UDESC

Projeto e Análise de Algoritmos

Prof. Diego Buchinger

diego.buchinger@outlook.com

diego.buchinger@udesc.br

Prof. Cristiano Damiani Vasconcellos

cristiano.vasconcellos@udesc.br

Bibliografia

Algoritmos. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein. Campus. [**Bíblia**]

Algorithms. Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. McGraw Hill.

Complementar:

Complexidade de Algoritmos. Toscani, L.V. e Veloso, P.A.S. Instituto de Informática da UFRGS. Editora Sagra.

Análise de Algoritmos

Analisar um algoritmo significa prever os recursos que algoritmo necessita. Por exemplo, memória, largura de banda e mais frequentemente o tempo de computação.

Para analisar um algoritmo é necessário definir um modelo de computação. O modelo de computação do computador tradicional é o RAM (*Random Access Machine*) onde as instruções são executadas em sequência, sem concorrência, e os dados são armazenados em células de memória com acesso aleatório.

Análise de Algoritmos

Contar o número de todas as instruções que são executadas pelo algoritmo.

Por exemplo: m **load**, n **store**, o **add**, p **sub**, q **div**, r **mul**, s **call**, t **ret**, u **cmp**, v **jump**, etc.

Qual o tempo de execução?

```
int pesquisa(Estrutura *v, int n, int chave){
	int i;
	for (i = 0; i < n; i++)
		if (v[i].chave == chave)
			return i;
	return -1;
}
```

O número de instruções e o tempo de execução depende do processador, compilador, velocidade de acesso à memória, tamanho de memória (cache e ram) etc.

Qual o tempo de execução?

Comparação de desempenho na resolução de sistemas lineares considerando tempos de operações de um computador real:

n	Método de Cramer	Método de Gauss
2	22 µs	50 µs
3	100 µs	159 µs
4	463 µs	353 µs
5	2,15 ms	666 µs
10	4,62 s	4,95 ms
20	247 dias	38,63 ms
40	$1{,}45 * 10^{13}$ anos	0,315 s

Qual o tempo de execução?

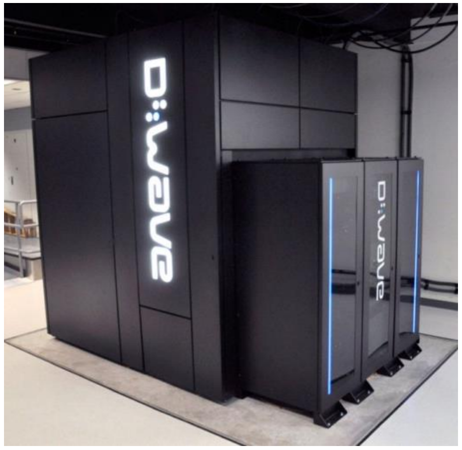
- Ok, mas e o avanço tecnológico, produzindo máquinas cada vez mais rápidas não faz o estudo de complexidade perder importância?

Computador 19xx

100x mais rápido

Computador 2016

2^b mais rápido



Computador quântico

Qual o tempo de execução?

- Análise de impacto do aumento de velocidade dos computadores para o Método de Cramer:

n	Computador 19xx	Computador 2016
3	100 µs	1 µs
5	2,15 ms	21,5 µs
7	46,274 ms	463 µs
10	4,62 s	46,2 ms
12	1,66 min	1 s
15	2,76 horas	1,656 min
20	247 dias	2,47 dias
40	$1{,}45 * 10^{13}$ anos	$1{,}45 * 10^{11}$ anos

Análise de Algoritmos

Prever os recursos de que o algoritmo necessitará

A complexidade vem ganhando destaque a ponto de que alguns autores dizem que este tema é o coração da Computação [Toscani e Veloso, 2001].

- Complexidade na fase de projeto do algoritmo
- Intratabilidade de problemas:
  - Problemas NP-Completos e NP-Difícil
  - Soluções alternativas (aproximações), uso de programação dinâmica.

Programa e Plano de Ensino

- Plano de Ensino
  - Objetivos e ementa
  - Conteúdo programático
  - Avaliação
  - Bibliografia
- Plano de Aulas

Disponível na página!

Atividade 1

- Elabore o melhor algoritmo para receber uma sequencia de ‘*n*’ números inteiros e dizer quantas vezes o número ‘*m*’ apareceu nesta sequência.

- NOTA: existe alguma consideração diferente caso ‘*m*’ seja um inteiro entre 0 e 10.000, ou um inteiro entre 0 e 1.000.000.000.000?

Conceitos Básicos de Complexidade

Medidas de Complexidade

- Como calcular a quantidade de trabalho requerido por um algoritmo, ou seja, sua complexidade?

Medidas de Complexidade

- Como calcular a quantidade de trabalho requerido por um algoritmo, ou seja, sua complexidade?
  - Depende do tamanho da entrada;
  - Depende dos valores da entrada;

Ex: ordenação de uma lista de ‘*n*’ elementos:

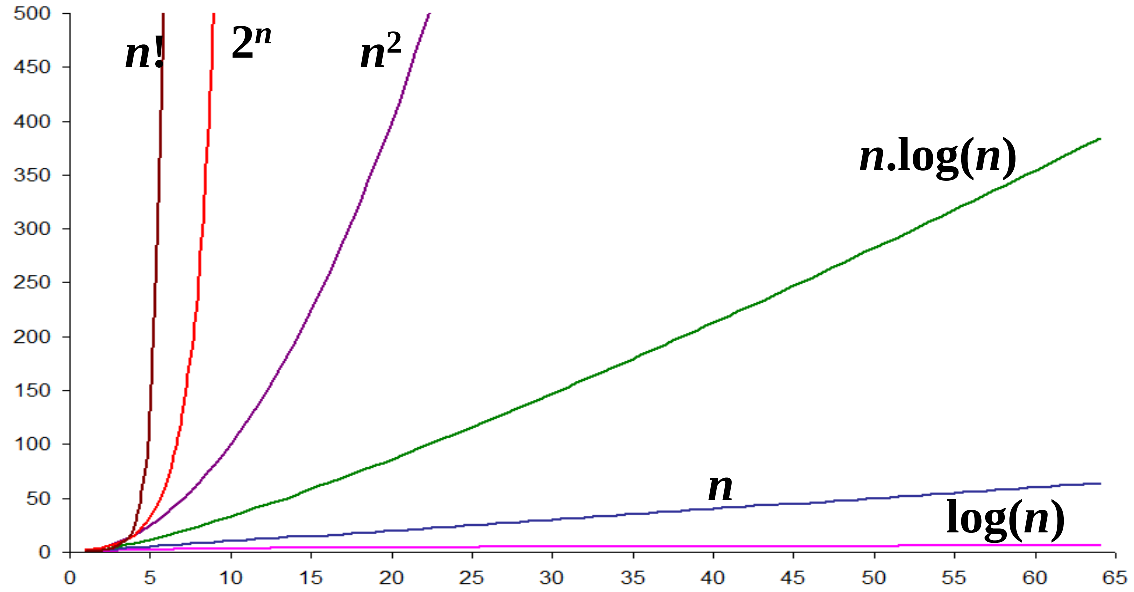
lista com elementos já ordenados

vs

lista com elementos totalmente desordenados

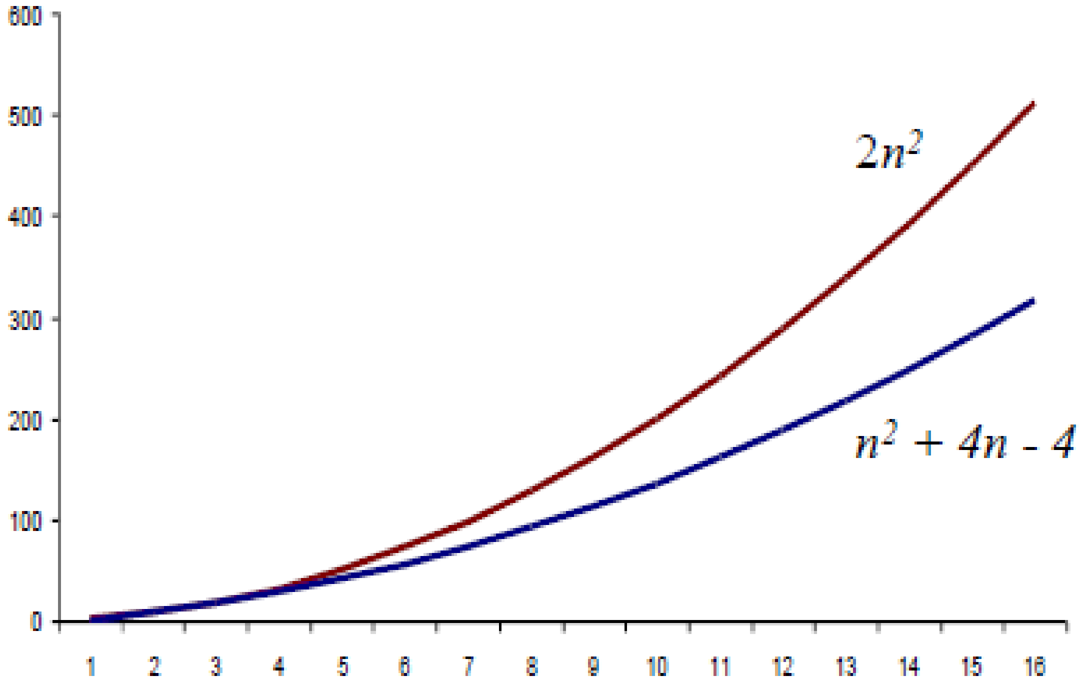
Medidas de Complexidade

CONSIDERAÇÃO I: trabalhar com valores grandes para ‘n’ (entrada). Assim, ordens de crescimento são destacadas.



Notação Assintótica
(Notação O grande – Limite Superior)

Uma função *g(n)* domina assintoticamente outra função *f(n)* se existem duas constantes positivas *c* e n_0 tais que, para $n > n_0$, temos $|f(n)| \leq c.|g(n)|$ $\rightarrow$ $f(n) = O(g(n))$



$$n^2 + 4n - 4 = O(n^2)$$

$$2n^2 = O(n^2)$$

Algumas Operações com Notação O

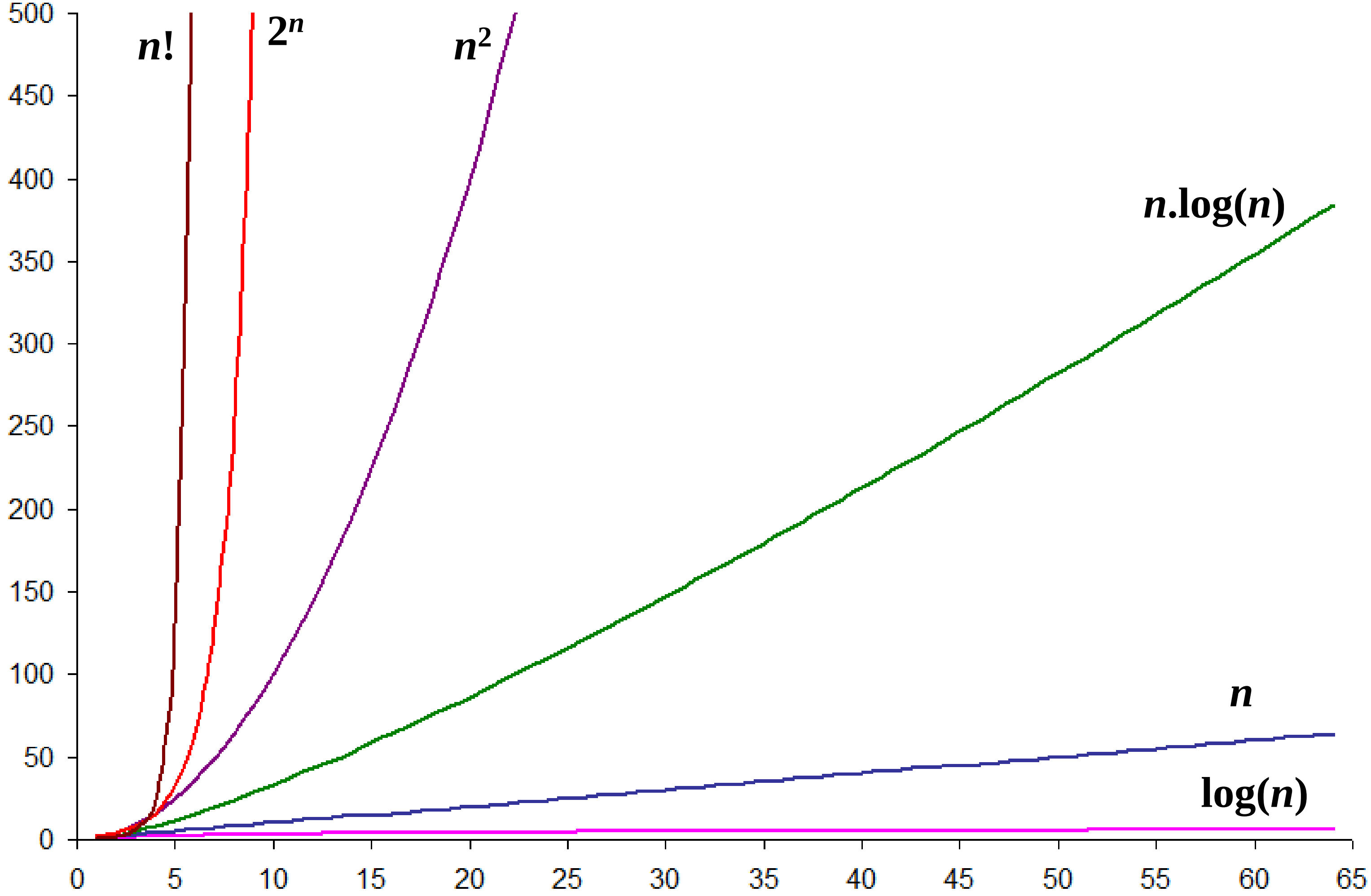
$c.O(f(n)) = O(f(n))$, onde c é uma constante.

