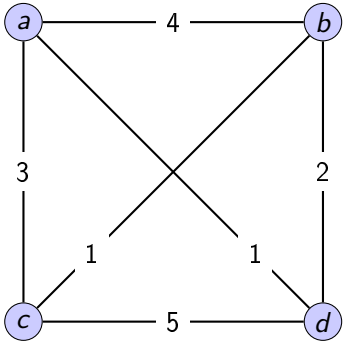
S	b	c	d
$\{a,b\}$	4	-	-
$\{a,c\}$	-	3	-
$\{a,d\}$	-	-	1

S	b	c	d
$\{a,b,c\}$	4	5	-
$\{a,b,d\}$	3	-	6
$\{a,c,d\}$	-	6	8



S	b	c	d
$\{a,b,c\}$	4	5	-
$\{a,b,d\}$	3	-	6
$\{a,c,d\}$	-	6	8

S	b	c	d
$\{a,b,c,d\}$	7	4	6



S	b	c	d
$\{a,b,c,d\}$	7	4	6

Por fim a ultima aresta,
$(b,a) = 7+4 = 11$
$(c,a) = 4+3 = 7$
$(d,a) = 6+1 = 7$

Algoritmo 25: BellmanHeldKarp

Entrada: $G(V,E)$, $V = \{1,2,...,n\}$, custos c_{ij} para $(i,j) \in E$
Saída: Custo mínimo de um percurso de caixeiro viajante em G

```
1 //A[S][j] indica o custo mínimo de chegar em j passando uma vez por cada vértice de S ⊆ V
2 A[2^{n-1} − 1][n − 1];
3 para j = 2 até n faça
4     A[{1, j}][j] = c_{1j};
5 para s = 3 até n faça
6     para todo S com |S| = s e 1 ∈ S faça
7         para j ∈ S\{1} faça
8             A[S][j] = min_{k∈S\{1,j}}(A[S\{j}][k] + c_{kj});
9 devolve min_{j∈S\{1}}(A[V][j] + c_{j1});
```

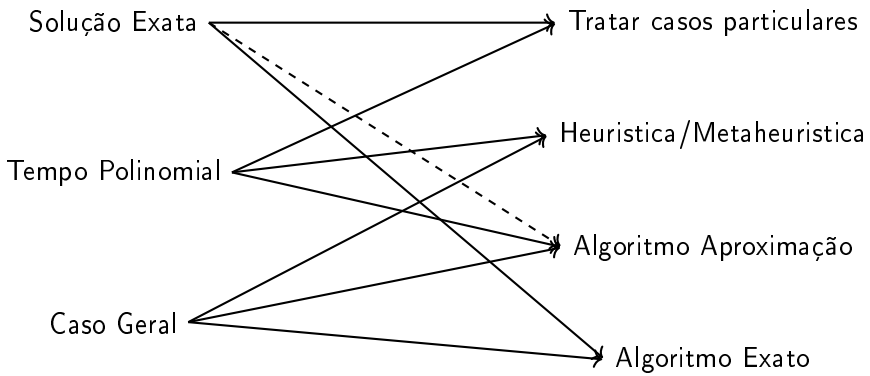
Complexidade.

- Temos $O(2^n)$ possíveis escolhas de S.
- Para cada S temos $O(n)$ problemas (um para cada membro de S)
- O total de subproblemas é $O(n2^n)$
- Para cada subproblema temos que procurar o mínimo em $O(n)$ subproblemas.
- Dessa forma a complexidade total de BellmanHeldKarp é $O(n^2 2^n)$

Considere que vamos utilizar um computador de 4Ghz

n	$n!$	$n^2 2^n$
10	0 s	0s
15	300 s	0s
18	18 dias	0s
20	19 anos	0,1 s
23	200 milênios	1 s
35	?	3 horas
40	?	5 dias

Se $P \neq NP$ não conseguiremos para um problema NP-Difícil:



Algoritmos

- Casos Particulares
  - PD para Conjunto Independente de Peso Máximo no Grafo Caminho.
- Heurísticas
- Algoritmos de Aproximação
- Algoritmos Exatos
  - PD para o Knapsack, BackTracking para o Vertex-Cover, PD o TSP.

Problema da Mochila

Dado uma coleção I de n itens e uma capacidade inteira W. Cada item $i \in I$ tem:

- Um valor v_i (não negativo)
- Um peso w_i (não negativo e inteiro)

Encontrar $S \subseteq I$ cujo peso não ultrapasse W, ou seja,

$$\sum_{i \in S} w_i \leq W$$

e que maximiza $\sum_{i \in S} v_i$.

Heurísticas Gulosas

- Estamos interessados em algoritmos rápidos.
- Mas que não necessariamente chegam na solução ótima.
- Podemos então voltar a usar o paradigma de algoritmos gulosos.

Heurística Gulosa para o Knapsack

- Ordenar os itens seguindo algum critério.
- Colocar os itens na solução seguindo essa ordenação até que algum item não caiba.

Tentativa 1

- Ordenar os itens pelos mais valiosos
- Exemplo para $W = 10$:

$v_1 = 3$	$v_2 = 2$	$v_3 = 2$	$\ldots$	$v_{11} = 2$
$w_1 = 10$	$w_2 = 1$	$v_3 = 1$	$\ldots$	$w_{11} = 1$

Tentativa 2

- Ordenar pelo valor proporcional ao peso.

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \frac{v_3}{w_3} \geq \ldots \geq \frac{v_n}{w_n}$$

- Colocar os itens na solução até que um não caiba. (Na prática você pode continuar analisando a lista e continuar colocando os itens que couberem na solução)

Algoritmo 26: GreedyKnap(I, W)

Entrada: Um conjunto de itens I e uma capacidade W
Saída: Solução S

```
1 Ordenar os itens tal que v1/w1 ≥ v2/w2 ≥ v3/w3 ≥ ... ≥ vn/wn;
2 S = ∅;
3 para i = 1, 2, ..., n faça
4     se item i cabe em S então
5         S = S ∪ {i};
6     senão
7         Pare;
8 devolva S;
```


Quiz

- Considere uma instância da mochila com W = 1000.

$v_1 = 2$	$v_2 = 1000$
$w_1 = 1$	$w_2 = 1000$

- Qual seria o valor de solução dada pelo algoritmo guloso, e qual seria a solução ótima?

a) 2 e 1000
b) 2 e 1002
c) 1000 e 1002
d) 1002 e 1002

- Nessa instância a solução gulosa é 0.004% da solução ótima.
- De fato ela pode ser arbitrariamente ruim em relação a solução ótima.
- Ou seja, é possível encontrar uma instância que faça a solução gulosa ser $X\%$ da ótima, para um X tão pequeno quanto você queira.

Algoritmo de Aproximação para Knapsack

- Podemos melhorar a Heurística Gulosa adicionando o seguinte passo:

Algoritmo 27: ApproxKnap(I, W)

Entrada: Um conjunto de itens I e uma capacidade W
Saída: Solução S

1 $S = GreedyKnap(I, W)$;
2 $S' =$ o item mais valioso;
3 devolva o melhor entre S e S';

Teorema

O valor da solução de ApproxKnap é maior ou igual a 50% *da solução ótima.*

- Um algoritmo com essa caracteristica é dito um Algoritmo de $\frac{1}{2}$ aproximação, ou $\frac{1}{2}$-aproximado.