$O(f(n)) + O(g(n)) = O(MAX(f(n), g(n)))$

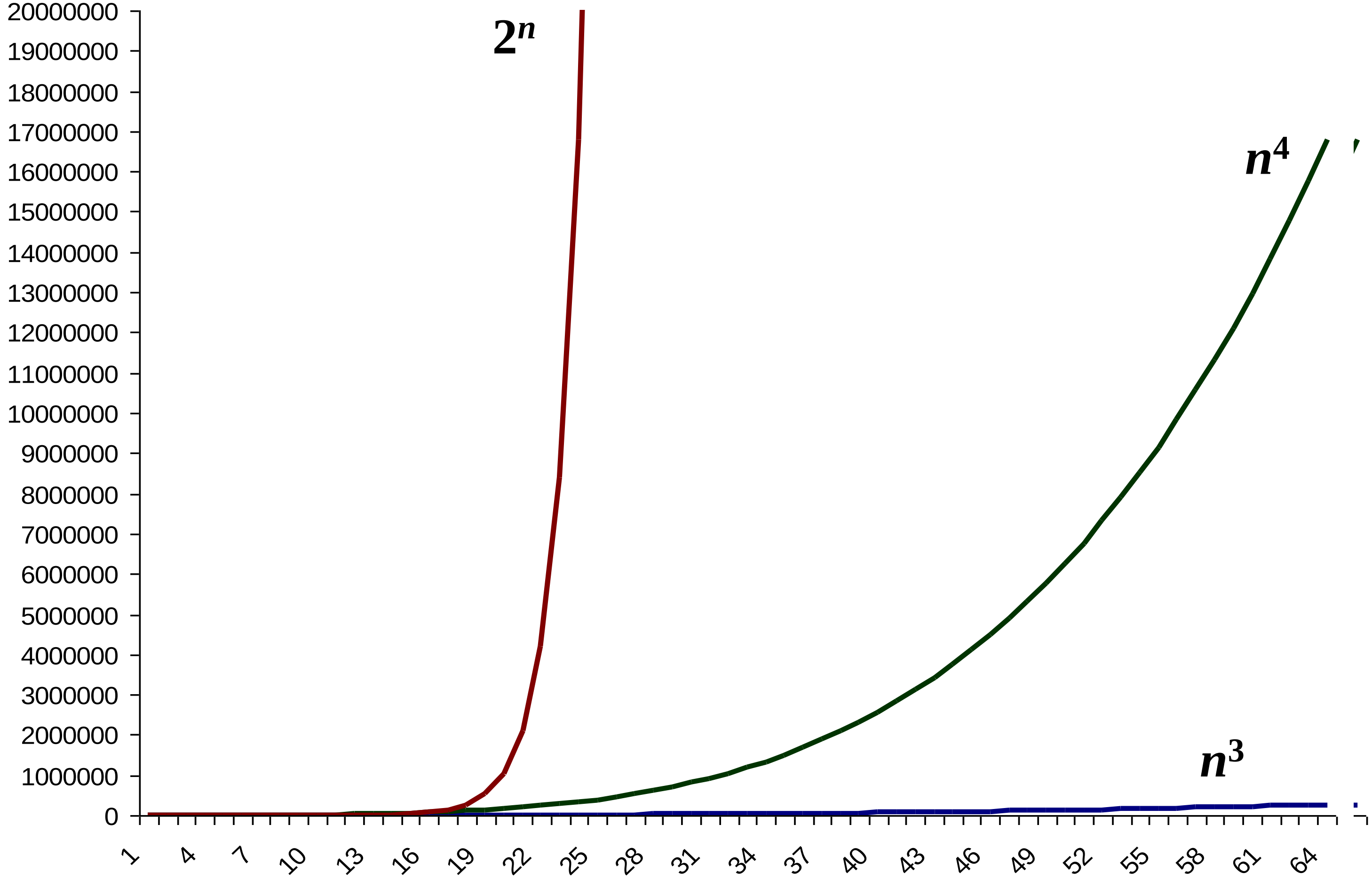
$n.O(f(n)) = \mathrm{O}(n.f(n))$

$O(f(n)).O(g(n)) = O(f(n).g(n))$

Crescimento de Funções



Crescimento de Funções



Hierarquia de funções

Hierarquia de funções do ponto de vista assintótico:

$$1 \prec \log\log n \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}$$

onde ε e c são constantes arbitrárias tais que $0 < \varepsilon < 1 < c$.

Medidas de Complexidade

CONSIDERAÇÃO II: Ignorar o custo das instruções (tempo constante) e focar na análise do crescimento do uso de um recurso (tempo, espaço) em relação ao crescimento da entrada.

Ex: ordenar uma lista de ‘n’ elementos e mostrar a lista ordenada

n	Ordenação Bolha	printf vetor
100	37,8 µs	8,532 ms
200	148,4 µs	17,847 ms
1.000	3,748 ms	91,569 ms
10.000	247 ms	860,205 ms
50.000	5,307 s	4,277 s
100.000	20,422 s	8,693 s

Medidas de Complexidade

CONSIDERAÇÃO III: pode-se analisar os valores de entrada com perspectivas diferentes:

- **Melhor caso** => menor complexidade para um valor de ‘n’;
- **Pior caso** => maior complexidade para um valor de ‘n’;
- **Complexidade esperada ou média** => leva-se em conta a probabilidade de ocorrência de cada entrada de um mesmo tamanho ‘n’.

- Pode-se antecipar alguma relação entre as complexidades média e pior caso de um algoritmo qualquer?

Medidas de Complexidade

```
int pesquisa(Estrutura *v, int n, int chave) {
    int i;
    for (i = 0; i < n; i++)
        if (v[i].chave == chave)
            return i;
    return -1;
}
```

Em que situação ocorre o melhor caso?

Em que situação ocorre o pior caso? E o caso médio?

Medidas de Complexidade

Melhor caso: Caso o primeiro registro seja o registro procurado será necessária apenas uma comparação.

Logo, podemos dizer que a complexidade é **constante**:

O(1)

(o correto seria usar outra letra grega para
o melhor caso mas, vamos por partes)

Medidas de Complexidade

Pior caso: Caso o último registro acessado seja aquele que se procura:

```
int pesquisa(Estrutura *v, int n, int chave) {
    int i;                               // O(1)
    for (i = 0; i < n; i++)              // O(n)
        if (v[i].chave == chave)         // O(1)
            return i;                    // O(1)
    return -1;                           // O(1)
}
```

Logo, podemos dizer que a função pesquisa tem complexidade **O(n)** para o pior caso.

Medidas de Complexidade

Caso médio: Caso o *i-ésimo* registro seja o registro procurado são necessárias i comparações. Sendo p_i a probabilidade de procurarmos o *i-ésimo* registro temos:

$$f(n) = 1.p_1 + 2.p_2 + \ldots + n.p_n.$$

Considerando que a probabilidade de procurar qualquer registro é a mesma probabilidade, temos:

$$p_i = 1 / n \quad \text{para todo } i.$$

$$f(n) = \frac{1}{n}(1 + 2 + \ldots + n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{(n+1)}{2}$$

Logo, temos uma complexidade linear: $\frac{(n+1)}{2} = \boldsymbol{O}(\,\boldsymbol{n}\,)$

Medidas de Complexidade

CONSIDERAÇÃO IV: pode-se analisar a complexidade em relação a diferentes recursos. Os mais usuais são: <u>tempo</u> e <u>espaço</u>.

Complexidade de espaço:

- Se os dados possuem representação natural, (ex. matriz) considera-se apenas o espaço extra utilizado pelo algoritmo;
- Se os dados podem ser representados de várias formas (ex. grafo) deve-se considerar o espaço utilizado por sua representação (matriz ou lista encadeada).

Exemplo
(Bubble Sort)

```
void bubble(int *v, int n){
    int i, j, aux;
    for (i = n - 1; i > 0; i--){
        for (j = 0; j < i; j++){
            if (v[j] > v[j+1]){
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
        }
    }
}
```

Exemplo (Ordenação por Seleção)

```c
void selectionSort(int *v, int n){
    int i, j, x, aux;
    for (i = 0; i < n; i++){
        x = i;
        for (j = i+1; j < n; j++){
            if( v[j] < v[x] )
                x = j;
        }
        aux = v[i];
        v[i] = v[x];
        v[x] = aux;
    }
}
```

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Exemplo (Ordenação por Inserção)

```c
void insercao(int *v, int n){
    int i, j, x;
    for (i = 1; i < n; i++){
        x = v[i];
        j = i - 1;
        while (j >= 0 && v[j] > x){
            v[j+1] = v[j];
            j--;
        }
        v[j+1] = x;
    }
}
```

6 5 3 1 8 7 2 4

Atividade

Elabore os seguintes algoritmos e analise o seu tempo de execução para diferentes entradas e a sua complexidade de tempo.

- Implemente um algoritmo (função) que recebe como parâmetro dois valores inteiros a e b e calcula a^b.

- Implemente um algoritmo (função) que recebe duas matrizes quadradas de mesma ordem (n) e realiza a multiplicação entre elas.

Análise de Complexidade de Tempo de Algoritmos Recursivos

Algoritmos Recursivos

Geralmente a análise de complexidade de algoritmos recursivos é um pouco mais complexa pois é preciso entender bem a recursividade do algoritmo.

- Quantas vezes um algoritmo chama a si mesmo?
- Existe um pior caso? Quantas chamadas recursivas são feitas?
- Devemos considerar o tempo gasto na chamada de função? E o espaço utilizado na memória?

Exemplo (Fatorial)

```
int fatorial( int n ){
   if (n == 0)
        return 1;
   return n * fatorial( n-1 );
}
```

Exemplo (Fatorial)

```
int fatorial( int n ){
   if (n == 0)
       return 1;
   return n * fatorial( n-1 );
}
```

Relação de Recorrência:

$T(n) = T(n\text{-}1) + O(1)$

$T(0) = O(1)$

Exemplo (Pesquisa Binária)

```
int pesqbin(int *v, int p, int r, int e){
   int q;
   if ( r < p )
       return -1;
   q = (p + r)/2;
   if (e == v[q])
       return q;
   if (e < v[q])
       return pesqbin(v, p, q-1, e);
   return pesqbin(v, q+1, r, e);
}
```

** Lembrete: o vetor deve estar ordenado!!*

Exemplo
(Pesquisa Binária)

```
int pesqbin(int *v, int p, int r, int e){
   int q;
   if ( r < p )
        return -1;
   q = (p + r)/2;
   if (e == v[q])
        return q;
   if (e < v[q])
        return pesqbin(v, p, q-1, e);
   return pesqbin(v, q+1, r, e);
}
```

Relação de Recorrência:

$T(n) = T(n/2) + O(1)$

$T(1) = O(1)$

A relação abaixo descreve de forma mais precisa a execução, mas a simplificação acima não altera a complexidade:

$T(n) = T(n/2 - 1) + O(1)$

$T(1) = O(1)$

Relação de Recorrência

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
T(n/2) &= T(n/2^2) + 1 \\
T(n/2^2) &= T(n/2^3) + 1 \\
\ldots & \qquad \ldots \\
T(n/2^{h-1}) &= T(n/2^h) + 1
\end{aligned}
$$

Relação de Recorrência:

$T(n) = T(n/2) + O(1)$

$T(1) = O(1)$

Relação de Recorrência

$$
\begin{aligned}
T(n) &= \cancel{T(n/2)} + 1 \\
\cancel{T(n/2)} &= \cancel{T(n/2^2)} + 1 \\
\cancel{T(n/2^2)} &= \cancel{T(n/2^3)} + 1 \\
&\dots \\
\cancel{T(n/2^{h-1})} &= T(n/2^h) + 1
\end{aligned}
$$

Relação de Recorrência:

$T(n) = T(n/2) + O(1)$

$T(1) = O(1)$

$$T(n) = \underbrace{1 + 1 + \dots + 1}_{} + T(1)$$

h vezes => ops... Mas quanto vale h??

Relação de Recorrência

$T(n) \quad = \cancel{T(n/2)} + 1$

$\cancel{T(n/2)} \quad = \cancel{T(n/2^2)} + 1$

$\cancel{T(n/2^2)} \quad = \cancel{T(n/2^3)} + 1$

...