Para provar o Teorema, vamos considerar o seguinte:

- No GreedyKnap paramos de empacotar no item K, Suponha que pudêssemos completar a mochila com uma fração do item $K + 1$.
- Vamos chamar essa de uma solução Gulosa Fracionária.
- Exemplo: $W = 3$, $v_1 = 3$, $v_2 = 2$, $w_1 = w_2 = 2$.
- Solução fracionária: 4

Quiz

Seja F o valor da solução Gulosa Fracionária. Seja OPT o valor da solução ótima. Qual das alternativas é verdade:

a. $F = OPT$
b. $F > OPT$
c. $F \leq OPT$
d. $F \geq OPT$

- Seja Ap o valor da solução devolvida por *ApproxKnap*. F o valor da solução Gulosa Fracionária e OPT o valor da solução ótima. Então:

$$Ap \geq \sum_{i=1}^{K} v_i$$
$$Ap \geq v_{k+1}$$
$$2 * Ap \geq \sum_{i=1}^{K+1} v_i \geq F \geq OPT$$
$$Ap \geq \frac{1}{2} OPT$$

- Será que não poderíamos fazer uma análise mais forte e provar que a solução encontrada não é melhor que 50%?
- Será que poderíamos encontrar características na instância que nos permitiriam mostrar que na verdade a solução é melhor? (Por exemplo: todos os items tem no máximo peso $W/10$)
- Modificar o algoritmo para conseguir uma aproximação maior.

A análise de ApproxKnap é justa

- Considere a seguinte instância do problema da mochila com $W = 1000$

$v_1 = 502$	$v_2 = 500$	$v_3 = 500$
$w_1 = 501$	$w_2 = 500$	$w_3 = 500$

- A solução de ApproxKnap é 502.
- A solução ótima é 1000.
- De fato é possível construir instâncias que a solução de ApproxKnap é de fato 50% da ótima.

- Suponha então que todo item i tem $w_i \leq 10\%W$
- Se ApproxKnap (ou GreedyKnap) falharem em colocar todos os itens na solução, então a mochila está pelo menos 90% cheia.

$$Ap \geq 90\%F$$
$$\geq 90\%OPT$$

Aproximação Arbitrariamente Boa

- Dado um parâmetro $0 < \epsilon < 1$ (por exemplo, $\epsilon = 0.01$). Garantir uma $(1 - \epsilon)$-Aproximação.
- Parece bom demais. Entretanto o tempo de execução aumenta quando ϵ diminui.
- Pelo lado bom, podemos calibrar ϵ para uma boa troca entre qualidade de solução e tempo de execução.
- Esse é o melhor cenário para problemas NP-Difíceis em relação a aproximação.

- Para vários problemas não existe um algoritmo com aproximação arbitrariamente boa (de tempo polinomial) a menos que $P = NP$.
- Por exemplo: Vertex-Cover.

Arredondamento dos Valores dos Itens

- Ideia: Resolver de forma exata uma instância da Mochila ligeiramente incorreta, porém mais fácil que a instância original.
- Obs: Se os w_i e W são inteiros, podemos resolver a Mochila por Programação Dinâmica em tempo $O(nW)$. (note que no caso particular que W é polinomial em n, o algoritmo também é polinomial)
- Alternativamente: Se os v_i são inteiros, podemos resolver usando programação dinâmica em tempo $O(n^2 \max\{v_i\})$.

- Se todos os v_i forem pequenos (polinomiais em n), então usamos esse algoritmo para obter tempo polinomial.
- Plano: Jogar fora os bits menos significativos dos v_i's.

O algoritmo

- Passo 1: Arredondar para baixo os v_i para o múltiplo de m mais próximo. m depende de ϵ. Quanto Maior o m mais informação é jogada fora, e portanto menos acurado será a solução.

$$\hat{v}_i = \left\lfloor \frac{v_i}{m} \right\rfloor$$

- Passo 2: Resolva a mochila com valores $\hat{v}_i$, pesos w_i e capacidade W.