$\cancel{T(n/2^{h-1})} \quad = T(\ n/2^h\ ) + 1$

$$\boxed{\frac{n}{2^h} = 1}$$

$$\boxed{\begin{array}{l} n = 2^h \\ h = log_2\, n \end{array}}$$

$$T(n) = \underbrace{1 + 1 + \ldots + 1}_{} + T(1)$$

h vezes, ou seja, $log_2\, n$ *vezes*

O(log n)

Mudança de base:

$$\log_b n = \frac{\log_a n}{\log_a b}$$

Exemplo (Pesquisa Binária)

```
int pesqbin2 (int *v, int n, int e) {
    int p, q, r;

    p = 0; r = n-1;
    do {
        q = (p + r) / 2;
        if (e == v[q])
            return q;
        if (e < v[q])
            r = q -1;
        else
            p = q + 1;
    } while (p<= r);
    return -1;
}
```

Recursividade de cauda

Uma função apresenta recursividade de cauda se nenhuma operação é executada após o retorno da chamada recursiva, exceto retornar seu valor.

Em geral, compiladores, que executam otimizações de código, substituem as funções que apresentam recursividade de cauda por uma versão não recursiva dessa função.

Exemplo (Fibonacci)

```
/* Implementação ruim */
int fib( int n ){
    if( n == 0 || n == 1 )
        return 1;
    return fib( n-1 ) + fib( n-2 );
}
```

Exemplo (Fibonacci)

```
/* Implementação ruim */
int fib( int n ){
    if( n == 0 || n == 1 )
        return 1;
    return fib( n-1 ) + fib( n-2 );
}
```

Relação de Recorrência:

$T(n) = T(n\text{-}1) + T(\text{n-2}) + O(1)$

$T(1) = O(1)$

$T(0) = O(1)$

Exemplo (Fibonacci)

```
/* Implementação ruim */
int fib( int n ){
    if( n == 0 || n == 1 )
        return 1;
    return fib( n-1 ) + fib( n-2 );
}
```

Relação de Recorrência simplificada:

$T(n) = 2T(n\text{-}1) + O(1)$

$T(0) = O(1)$

Relação de Recorrência

T(n) = 2T(n-1) + O(1)

2T(n-1) = 2^2T(n-2) + 2O(1)

2^2T(n-2) = 2^3T(n-3) + 2^2O(1)

… …

2^{n-1}T(1) = 2^nT(n-n) + 2^{n-1}O(1)

2^nT(0) = 2^n O(1)

$$T(n) = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\boldsymbol{O(\ 2^n\ )}$$

Relação de Recorrência:

T(n) = 2T(n-1) + O(1)

T(0) = O(1)

Exemplo (Fibonacci)

Ao usar a relação de recorrência correta chegaríamos em uma resposta parecida:

Relação de Recorrência:

$T(n) = T(n\text{-}1) + T(n\text{-}2) + O(1)$

$T(1) = O(1)$

$T(0) = O(1)$

$$\boldsymbol{O(\ \varphi^{n}\ )}$$

$$\varphi = \frac{1+\sqrt{5}}{2}$$

$$\varphi \cong 1{,}6180$$

Propriedades dos Somatórios

$$\sum_{i=0}^{n} ca_i = c\sum_{i=0}^{n} a_i \quad \text{(Distributiva)}$$

$$\sum_{i=0}^{n} (a_i + b_i) = \sum_{i=0}^{n} a_i + \sum_{i=0}^{n} b_i \quad \text{(Associativa)}$$

$$\sum_{i=n}^{0} a_i = \sum_{i=0}^{n} a_i \quad \text{(Comutativa)}$$

Propriedades dos Somatórios

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

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$$

$$\sum_{i=1}^{\log n} i = n \log n$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

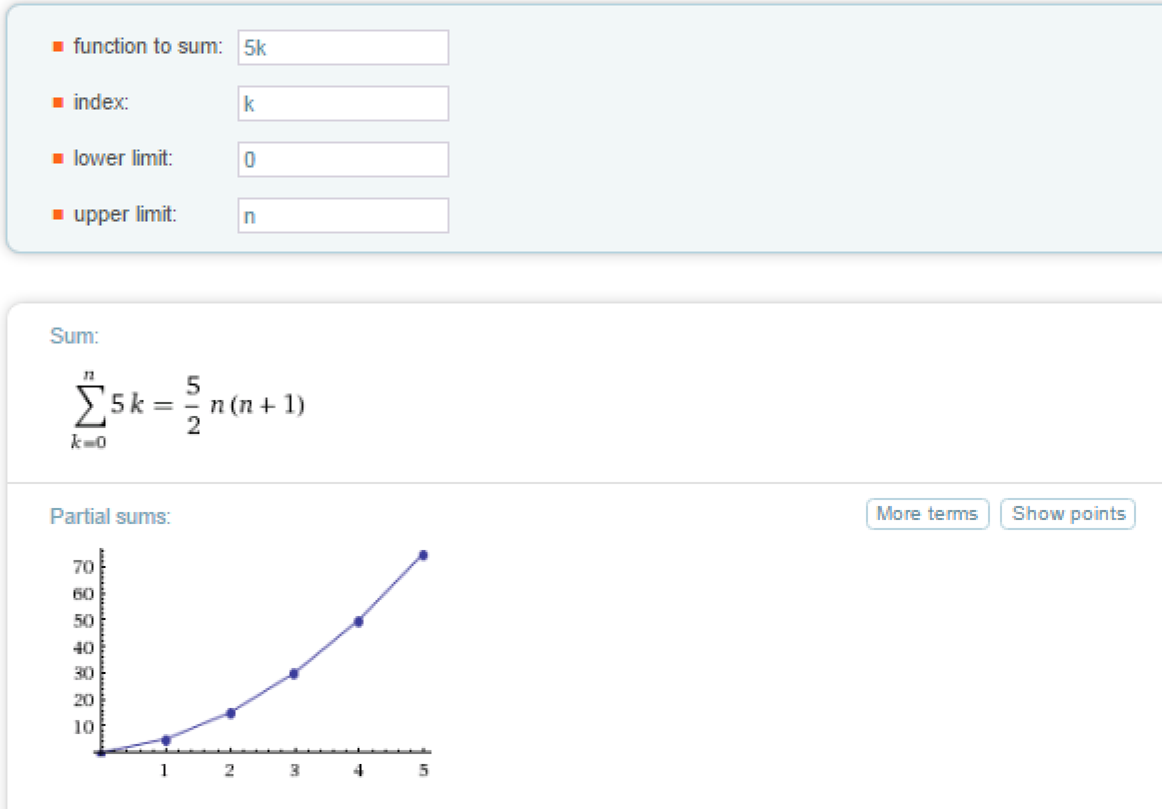
$$\sum_{i=0}^{\log n} 2^i = 2n - 1$$

$$\sum_{i=1}^{n} \log i = \log(n!)$$

$$\boxed{\frac{1}{2} n \log n \leq \log(n!) \leq n \log n \quad \therefore \quad \log(n!) = \Theta(n\ log\ n)}$$

Somatórios

Existem algumas ferramentas que calculam as fórmulas dos somatórios e até mesmo geram gráficos.



Ex: https://www.wolframalpha.com/input/?i=sum

UDESC

Propriedades das Potências

$$a^m a^n = a^{m+n}$$

$$\frac{a^m}{a^n} = a^{m-n}, a \neq 0$$

$$(ab)^n = a^n b^n$$

$$\left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}, b \neq 0$$

$$(a^m)^n = a^{nm}$$

UDESC

Propriedades dos Logaritmos

$$a^b = c \Leftrightarrow \log_a c = b$$

$$a^{\log_a b} = b$$

$$\log_c(ab) = \log_c a + \log_c b$$

$$\log_b a^c = c\log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$a^{\log_b c} = c^{\log_b a}$$

Atividade

Escreva duas versões do algoritmo de fibonacci: a versão “ruim” apresentada e uma versão “boa” usando vetores. Faça uma comparação de tempo de execução com valores entre 25 e 60.

Qual a complexidade da versão “boa” do algoritmo?

Atividade

Elabore um algoritmo recursivo para calcular a raiz quadrada de um número real (positivo) qualquer. Após implementar o algoritmo escreva a relação de recorrência e a respectiva complexidade do algoritmo para o pior caso.

*Lembrete: comparação de números reais através de aproximação => |x - 3.2| <= 0.00001

*OBS: $r = \sqrt{n}$ quando n >= 1, 1 ≤ r ≤ n

quando n < 1, n ≤ r ≤ 1

UDESC

Complexidade de Espaço, Notação Assintótica e Teorema Mestre

Complexidade de Espaço

Em alguns casos é importante considerarmos também a complexidade de espaço, i.e. o ‘espaço’ que é utilizado em função de ‘n’.

- Qual a complexidade do algoritmo abaixo?

```
int fatorial( int n ){
    if (n == 0)
        return 1;
    return n * fatorial( n-1 );
}
```

Complexidade de Espaço

- E qual a complexidade do algoritmo abaixo?

```
int fatorial( int n ){
   int a, b, c, d, e, f, g;
   if (n == 0)
      return 1;
   return n * fatorial( n-1 );
}
```

Complexidade de Espaço

- E deste algoritmo?

```
int pesqbin(int *v, int p, int r, int e){
   int q;
   if ( r < p )
       return -1;
   q = (p + r)/2;
   if (e == v[q])
       return q;
   if (e < v[q])
       return pesqbin(v, p, q-1, e);
   return pesqbin(v, q+1, r, e);
}
```

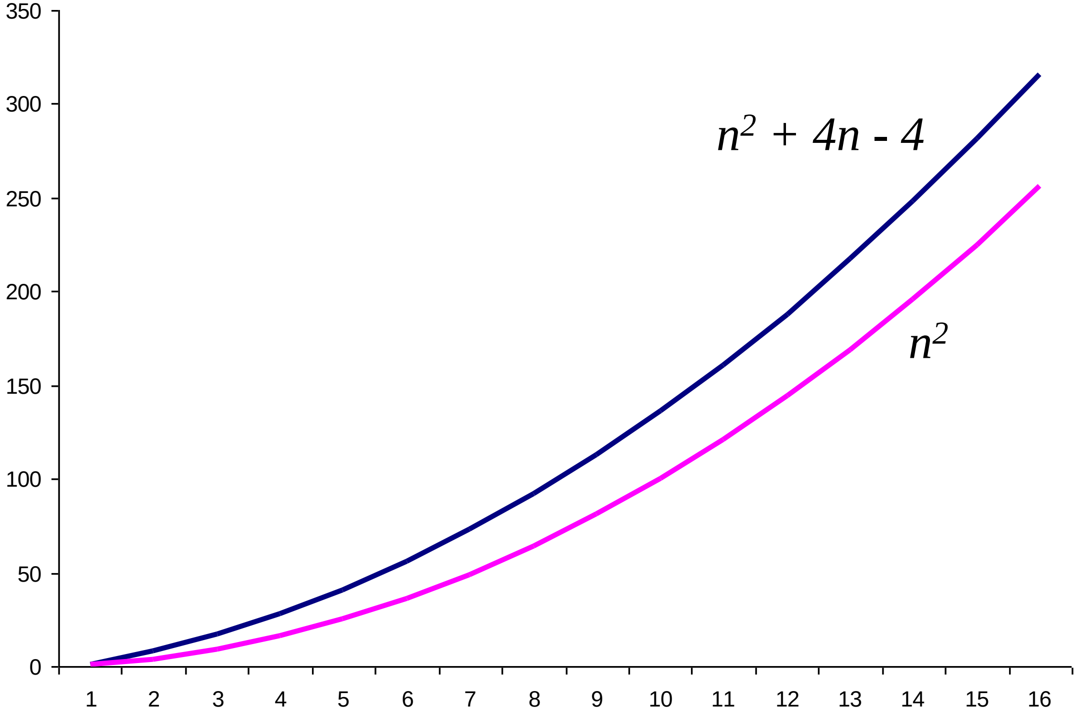
Complexidade de Espaço

- E deste outro?

```
/* Algoritmo muuuuito útil */
int foo( int n ){
    int v[n], i, s = 0;
    if ( n==0 ) return 0;
    for( i=0; i<n; i++ )
        v[i] = i*2;
    return  foo( n-1 );
}
```

Notação Assintótica
Limite Inferior (Notação Ω)