Knapsack: PD nos Valores

Nosso primeiro algoritmo de PD (nos pesos) para o Knapsack

- Pesos w_i e capacidade W eram inteiros.
- Tempo de execução $O(nW)$
- Uma das dimensões da tabela era W

Algoritmo de PD (nos valores) para o Knapsack

- Valores v_i são inteiros.
- Tempo de execução $O(n^2 \max\{v_i\})$
- Uma das dimensões da tabela é $n\max\{v_i\}$

- Ideia: Uma das dimensões da tabela será i que indica o prefixo $1, \ldots, i$ que é permitido usar.
- O segundo parâmetro x será o valor que desejamos obter (ou maior). E vamos procurar o menor peso que consegue obter aquele valor. $x = 0, 1, \ldots, n \cdot v_{max}$.
- $A[i][x]$ indicará o menor peso necessário para obter valor pelo menos x usando apenas os itens $1, \ldots, i$.

$$A[i][x] = \begin{cases} A[i-1][x] \\ w_i + A[i-1][x - v_i] \end{cases}$$

- $A[i-1][x-v_i]$ é zero se $v_i \geq x$

Algoritmo 28: KnapsackPDV(I, W)

Entrada: Um conjunto de Itens I e uma capacidade W
Saída: Valor de uma solução ótima

```
1 A[n][n max{v_i}];
2 A[0][0] = 0;
3 para x = 1, 2, ..., n max{v_i} faça A[0][x] = +∞;
4 para i = 1, 2, ..., n faça
5     para x = 1, 2, ..., n max{v_i} faça
6         A[i][x] = min{A[i − 1][x]; w_i + A[i − 1][x − v_i]};
7 devolva o maior x tal que A[n][x] ≤ W;
```

- Tempo de Execução $O(n^2 \max\{v_i\})$

Algoritmo $(1-\epsilon)$-aproximado

- Passo 1: Calcular $\hat{v}_i = \lfloor \frac{v_i}{m} \rfloor$ para todo item.
- Passo 2: Resolver o problema com $\hat{v}$ usando KnapsackPDV.

Plano:

- Quão grande pode ser m, que ainda garanta uma $(1-\epsilon)$-aproximação
- Dado esse m qual é o tempo de execução do algoritmo?

Quiz

Suponha que transformamos v_i em $\hat{v}_i$. Qual das seguintes alternativas é verdade?

a. $\hat{v}_i$ está entre $v_i - m$ e v_i
b. $\hat{v}_i$ está entre v_i e $v_i + m$
c. $m \cdot \hat{v}_i$ está entre $v_i - m$ e v_i
d. $m \cdot \hat{v}_i$ está entre $v_i - 1$ e v_i

Análise da Acurácia

Concluimos que:

- 1) $v_i \geq m \cdot \hat{v}_i$
- 2) $m \cdot \hat{v}_i \geq v_i - m$

Seja S^* a solução ótima para o problema original, e S a solução para o problema com $\hat{v}_i$. Como resolvemos de forma ótima o problema usando $\hat{v}_i$ obtemos:

- 3)

$$\sum_{i \in S} \hat{v}_i \geq \sum_{i \in S^*} \hat{v}_i$$

$$\sum_{i \in S} \hat{v}_i \geq \sum_{i \in S^*} \hat{v}_i$$

$$m \cdot \sum_{i \in S} \hat{v}_i \geq m \cdot \sum_{i \in S^*} \hat{v}_i$$

$$\sum_{i \in S} v_i \geq m \cdot \sum_{i \in S} \hat{v}_i \geq m \cdot \sum_{i \in S^*} \hat{v}_i \geq \sum_{i \in S*} (v_i - m)$$

$$\sum_{i \in S} v_i \geq (\sum_{i \in S*} v_i) - nm$$

$$\sum_{i \in S} v_i \geq (\sum_{i \in S*} v_i) - nm$$

Queremos obter

$$\sum_{i \in S} v_i \geq (1 - \epsilon) \sum_{i \in S*} v_i = \sum_{i \in S*} v_i - \epsilon \sum_{i \in S*} v_i$$

Para isso precisamos escolher um m pequeno o bastante tal que:

$$mn \leq \epsilon \sum_{i \in S*} v_i$$

$$mn \leq \epsilon \sum_{i \in S*} v_i$$

Não sabemos qual a solução ótima S^*, mas podemos pegar um m ainda menor.

$$mn = \epsilon \max\{v_i\}$$

$$m = \frac{\epsilon \max\{v_i\}}{n}$$

Algoritmo $(1 - \epsilon)$-aproximado

Algoritmo 29: $(1 - \epsilon)$-ApproxKnap(I, W, ϵ)

Entrada: Um conjunto de itens I, uma capacidade W e um fator ϵ
Saída: Valor de uma solução ótima

1 Calcule v_{max};
2 $m = \frac{\epsilon v_{max}}{n}$;
3 **para** $i = 1, 2, \ldots, n$ **faça**
4 $\quad \hat{v}_i = \lfloor \frac{v_i}{m} \rfloor$;
5 devolva *KnapsackPDV*(I com valores $\hat{v}$, W);

Complexidade

- Escolhendo $m = \frac{\epsilon v_{max}}{n}$ garantimos que o valor da nossa solução é $\geq (1-\epsilon)\cdot$ valor do ótimo
- Como o calculo dos $\hat{v}_i$ é linear, a complexidade total do algoritmo é a mesma do *KnapsackPDV*.

$$O(n^2 \hat{v}_{max})$$

$$\hat{v}_{max} \leq \frac{v_{max}}{m} = \frac{v_{max}}{\frac{\epsilon v_{max}}{n}} = \frac{v_{max} n}{\epsilon v_{max}} = \frac{\cancel{v_{max}} n}{\epsilon \cancel{v_{max}}} = \frac{n}{\epsilon}$$

Dessa forma a complexidade do nosso algoritmo $(1-\epsilon)$-aproximado é

$$O(n^2 \hat{v}_{max}) = O\left(n^2 \cdot \frac{n}{\epsilon}\right) = O\left(\frac{n^3}{\epsilon}\right)$$

Limitantes

Em um problema de maximização:

- Um limitante superior é um valor maior ou igual que o ótimo. Pode ser obtido através de alguma relaxação do problema por exemplo. (limitante dual)
- Um limitante inferior é um valor menor ou igual que o ótimo. Poder ser obtido por uma heurística por exemplo. (limitante primal)

Exemplo: Problema da Mochila Binária.

Itens	1	2	3	4	5
Pesos	2	5	4	6	5
Valor	10	11	10	13	14

Capacidade da Mochila: 14

Limitante Inferior:

- Qualquer solução pode ser considerada um limitante inferior.
- Por exemplo se pegarmos os itens na ordem que foram dados até encher a mochila.

Pesos	**2**	**5**	**4**	6	5
Valor	**10**	**11**	**10**	13	14

Solução: 31

- A solução ótima que tem valor z^* é pelo menos 31. Ou seja $z^* \geq 31$
- Podemos tentar fortalecer esse limitante.
- E se escolhermos os itens na ordem dos que tem o melhor custo-benefício (valor/peso)

Valor/Peso (Custo-Benefício) 5 2.2 2.5 2.166 2.8

Pesos	**2**	5	**4**	6	**5**
Valor	**10**	11	**10**	13	**14**

Solução: 34, ou seja $z^* \geq 34$

Limitante Superior (Dual)

Itens	1	2	3	4	5
Pesos	2	5	4	6	5kg
Valor	10	11	10	13	14

- Se conseguíssemos preencher toda a mochila com o item que vale mais a cada kg?

Valor/Peso (Custo-Benefício) 5 2.2 2.5 2.166 2.8

- Se enchermos os 14kg que cabem na mochila com 5$ por kg.
- Um limitante superior para esse problema é $14 * 5 = 70$, ou seja $z^* \leq 70$
- Ou seja é verdade que nenhuma solução (incluindo a ótima) pode ser maior que 70.
- Esse limitante claro é bem fraco. E se enchermos a mochila só com os itens que temos até encher completamente a mochila, relaxando a integralidade.