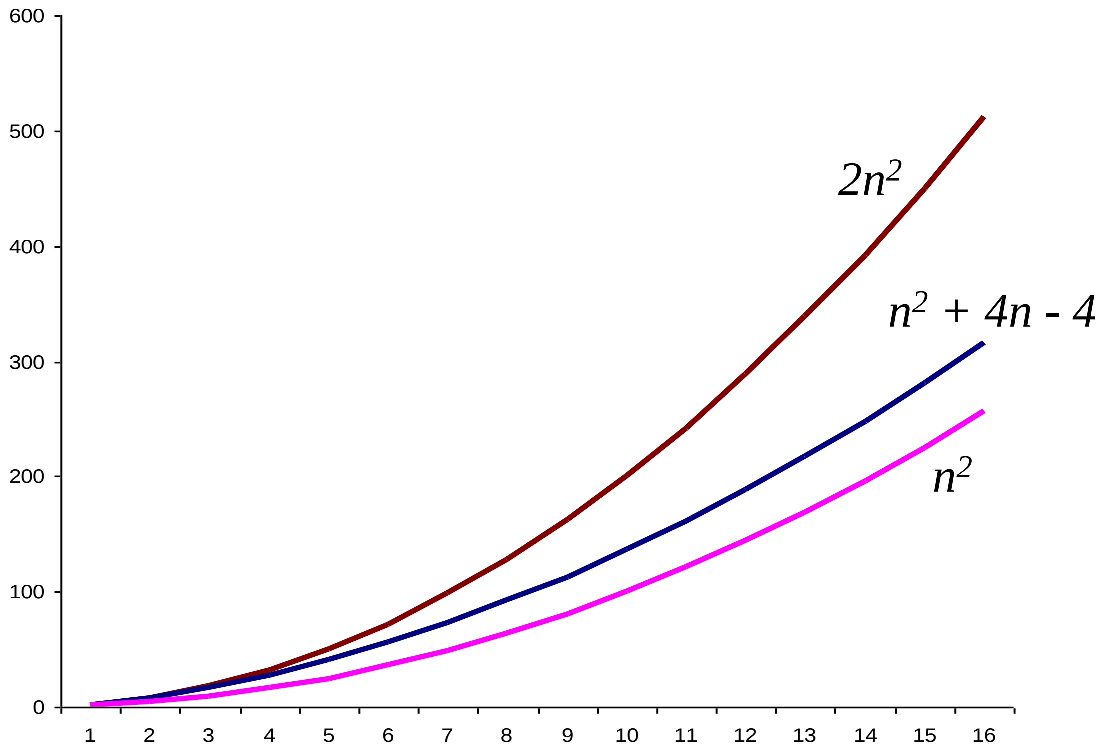
Uma função *f(n)* é o limite inferior de outra função *g(n)* se existem duas constantes positivas *c* e n_0 tais que, para $n > n_0$, temos $|g(n)| \geq c.|f(n)|$, $g(n) = \Omega(f(n))$.



$$n^2 + 4n - 4 = \Omega(n^2)$$

Notação Assintótica
Limite Firme (Notação Θ)

Uma função $f(n)$ é o limite restrito (ou exato) de outra função $g(n)$ se existem três constantes positivas c_1, c_2, e n_0 tais que, para $n > n_0$, temos $c_1.|f(n)| \geq |g(n)| \geq c_2.|f(n)|$, $g(n) = \Theta(f(n))$



[funções crescem com mesma rapidez]

$$n^2 + 4n - 4 = \Theta(n^2)$$

Notação Assintótica

Agora considere as funções f(x) = $n^{1-\epsilon}$ (onde $\epsilon > 0$ e $\epsilon < 1$) e g(x) = $\log n$.

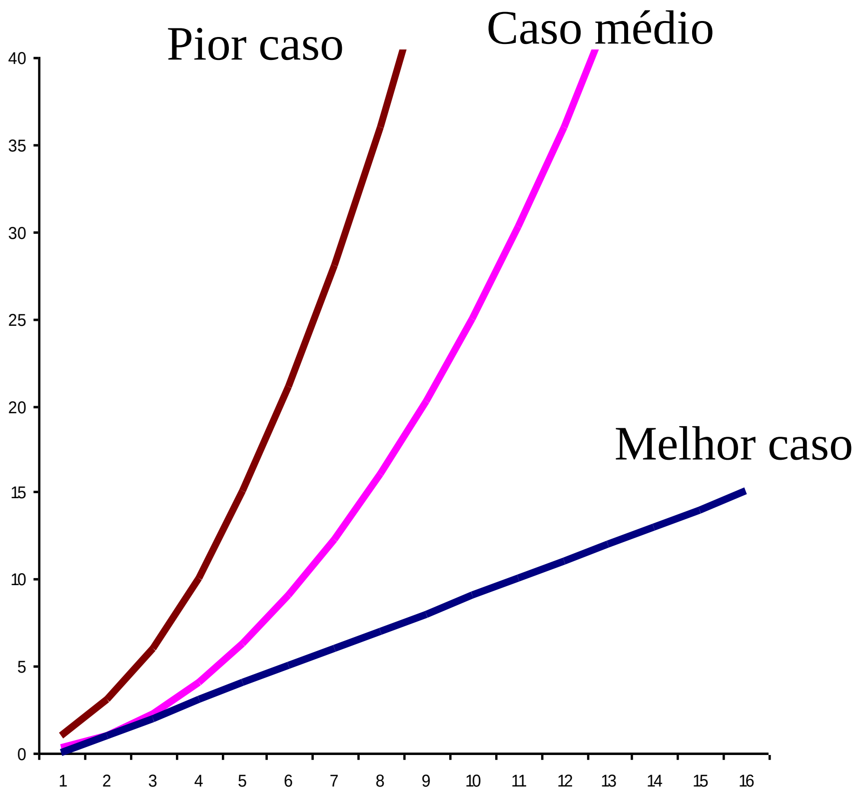
Qual a relação entre f(x) e g(x)?

$$f(x) = \mathrm{O}\ g(n)$$

$$f(x) = \Omega\ g(n)$$

$$f(x) = \Theta(g(n)$$

Ordenação por Inserção



Pior Caso: $\underline{O(n^2)}$

Caso Médio: $n^2/4$

Melhor Caso: $\underline{\Omega(n)}$.

Teorema Mestre

Livro de receitas para resolver recorrências da forma:

$$T(n) = aT(n / b) + f(n)$$

Sejam $a \geq 1$ e $b > 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definida sobre os inteiros não negativos pela recorrência acima, então $T(n)$ pode ser limitado assintoticamente como:

Teorema Mestre

Caso 1:

Se

$$f(n) = O\ (n^{\log_b a - \epsilon})$$

para alguma constante $\epsilon > 0$, então:

$$\boxed{T(n) = \Theta(n^{\log_b a})}$$

Exemplo: T(n) = 9T(n / 3) + n

UDESC

Teorema Mestre

Caso 2:

Se

$$f(n) = \Theta\,(n^{\log_b a})$$

então:

$$\boxed{T(n) = \Theta(n^{\log_b a} \times \log n)}$$

Exemplo: T(n) = T(2n / 3) + 1

UDESC

Teorema Mestre

Caso 3:

Se

$$f(n) = \Omega\left(n^{\log_b a+\epsilon}\right)$$

para alguma constante $\epsilon > 0$, e se
$af(n / b) \leq cf(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então:

$$\boxed{T(n) = \Theta(f(n))}$$

Exemplo: $T(n) = 3T(n / 4) + n \log n$

Teorema Mestre

Casos especiais:

Se as condições do caso 1, 2 ou 3 não forem satisfeitas então não é possível resolver a recorrência usando o teorema mestre!

Exemplo: $T(n) = 2T(n / 2) + n \log n$

Atividade

Considere o algoritmo abaixo e descreva a sua relação de recorrência, sua complexidade de tempo e espaço para o melhor caso e o pior caso:

```
double foo( int* v, int n, int p ){
    if( n<=0 )
        return 0;
    int i, soma = 0;
    for( int i=0; i<n ; i=i+p )
        soma += v[i];
    return sqrt(soma) + foo( v, soma%n, p );
}
```

Atividade

Considere o algoritmo abaixo e descreva a sua relação de recorrência, sua complexidade de tempo e espaço para o melhor e pior caso:

```
int pow(int base, int exp){
    if( exp==0 ) return 1;
    int ret = pow( base, exp/2 );
    ret = ret * ret;
    if( exp%2 == 1 ) ret = ret * base;
    return ret;
}
```

Atividade

Use o método mestre para fornecer limites assintóticos restritos para as seguintes recorrências:

a. $T(n) = T(n/2) + \Theta(1)$

b. $T(n) = 4T(n/2) + \Theta(n)$

c. $T(n) = 4T(n/2) + \Theta(n^3)$

Resolva as seguintes recorrência por substituição:

d. $T(n) = T(n-1) + \Theta(\log n)$

$T(1) = \Theta(\log 1)$

e. $T(n) = T(\sqrt{n}) + \Theta(1)$

$T(2) = \Theta(1)$

Algoritmos Eficientes de Ordenação: Merge Sort, Quick Sort e Heap Sort

Dividir e Conquistar

Desmembrar o problema original em vários subproblemas semelhantes, resolver os subproblemas (executando o mesmo processo recursivamente) e combinar as soluções.

Merge Sort

A principal ideia do Merge Sort é ordenar partições do vetor e então reordenar o conjunto através da operação **merge**, uma mesclagem ordenada de dois vetores [*O(n)*].

O algoritmo de Merge Sort necessita de um espaço adicional de memória para trabalhar [*O(n)*].

2 3 7 9 10

1 4 5 6 8

Merge Sort

6 5 3 1 8 7 2 4

Merge Sort

Qual a complexidade de tempo do Merge Sort?

Existe um pior caso? Qual seria?

Qual a complexidade de espaço do Merge Sort?

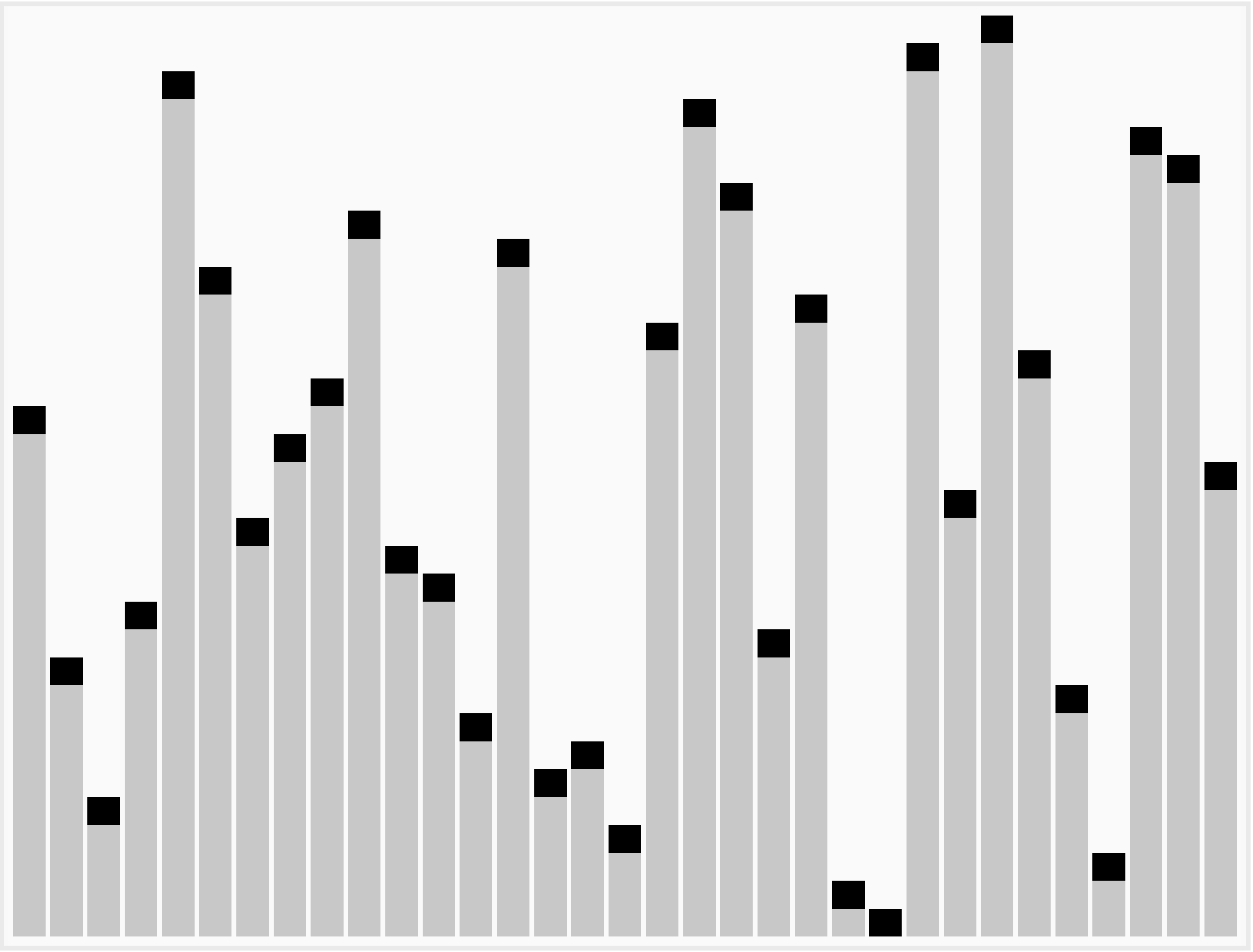
Quick Sort

A principal ideia do Quick Sort é a ordenação com base em um elemento denominado **pivô**.

Deve-se ordenar o vetor mantendo todos os elementos menores do que o pivô a sua esquerda e todos os elementos maiores do que o pivô a sua direita.

Após este processo realiza-se o mesmo procedimento para o grupo a esquerda do pivô e depois para o grupo a direita do pivô – enquanto o número de elementos for maior do que um.

Quick Sort



Quick Sort

Qualquer pivô serve?

O que pode acontecer se escolhermos um pivô ruim?

Como escolher um pivô adequado?

- Aleatoriamente
- Escolher no máximo um pivô “menos pior”

Qual o pior caso para o Quick Sort?