Itens	1	2	3	4	5
Pesos	2	5	4	6	5kg
Valor	10	11	10	13	14

Valor/Peso (Custo-Benefício) 5 2.2 2.5 2.166 2.8

Pegamos o item 1, depois pegamos o item 5, depois o item 3 e enchemos o resto da mochila com 0.6 do item 2.

$$10 + 0.6 * 11 + 10 + 14 = 40.6$$

- Concluímos que a solução ótima pode ser no máximo 40.6.
- De fato como todos os valores são inteiros, podemos afirmar que $z^* \leq 40$

Itens	1	2	3	4	5
Pesos	2	5	4	6	5kg
Valor	10	11	10	13	14

- Concluimos que $34 \leq z^* \leq 40$.
- Se encontrarmos uma solução com custo igual ao limitante superior (40 no exemplo), temos certeza que ela é ótima.
- Mas a solução ótima para esse problema é a seguinte

Pesos	**2**	5	4	**6**	**5kg**
Valor	**10**	11	10	**13**	**14**

Solução: 37.

Branch-and-Bound

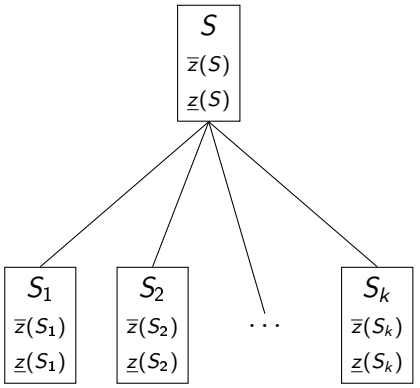
- Técnica empregada na resolução de problemas difíceis de otimização combinatória.
- O BnB enumera implicitamente todas as soluções do problema.
- Implicitamente, pois se explorasse de fato todas as soluções seria equivalente a um algoritmo de força-bruta.

- Os algoritmos de BnB são uma aplicação do paradigma de divisão e conquista.
- O Branch-and-Bound tem duas partes principais, que como você pode imaginar são:
  - *Branch*: ramificação, que é o processo de dividir o problema.
  - *Bound*: que é a busca por limitantes (inferiores e superiores) para cada subproblema.

Branch-and-Bound

- Considere um problema de maximização qualquer, em que desejamos encontrar a solução ótima.
- Seja S o conjunto de todas as soluções viáveis.
- Seja $f : S \to \mathbb{R}$ a função objetivo que mapeia cada solução $s \in S$ a um valor real.

- Vamos considerar que podemos calcular bons limitantes superiores e inferiores para um conjunto $S' \subseteq S$.
  - Seja $\overline{z}(S')$ um limitante dual (superior) para S', ou seja, um valor tal que nenhuma solução de S' tenha um valor de função objetivo maior que $\overline{z}(S')$. $f(s \in S') \leq \overline{z}(S')$
  - Seja $\underline{z}(S')$ um limitante primal (inferior) para S', ou seja, existe (com certeza) alguma solução em S' que tenha valor igual ou superior a $\underline{z}(S')$. $max\{f(s)|s \in S'\} \geq \underline{z}(S')$
- Seja $\{S_1, S_2, \ldots, S_k\}$ uma partição de S, ou seja, $S_1 \cup S_2 \cup \ldots \cup S_k = S$.
- Podemos calcular os limitantes primais e duais tanto para S quanto para S_1, S_2, ..., S_k.