Quick Sort

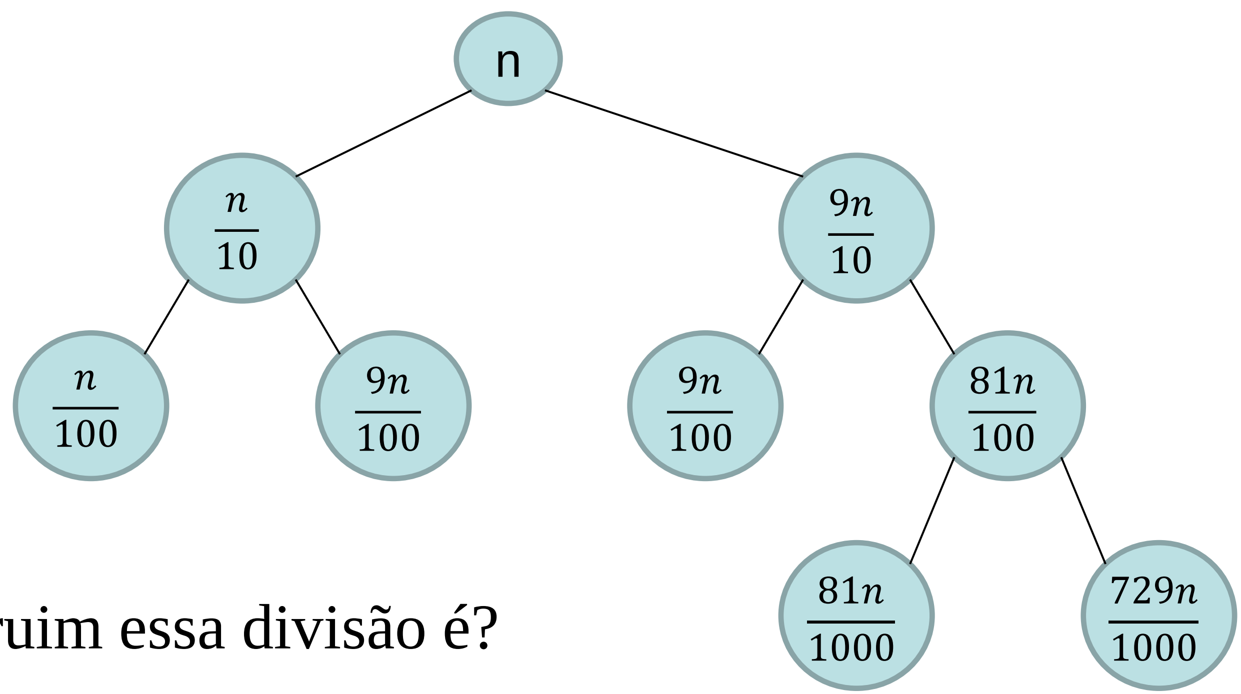
Qual a complexidade de tempo do Quick Sort para o pior caso /cenário?

Qual a complexidade de tempo esperada / média do Quick Sort?

Qual a complexidade de espaço do Quick Sort?

Quick Sort

Considere o seguinte cenário onde ocorre uma divisão desbalanceada de proporção constante 1:9



Quão ruim essa divisão é?

Quick Sort

Quão ruim essa divisão é?

$$T(n) = T(n/10) + T(9n/10) + n$$

$$T(1) = 1$$

Heap Binário

É um arranjo, onde os dados estão organizados de forma que podem ser acessados como se estivessem armazenados em uma árvore binária.

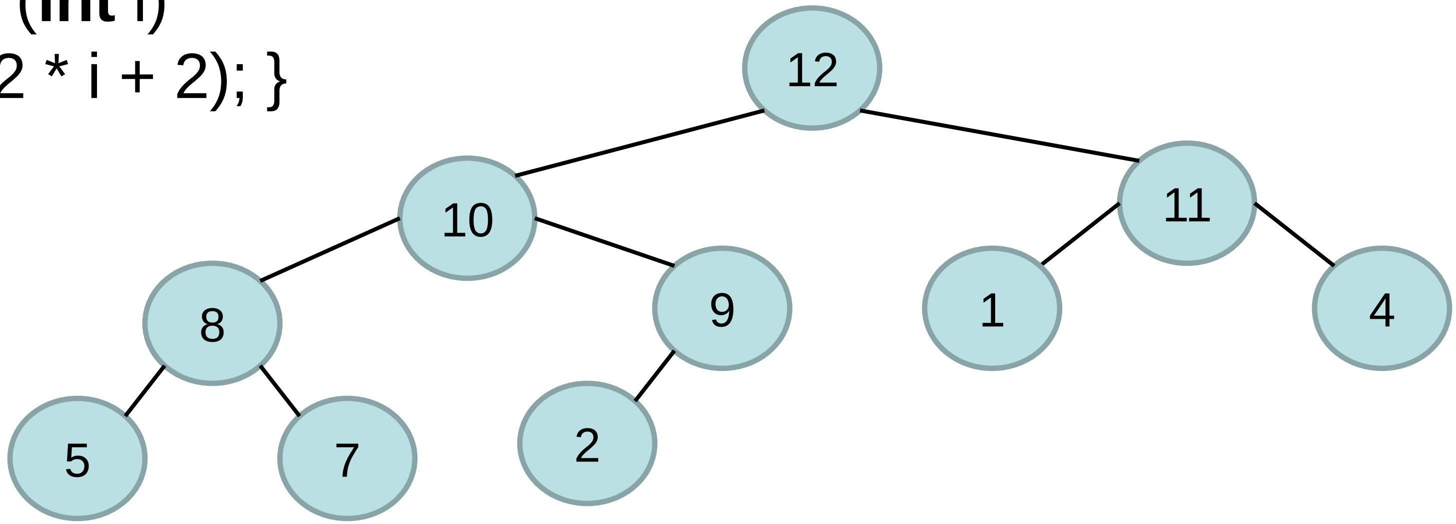
No caso de um ***heap máximo***, os elementos armazenado em uma sub-árvore serão sempre menores que o elemento armazenado na raiz. Essa árvore é completa todos seus níveis, com a possível exceção do nível mais baixo.

Heap Binário

0	1	2	3	4	5	6	7	8	9
12	10	11	8	9	1	4	5	7	2

int esquerda (**int** i)
{ **return** (2 * i + 1); }

int direita (**int** i)
{ **return** (2 * i + 2); }

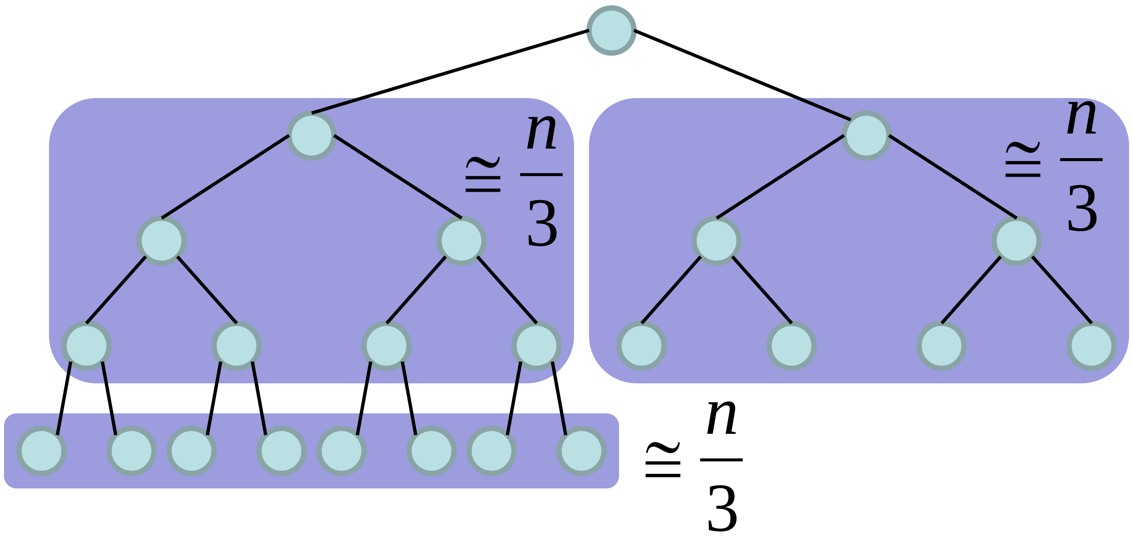


Heap Binário

```
// n = tamanho // i = índice
void heapify (int *a, int n, int i) {
        e = esquerda( i );
        d = direita( i );
        if (e < n  &&  a[e] > a[i])
                maior = e;
        else    maior = i;

        if (d < n  &&  a[d] > a[maior])
                maior = d;
        if ( maior != i ) {
                swap (&a[i], &a[maior]);
                heapify(a, n, maior);
        }
}
```

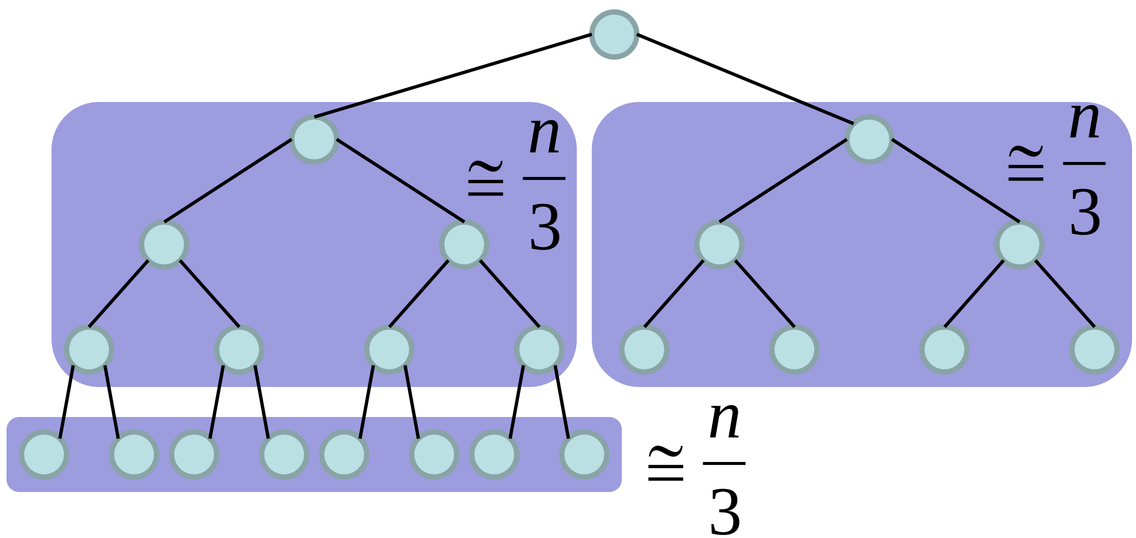
Heap Binário



$$T(n) = T(2n/3) + O(1)$$

$$T(1) = O(1)$$

Heap Binário



Note ainda que existem no máximo: $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nós de altura h

n=23 => h=0 : 12 (folha)

h=1 : 6

h=2 : 3

h=3 : 2

Obs: altura de baixo p/ cima devido ao heapify que também é de baixo p/ cima

Heap Binário

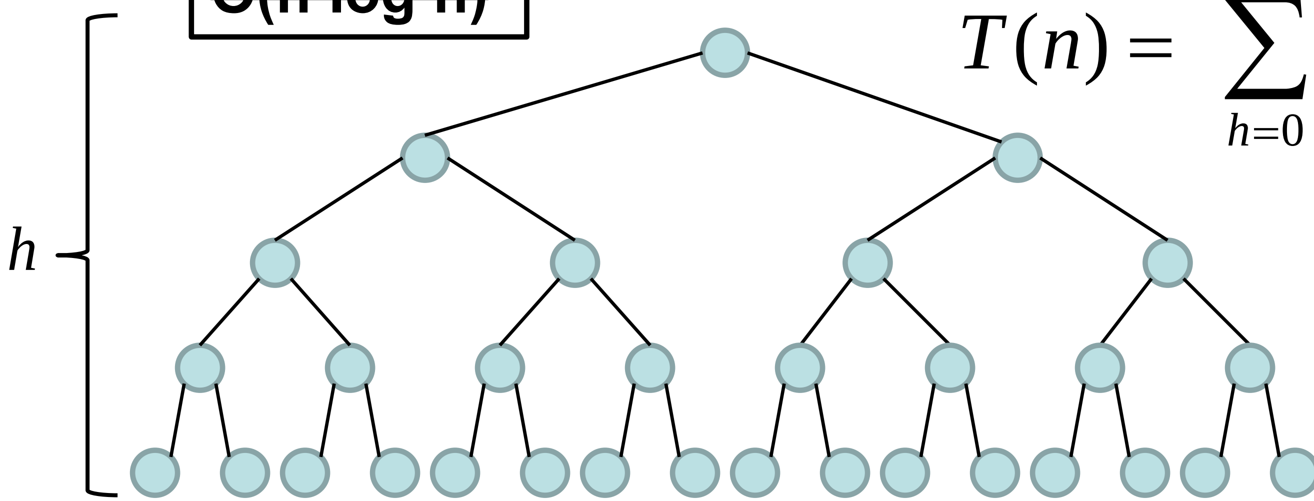
```
void  buildHeap ( int *a, int n ) {
    int i;
    for (i = n/2; i >= 0; i--)      //O(n)
        heapify(a, n, i);           //O(log n)
}
```

O(n log n)

~~**Θ(n log n)**~~

heapify varia com a altura do nó!

$$T(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} [(n/2^{h+1}) O(h)]$$



Θ(n)