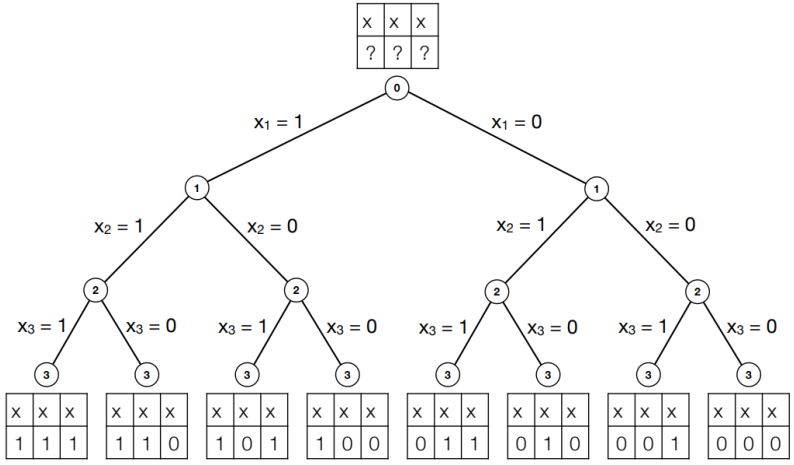


- Suponha que $\overline{z}(S_i) \leq \underline{z}(S_j)$. Como estamos buscando a solução ótima, ou seja, a solução com o maior valor. Sabemos que a melhor solução em S_i vai ser pior (ou igual) a melhor solução em S_j e por isso podemos buscar a solução apenas em S_j, *podando* S_i. Essa é a comumente chamada **poda por limitante**.

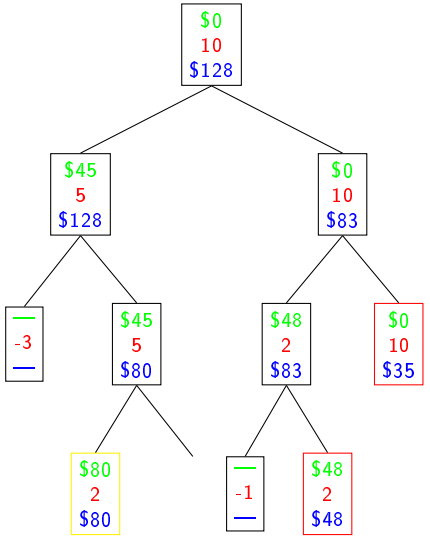
$$\begin{aligned} \text{maximize} \quad & 45x_1 + 48x_2 + 35x_3 \\ \text{subject to} \quad & \\ & 5x_1 + 8x_2 + 3x_3 \leq 10 \\ & x_i \in \{0, 1\} \quad (i \in 1..3) \ {}_1 \end{aligned}$$

- Como vamos subdividir o espaço de busca? Vamos dividir entre as decisões que podemos fazer em cada item.

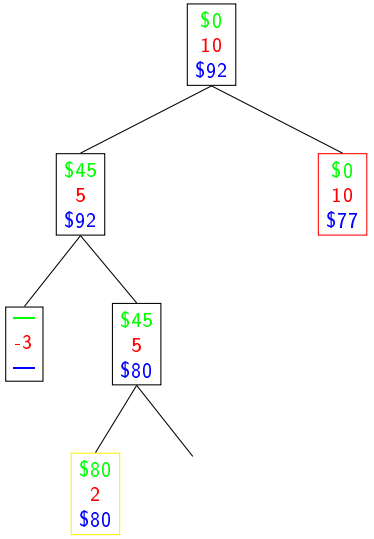
[1] Discrete Optimization, Professor Pascal Van Hentenryck



- Vamos simular a exploração
- Limitante inferior: Os itens que já estão na solução parcial.
- Limitante superior: Vamos relaxar a capacidade da mochila. (Ainda vamos verificar quando a mochila ultrapassa a capacidade)
- Vamos denotar cada solução pelo valor do limitante inferior (verde) a capacidade residual (vermelho) e o limitante superior (azul).



- Limitante superior: Vamos relaxar a integralidade da mochila, ou seja, podemos pegar uma fração de um item.