Heap Binário

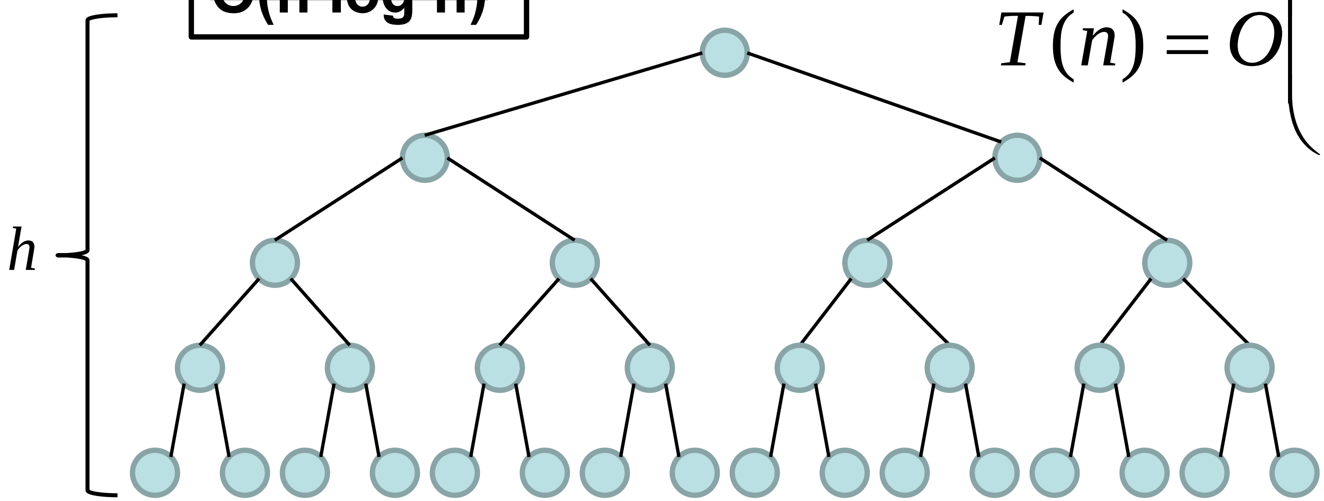
```c
void  buildHeap ( int *a, int n ) {
   int i;
   for (i = n/2; i >= 0; i--)      //O(n)
      heapify(a, n, i);            //O(log n)
}
```

O(n log n)

~~Θ(n log n)~~

heapify varia com a altura do nó!

$$T(n) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} (h/2^h) \right)$$



Θ(n)

Heap Sort

```
void heapSort( int *a, int n) {
    buildHeap( a , n );                  // O(n)
    for (i=n-1; i>0; i--) {              // O(n)
        swap(&a[0], &a[i]);              // O(1)
        heapify(a, i, 0);                // O(log n)
    }
}
```

O(n log n)

Ordenação em tempo Linear
Counting Sort + Bucket Sort

Ordenações Lineares

Métodos de ordenação por comparação:

$$\Omega(n \log n)$$

Ordenações lineares [$\Omega(n)$] só são possíveis em **determinadas** condições.

Counting Sort

Pressupõe valores inteiros no intervalo 1 a k.

Algoritmo:

- contar o nº de elementos menores que ‘x’;
- usar esta informação para alocar o elemento na sua posição correta no vetor final;

Exemplos:

Caso bom	2	5	3	0	2	3	0	5	3	0

Caso ruim	2	25	13	100	250	300.000	1	5	3	0

Bucket Sort

Pressupõe que a entrada consiste de elementos com distribuição de valores uniforme.

Algoritmo:

- separar os elementos em grupos / baldes;
- ordenar os elementos nos seus baldes;

Exemplos:

Caso bom	20	16	43	0	22	13	33	40	31	7
Caso ruim	2	43	3	5	0	7	8	1	6	4

Trabalho 1

Escreva os algoritmo **Bubble Sort**, **Insert Sort**, **Merge Sort, Quick Sort**, **Heap Sort, Counting Sort e Bucket Sort**. O algoritmo Quick Sort deve ser implementado de duas maneiras: (1) usando o primeiro elemento como pivô, (2) selecionando um elemento qualquer como pivô.

Teste os algoritmos com vetores de 25.000, 50.000, 75.000, 100.000 e 1.000.000 números entre 0 e o tamanho do vetor: (1) em ordem crescente, (2) em ordem decrescente, (3) em ordem aleatória e (4) em ordem aleatória mas com um elemento 100.000.000 (em qualquer posição).

Escreva um relatório comparativo apresentando os tempos de execuções de cada algoritmo para cada tipo de entrada, apresentando um gráfico para comparação dos tempos.

Trabalho 1

O relatório deve seguir o padrão de artigos da SBC e deve apresentar apenas a análise comparativa dos algoritmos. Não é necessário apresentar os mecanismos de funcionamento teóricos dos algoritmos, mas deve-se explicar como os algoritmos foram implementados.

Dê maior ênfase para a comparação de **tempo** e **espaço** entre os algoritmos, destacando o comportamento para cada tipo de entrada.

O trabalho poderá ser realizado em **duplas**.

Análise de complexidade de Estruturas de Dados Fundamentais

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Fila tem um ponteiro para o início e o fim!

```
Fila::enfileirar( elemento );
Fila::remover( );
Fila::busca( elemento );
Fila::tamanho( );
```

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

- OBS: Lista Encadeada Simples => elemento + ponteiro p/ próximo

```
ListaSimples::adicionarNoInicio( elemento );
ListaSimples::adicionarNoFim( elemento );
ListaSimples::busca( elemento );
ListaSimples::anterior( *elemento );
ListaSimples::proximo( *elemento );
```

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Lista Dup. Enc. => elemento + ponteiro p/ anterior e próximo

```
ListaDup::adicionarNoInicio( elemento );
ListaDup::adicionarNoFim( elemento );
ListaDup::busca( elemento );
ListaDup::anterior( *elemento );
ListaDup::proximo( *elemento );
```

Estruturas de Dados Básicas

Pilha, Fila, Lista Encadeada Simples, Lista Duplamente Encadeada, Árvore Binária, Árvore Rubro Negra

Analisar os seguintes métodos:

* OBS: Árvore => elemento + ponteiro para filhos

Arvore::inserir(elemento);

Arvore::buscar(elemento);

Arvore::pai(*no);

Arvore::maximo( );

** E se cada nó tiver um ponteiro para o pai?

Atividade

Analise o código fonte do arquivo estruturas-dados.cpp e informe as complexidades de tempo e espaço dos seguintes métodos considerando o **pior** e **melhor** caso. Ilustre um exemplo de situação onde o **pior** caso e o **melhor** caso ocorreria para cada uma das funções.

[Faça em uma folha para entregar] [no máximo **duplas**]

Pilha	ListaEncadeada	ÁrvoreVP
Empilha	Adicionar	rotateLeft
Desempilha	Anexar	Inserir
Limpar	Remover	Busca
Print	Encontrar	PreOrder
Tamanho	RemoverDuplicatas	Maximo

Atividade

Resolva os seguintes problemas do URI online Judge e determine a complexidade para o pior caso:

- 1861 – O Hall dos Assassinos

  (http://www.urionlinejudge.com.br/repository/UOJ_1861.html)

OBS: Listagem de resultados:

Accepted – sua solução foi aceita;

Wrong Answer – o algoritmo produziu uma resposta incorreta para alguma situação;

Time Limit Exceded – seu algoritmo entrou em loop infinito ou demorou muito (rever algoritmo e reduzir a sua complexidade de tempo);

Runtime Error – seu programa fez alguma operação indevida (divisão por zero ou acesso a memória inválida);

Hash

Hash - Introdução

Uso de vetores para armazenar elementos é bom.

- Qual a complexidade para acessar um elemento?
- Qual a complexidade para adicionar um elemento?

Limitações dos vetores: os índices só podem ser inteiros

P: Alguma sugestão para podermos usar outros valores (de qualquer tipo) como índices?

Hash - Introdução

Podemos criar uma operação de conversão de um tipo qualquer para um valor inteiro positivo = **FUNÇÃO HASHING**

(chave) => índice

Como poderíamos criar uma função para converter uma string em um inteiro positivo?

1 – Somar o código ascii de cada letra

2 – Somar o código ascii de cada letra * posição

3 – Somatório(código ascii de cada letra ^ posição)

Testes: "Joao", "Maria", "ola", "alo", "oi", "io"

Hash - Introdução

Na conversão precisamos nos preocupar com mais um detalhe:

Qual é o tamanho do nosso vetor?

Se tivermos um vetor de **10 posições**, qual operação podemos usar para garantir que o nosso inteiro positivo esteja entre 0 e 9?

Fator de carga do Hash: $\lambda = \frac{n^{\underline{o}}\ elementos}{tamanho\ vetor}$

Hash - Introdução

Quando usamos uma função hashing que realiza um mapeamento único para cada índice do vetor temos um **hash perfeito**

Se os nomes só podem ter <u>no máximo 5 letras</u>, qual tamanho de vetor podemos utilizar para garantir um hash perfeito?

(note que um hash perfeito depende da função de hashing)

Hash - Introdução

Quando usamos uma função hashing que não realiza um mapeamento único para cada índice do vetor Precisamos considerar também que a função de hashing pode gerar **números iguais** para **chaves diferentes**:

Hashing (2) + vetor de 10 posições: "Maria", "Ana"

A esse evento chamamos de **COLISÃO**!

Nem sempre é viável utilizar um hash perfeito. Logo, uma função hashing deve sempre buscar **minimizar** o número de colisões

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

Abordagem 1: Não adicionamos o registro

Abordagem 2 – endereçamento aberto:

Procurar pela próxima posição vaga

Sondagem Linear: avança de um em um

Método deve ser usado para inserção e busca

exemplo: busca pelo 55 (hash: 0); busca pelo 66 (hash: 1)

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

<u>Abordagem 2 – endereçamento aberto</u>:

Sondagem Linear pode gerar agrupamentos (*clusters*) de elementos (não interessante)

<u>Sondagem Quadrádica</u>: avança usando os quadrados

x+1, x+4, x+9, x+16, x+25, (...)

Método deve ser usado para inserção e busca

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

UDESC

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

<u>Abordagem 2 – endereçamento aberto</u>:

<u>Hash duplo (rehash)</u>: aplicação de uma nova função hash

$$novo\ hash = rehash(\ hash\ antigo\ )$$

$$rehash(x) = (x + 1)\ \%\ tamanho\ vetor$$

rehash deve garantir que, eventualmente, todas as posições serão avaliadas

Método deve ser usado para inserção e busca

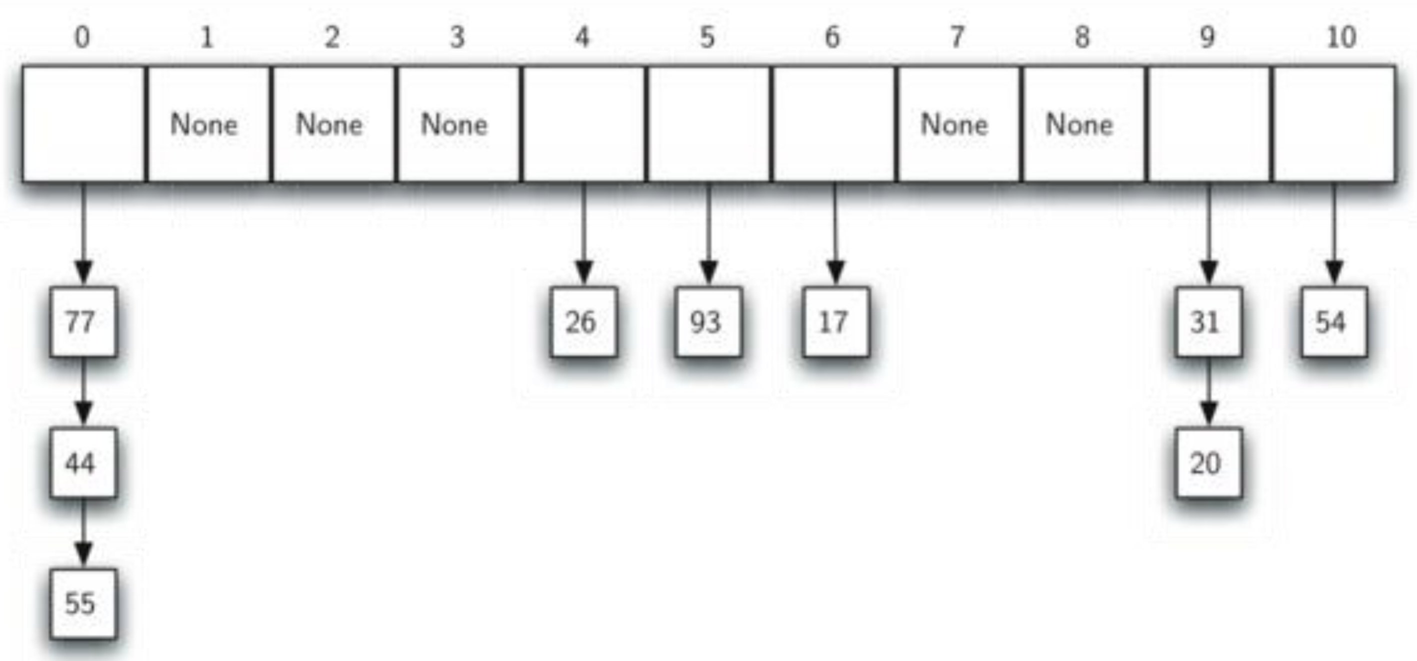
0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17	None	None	31	54

Hash - Colisões

Ok, Mas o que fazer quando ocorre uma colisão?