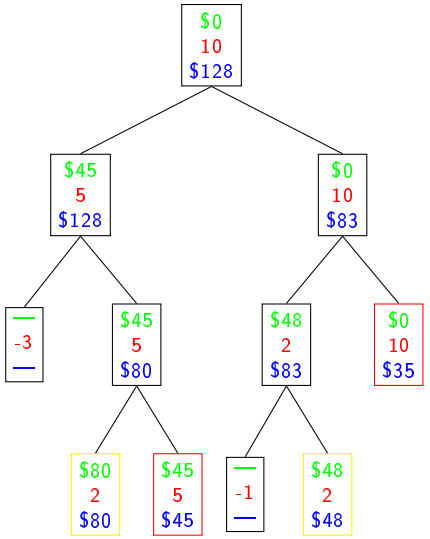


Branching Strategies

Estratégias de Ramificação

- Podemos elaborar diferentes estratégias para explorar os nós de uma árvore de branch-and-bound.
- A Depth First (que vimos anteriormente) sempre desce o mais profundo possível (indo sempre o mais a esquerda possível e depois para a direita)
- Cada estratégia pode ser um desempenho diferente dependendo do problema (e da instância).
- Mas podemos analisar alguns aspectos. Por exemplo, qual a eficiência de memória do Depth First?

Best-First



Best-First

- Sempre expande o nó "Mais promissor", ou seja, aquele que tem o maior limitante superior (no caso de um problema de maximização)
- Ele poda sempre que encontra um nó em que o limitante já é pior do que a melhor solução conhecida.
- Ele é eficiente em memória? Não muito, pode ter um número muito grande de nós ativos e cada nó precisa conter uma forma de recuperar a solução parcial.
- Ele é fácil de implementar? Normalmente é mais difícil do que o Depth-First

Ramificações

- A ideia então é dividir o espaço de soluções, calculando os limitantes e podando os nós que não fornecem soluções promissoras.
- A forma de dividir, depende do problema e da estratégia aplicada. A ideia é que a cada divisão represente uma escolha diferente.

- Por exemplo, no problema da mochila uma divisão pode ser decisão por colocar o item i na mochila e a decisão de não colocar o item i na mochila.
- Nesse exemplo obteríamos uma árvore binária, e cada nível dessa árvore pode representar a escolha de um item diferente.

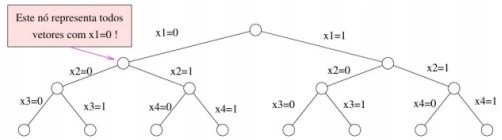


Figura 1.8. Árvore de enumeração completa para o problema binário da mochila com 3 itens.

[2]

[2] https://ic.unicamp.br/ fkm/lectures/intro-otimizacao.pdf

Outros exemplos:

- No problema do caixeiro viajante cada escolha poderia ser a escolha do próximo vértice a visitar, e nesse caso a árvore não seria binária.
- No problema de encontrar a clique máxima, uma escolha poderia ser a inclusão de um item na clique.
- No problema do *bin packing*, uma escolha poderia ser uma atribuição de um item a um contêiner.
- etc.

Algoritmo[3]

1. B&B; (* considerando problema de **maximização** *)
2. Ativos $\leftarrow$ {nó raiz}; melhor-solução $\leftarrow$ {}; $\underline{z} \leftarrow -\infty$;
3. **Enquanto** (Ativos não está vazia) **faça**
4. Escolher um nó k em Ativos para ramificar;
5. Remover k de Ativos;
6. Gerar os filhos de k: $n_1, \ldots, n_q$ computando $\overline{z}_{n_i}$ e $\underline{z}_{n_i}$ correspondentes;
   (* definir $\overline{z}_{n_i}$ e $\underline{z}_{n_i}$ iguais a $-\infty$ para subproblemas inviáveis *)
7. **Para** $j = 1$ **até** q **faça**
8. **se** ($\overline{z}_{n_i} \leq \underline{z}$) **então** podar o nó n_i; (* inclui os 3 casos *)
9. **se não**
10. **Se** (n_i representa uma única solução) **então** (* atualizar melhor limitante primal *)
11. $\underline{z} \leftarrow \overline{z}_{n_i}$; melhor-solução $\leftarrow$ {solução de n_i};
12. **se não** adicionar n_i à lista Ativos.

[3] https://ic.unicamp.br/ fkm/lectures/intro-otimizacao.pdf

Outros ramificações:

- Least-Discrepancy: Ramifica em ondas, a cada onda ele permite que uma curva a mais aconteça na árvore.