<u>Abordagem 3 – encadeamento</u>:

Ao invés de utilizarmos um vetor simples utilizamos um vetor de alguma estrutura de dados (lista, árvore)



Hash - Introdução

Hash - método da divisão:

Escolhemos um valor *m* como divisor de modo a evitar colisões (um número que não seja potência de 2 => 2^p)

Um bom valor pode ser um número primo ou então 2^p-1

Podemos realizar duas divisões: (valor % m) % tam

Hash – método da multiplicação:

Pesquisar =)

Hash – Principais funções

Hash(tamanho)

put(chave, valor) => boolean

get(chave) => valor

listar( ) => (chaves, valores)

Hash – Analise Complexidade

Considerando um hash de strings com capacidade ***n***, e tamanho de strings ***m***:

- Qual a complexidade para calcular a função hash (1), (2) e (3)?

- Qual a complexidade (melhor e pior caso) para inserir e buscar um elemento usando sondagem linear? E para listar todos os elementos do hash?

Hash – Analise Complexidade

Considerando um hash de strings com capacidade ***n***, e tamanho de strings ***m***:

- Qual a complexidade (melhor e pior caso) para inserir e buscar um elemento usando hash encadeado com lista? E para listar todos os elementos do hash?

- Qual a complexidade (melhor e pior caso) para inserir e buscar um elemento usando hash encadeado com árvore balanceada (árvore rubro-negra)?

Atividade

Comparação entre o uso de busca linear, busca binária e Hash

Escreva um programa que leia um número inteiro **`n`** e em seguida leia **`n`** pares **`p s`**, onde **`s`** indica o nome de um personagem fictício (nome e sobrenome) e **`p`** indica um número de ordem associado a esse personagem. Depois, leia um número inteiro **`m`** e mais **`m`** nomes (nome e sobrenome). Para cada um desses nomes, mostre o número de ordem associado ao personagem.

Escreva três versões para este programa e as compare:

1) Use um vetor ou lista e realize buscas lineares
2) Use um vetor ou árvore e realize buscas binárias
3) Use uma estrutura de Hash encadeado com lista ou árvores

Atividade

Comparação entre o uso de busca linear, busca binária e Hash

Os casos de teste e as respostas se encontram na página da disciplina: exercício-hash.zip. Trata-se de cinco casos de teste:

1 – 5.000 nomes / 1.000 consultas

2 – 25.000 nomes / 10.000 consultas

3 – 50.000 nomes / 10.000 consultas

4 – 50.000 nomes / 25.000 consultas

5 – 100.000 nomes / 75.000 consultas

Compare as suas respostas com os gabaritos fornecidos (saida-n.txt)

UDESC

Algoritmos de Grafos – Complexidade com múltiplas variáveis

Grafos

Grafos possuem dois elementos principais:

- Vértices – Vertex (V)
- Arestas – Edges (E)

Grafos costumam ser representados de duas formas, que utilizam uma quantidade diferente de **espaço**:

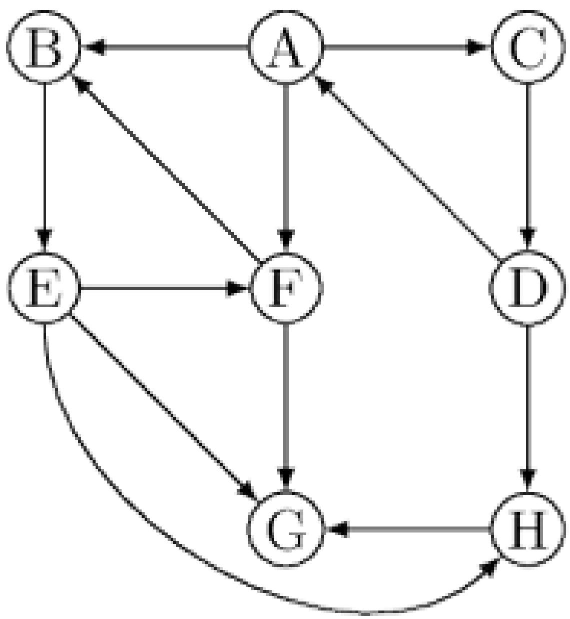
- Matriz de Adjacência $O(V^2)$
- Lista de Adjacência $O(V+E)$

*Este espaço de representação pode ou não ser considerado na análise de complexidade de espaço.

Depth First-Search

(*Busca em Profundidade*)

O **algoritmo de Busca em Profundidade** realiza uma busca ou travessia em um grafo. É utilizado para verificar quais vértices são acessíveis iniciando um caminho a partir de um vértice específico

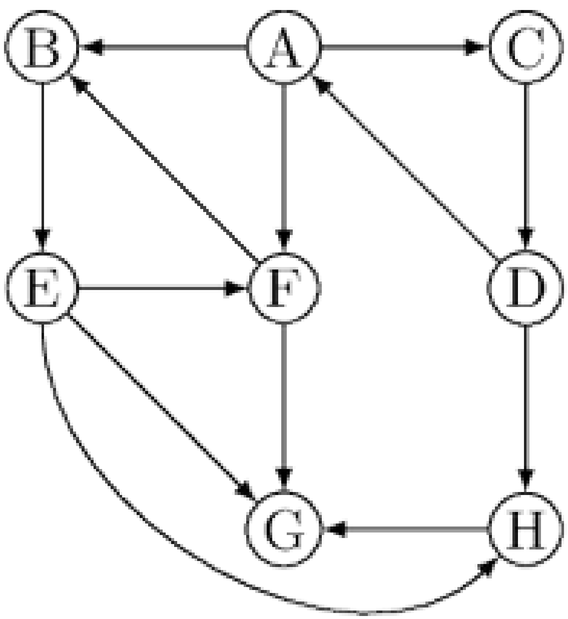


Quais vértices são acessíveis partindo de A?

Quais vértices são acessíveis partindo de E?

Depth First-Search
(*Busca em Profundidade*)

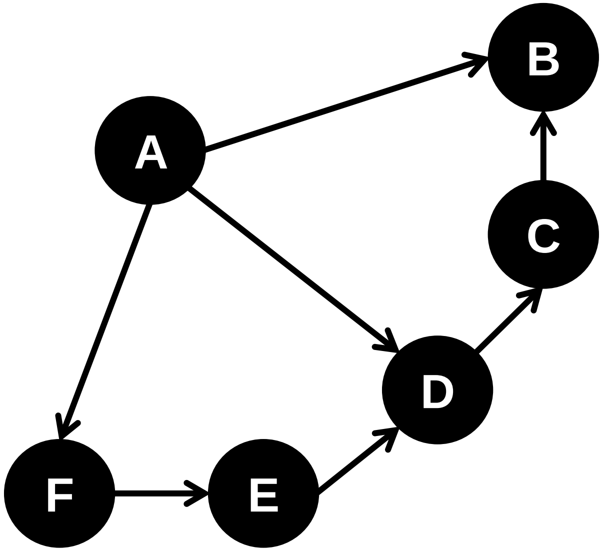
O **algoritmo de Busca em Profundidade** realiza uma busca ou travessia em um grafo. É utilizado para verificar quais vértices são acessíveis iniciando um caminho a partir de um vértice específico




```
// G = grafo, s = vértice inicial,
// v = vetor de visitação de vértices
DFS( G, s, v )
  v[s] = true
  para cada u ϵ G.adj[s]
    if( v[u] == false )
      DFS( G, u, v )
```

Depth First-Search

(*Busca em Profundidade*)



Complexidade de tempo:

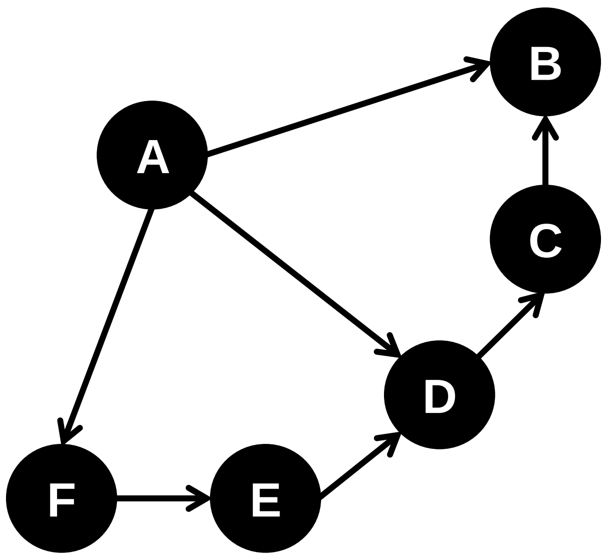
Matriz

Melhor caso:

Pior caso:

Complexidade de espaço:

Depth First-Search
(*Busca em Profundidade*)



<u>Complexidade de tempo</u>:

Matriz

Melhor caso: O(V)

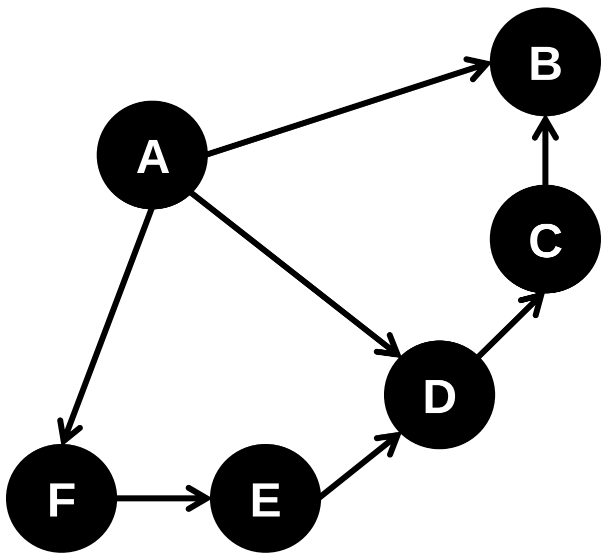
Nenhum nó está acessível a partir de um determinado nó

Pior caso: O(V^2)

Todos os outros nós estão acessíveis a partir de um nó

<u>Complexidade de espaço</u>: precisamos manter o vetor de nós visitados **O(V)**

Depth First-Search
(*Busca em Profundidade*)



Complexidade de tempo:

Lista

Melhor caso: O(V)

Necessidade de inicializar vértices como não visitados

Pior caso: O(V + E)

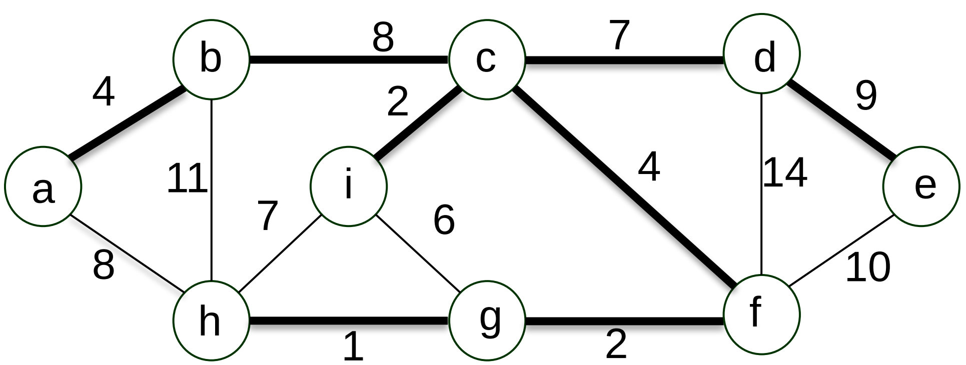
Todos os outros nós estão acessíveis a partir de um nó

Complexidade de espaço: precisamos manter o vetor de nós visitados **O(V)**

Prim

(*Minimum Spanning Tree*)

Uma **árvore geradora** para um grafo conexo é uma árvore que conecta todos os vértices do grafo e que o conjunto de arestas é um subconjunto das arestas desse grafo. A árvore geradora é mínima se o somatório dos custos associados cada arestas for o menor possível.

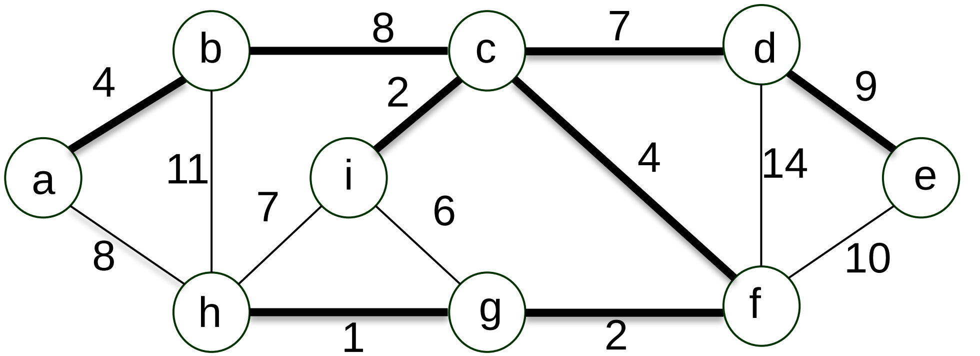


Prim

(Minimum Spanning Tree)

```
// G = grafo, s = vértice inicial, w = vetor de custo
MST-PRIM( G, s, w )
  para cada u ∈ G.V
    u.custo = w[u] = ∞
    u.pai = null
s.custo = w[s] = 0
q.add( {0,s} )

enquanto q ≠ ∅                        // |V| vezes
  u = extrai-minimo( q )             // Qual a estrutura usada??
  para cada v ∈ G.adj[u]             // Qual a estrutura usada??
    se v.custo < w[v]
      q.add( {v.custo, v} )          // Qual a estrutura usada??
      w[v] = v.custo
      v.pai = u
```



Prim

(Minimum Spanning Tree)

```
// G = grafo, s = vértice inicial, w = vetor de custo
MST-PRIM( G, s, w )
  para cada u ϵ G.V
    u.custo = w[u] = ∞
    u.pai = null
s.custo = w[s] = 0
q.add( {0,s} )

enquanto q ≠ ∅                  // |V| vezes
  u = extrai-minimo( q )        // |V| verificações
  para cada v ϵ G.adj[u]        // |V| vezes
    se v.custo < w[v]
      q.add( {v.custo, v} )     // O(1)
      w[v] = v.custo
      v.pai = u
```

Usando G=>Matriz e q=>Lista

Tempo:

$O(|V^2| + |V^2|)$

$O(|V^2|)$

Prim

(Minimum Spanning Tree)

```
// G = grafo, s = vértice inicial, w = vetor de custo
MST-PRIM( G, s, w )
  para cada u ∈ G.V
    u.custo = w[u] = ∞
    u.pai = null
s.custo = w[s] = 0
q.add( {0,s} )

enquanto q ≠ ∅                    // * { O(V) }
  u = extrai-minimo( q )          // * => O(V²) verificações
  para cada v ∈ G.adj[u]          // * => O(E)
    se v.custo < w[v]
      q.add( {v.custo, v} )       // O(1)
      w[v] = v.custo
      v.pai = u
```

Usando G=>Lista e q=>Lista

<u>Tempo</u>:

$O(|V^2| + |E|)$

Prim

(Minimum Spanning Tree)

```
// G = grafo, s = vértice inicial, w = vetor de custo
MST-PRIM( G, s, w )
  para cada u ∊ G.V
    u.custo = w[u] = ∞
    u.pai = null
s.custo = w[s] = 0
q.add( {0,s} )

enquanto q ≠ ∅                        // * { O(V) }
  u = extrai-minimo( q )              // * => O( V log V )
  para cada v ∊ G.adj[u]              // * => O( E )
    se v.custo < w[v]
      q.add( {v.custo, v} )           // (log V) [Se não gerar
      w[v] = v.custo                               duplicatas]
      v.pai = u
```

Usando G=>Lista e q=>Árvore

Tempo:

O(|V log V + E log V|)

O((V+E) log V)

Prim

(Minimum Spanning Tree)

```
// G = grafo, s = vértice inicial, w = vetor de custo
MST-PRIM( G, s, w )
  para cada u ∈ G.V
    u.custo = w[u] = ∞
    u.pai = null
s.custo = w[s] = 0
q.add( {0,s} )

enquanto q ≠ ∅                     // * { O(V) }
  u = extrai-minimo( q )           // * => O( V log V )
  para cada v ∈ G.adj[u]           // * => O( E )
    se v.custo < w[v]
      q.add( {v.custo, v} )        // (1)
      w[v] = v.custo
      v.pai = u
```

Usando G=>Lista e q=>Fib. Heap

Tempo:

O(E + V log V)

Atividade

Realizar a análise de complexidade de **tempo** e **espaço** dos seguintes algoritmos considerando uma implementação usando (a) matrizes e (b) listas de adjacências. Uma dupla será sorteada para apresentar cada algoritmo (15 minutos), na qual relembrará brevemente o algoritmo e ilustrará uma situação de pior caso destacando a complexidade do algoritmo. As notas de apresentação serão compartilhadas pela turma. Contudo, uma outra equipe pode se prontificar a apresentar de modo a melhorar a sua nota e aumentar a média da turma (equipe que apresentou fica com a nota inicial).

+ Dijkstra
+ BellmanFord
+ FloydWarshall
+ Ordenação Topológica

Algoritmos com Inteiros

Algoritmos com Inteiros

Até esse momento temos considerado constante a complexidade de operações como: adição, subtração, multiplicação, divisão, módulo e comparação.

Mas quando essas operações envolvem números cujo o tamanho, em número de bits, é muito maior que a palavra do processador (atualmente 32 ou 64 bits)?

Algoritmos com Inteiros - Soma

Um primeiro modelo de soma entre números inteiros não limitados a palavra do processador poderia ser similar ao método que utilizamos para somar números decimais:

Processador de 1 bit

n1 (110)				1	1	0	1	1	1	0
n2 (379)		1	0	1	1	1	1	0	1	1
soma										

Qual a complexidade de tempo para realizar esta soma?

Algoritmos com Inteiros - Soma

Contudo, um processador consegue trabalhar com palavras de tamanho *n* – ou seja – em apenas uma única operação ele soma dois números que tenham no máximo *n* bits

Processador de 8 bits

Carry:	11 1	
	1111010	(122)
+	1100011	(99)
	11011101	(221)

Qual a complexidade de tempo para realizar esta soma?

Algoritmos com Inteiros - Soma

Se o número for maior do que o tamanho do processador precisamos somente de uma memória auxiliar para o carry

Processador de 4 bits

Carry:		1		
		111	1010	(122)
	+	110	1011	(107)
		1110	0101	(229)

Qual a complexidade de tempo para realizar esta soma?

Algoritmos com Inteiros - Soma

De forma genérica

Adição de 2*n* bits, onde *n* é o tamanho da palavra do processador:

Carry

$\boxed{b_{2n}b_{2n-1} \;\; \ldots \;\; b_{n+2}b_{n+1}}$ $\boxed{b_{n}b_{n-1} \;\; \ldots \;\; b_{2}b_{1}}$

Assim, quando o número de bits (*k*) do número for próximo ao tamanho da palavra do processador teremos uma complexidade de tempo constante O(1)

Algoritmos com Inteiros - Soma

E quando o tamanho do número (*k*) for significativamente superior à palavra do processador, ou quando **k for variável**?

$n = 32$ / $k = 64$ ➔ serão necessárias 2 operações

$n = 32$ / $k = 480$ ➔ serão necessárias 15 operações

$n = 32$ / $k = 4992$ ➔ serão necessárias 156 operações

$$O(\, k \,/\, n \,) \text{ onde n é uma constante}$$

Assim, podemos considerar que a adição de grandes números tem complexidade $O(k)$, onde k é o número de bits do maior número.

Algoritmos com Inteiros (Multiplicação)

Um primeiro modelo de multiplicação seria realizar somas sucessivas de um valor:

x = 8

y = 5

$$\underbrace{8 + 8 + 8 + 8 + 8}_{\text{y vezes}} \quad = \quad 40$$

Algoritmos com Inteiros (Multiplicação)

```
bigInt mul (bigInt x, bigInt y) {
    bigInt r =  0;
    while (y > 0){    // Comparação entre inteiros grandes
        r = r + x;  // Adição entre inteiros grandes O(k).
        y--;
    }
    return r;
}
```

$$\underbrace{O(\mathrm{k}) + O(\mathrm{k}) + \ldots + O(k)}_{y \text{ vezes}}$$

Sendo *k* o número de bits de x,
qual a complexidade de tempo para este algoritmo?

Algoritmos com Inteiros (Multiplicação)

$$\underbrace{O(\mathrm{k}) + O(\mathrm{k}) + \ldots + O(k)}_{y \text{ vezes}}$$

Sendo *k* o número de bits de x,
qual a complexidade de tempo para este algoritmo?

O(y * k)

Se considerarmos que y tem k bits também, podemos dizer que
$y = 2^k$

Logo: $O(2^k * k) = \mathbf{O(2^k)}$

Algoritmos com Inteiros (Multiplicação)

Um segundo modelo seria similar ao método que utilizamos para multiplicar números na base decimal:

```
    12
 x 215
 -----
    60   (multiplica por 5, desloca 0)
   12    (multiplica por 1, desloca 1)
  24     (multiplica por 2, desloca 2)
 -----
  2580
```

```
        1010 (10)
 x      1101 (13)
 ----------------
        1010 (multiplica por 1, desloca 0)
       0000  (multiplica por 0, desloca 1)
      1010   (multiplica por 1, desloca 2)
     1010    (multiplica por 1, desloca 3)
 ----------------
     1000010 (130)
```

Algoritmos com Inteiros (Multiplicação)

```
bigInt mul( bigInt x, bigInt y ){
   bigInt r;
   if (y == 0) return 0;
   r = mul(x, y >> 1) // r = x * (y/2)
   if ( par(y) )
       return r << 1; // return 2*r
   else
       return x + r << 1; // return x+2*r
}
```

Sendo *k* o número de bits de x e y,
qual a complexidade de tempo para este algoritmo?

Algoritmos com Inteiros (Multiplicação)

Sendo *k* o número de bits de x e y,
qual a complexidade de tempo para este algoritmo?

São feitas k somas de complexidade O(k), logo:

$$k * O(k) = \mathbf{O(k^2)}$$

Será que tem como fazer melhor?

Algoritmos com Inteiros (Multiplicação)

A resposta é sim. Existe um método que utiliza a abordagem de divisão e conquista para realizar a multiplicação de forma mais eficiente!!

$$\boxed{xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + (x_R y_R)}$$

$$x = x_L \quad x_R = 2^{n/2} x_L + x_R$$

$$y = y_L \quad y_R = 2^{n/2} y_L + y_R$$

$$xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R)$$

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + (x_R y_R)$$

Algoritmos com Inteiros (Multiplicação)

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + (x_R y_R)$$

```
bigInt mul( bigInt x, bigInt y){
    bigInt xl, xr, yl, yr, p1, p2, p3, p4;
    int n = max( x.size(), y.size() ); // número de bits do maior número
    if (n == 1) return xy; // se numéro de bits for 1 (retorna 1 se x = 1 ou y =1).
    xl = leftMost(x, n/2); xr= rightMost(x, n/2); // bits mais a esquerda e mais a direita.
    yl = leftMost(y, n/2); yr= rightMost(y, n/2);
    p1 = mul (xl, yl);
    p2 = mul (xl, yr);
    p3 = mul (xr, yl);
    p4 = mul (xr, yr);
    return (p1 << n) + (p2 << (n/2)) + p4;
}
```

Algoritmos com Inteiros (Multiplicação)

Sendo *k* o número de bits de x e y,
qual a complexidade de tempo para este algoritmo?

$$T(n) = 4*T(k/2) + O(k)$$

<u>Será que tem como fazer melhor?</u>

Algoritmos com Inteiros (Multiplicação)

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + (x_R y_R)$$

```
bigInt mul( bigInt x, bigInt y){
    bigInt xl, xr, yl, yr, p1, p2, p3, p4;
    int n = max( x.size(), y.size() ); // número de bits do maior número
    if (n == 1) return xy; // se numéro de bits for 1 (retorna 1 se x = 1 ou y =1).
    xl = leftMost(x, n/2); xr= rightMost(x, n/2); // bits mais a esquerda e mais a direita.
    yl = leftMost(y, n/2); yr= rightMost(y, n/2);
    p1 = mul (xl, yl);
    p2 = mul (xl, yr);
    p3 = mul (xr, yr);
    return (p1 << n) + ((p2 - p1 - p3) << (n/2)) + p3;
}
```

Algoritmos com Inteiros (Multiplicação)

Sendo *k* o número de bits de x e y,
qual a complexidade de tempo para este algoritmo?

$$T(n) = 3*T(k/2) + O(k)$$

<u>Será que tem como fazer melhor?</u>

Algoritmos com Inteiros (Multiplicação)

Sendo *k* o número de bits de x e y,
qual a complexidade de tempo para este algoritmo?

$$T(n) = 3*T(k/2) + O(k)$$

<u>Será que tem como fazer melhor?</u>

Sim, de acordo com Cormen et al (2002) existe um algoritmo que tem complexidade **O(k log(k) log(log(k)))**

Um Pouco de Teoria dos Números

Teoria dos Números

Notação: d | a ➔ d "divide" a

sendo que $d \geq 1$ e $d \leq |a|$

- Todo inteiro a é divisível pelos **divisores triviais** 1 e *a*
- Número primo: únicos divisores são 1 e *a*
- Todo número composto pode representado pela multiplicação de seus fatores primos (42 = 2*3*7)
- Qual a complexidade de tempo para descobrir se um número *a* qualquer (**int**) é primo?

Divisores Comuns

Um número é dito divisor comum se ele divide dois números:

$d \mid a$ e $d \mid b$ => d é divisor comum de a e b

Propriedade dos divisores comuns:

$a \mid b$ implica em $|a| \leq |b|$

$a \mid b$ e $b \mid a$ implica em $a = b$

$d \mid a$ e $d \mid b$ implica em $d \mid (ax + by)$

O máximo divisor comum entre dois números a e b é denotado por: $mdc\,(a, b)$

$d \mid a$ e $d \mid b$ então $d \mid mdc\,(a, b)$

Divisores Comuns

Primos relativos ou **Primos entre si:**

Dois inteiros são chamados de primos relativos se o único inteiro positivo que divide os dois é 1: $mdc\,(a\,,\,b) = 1$.

Por exemplo, 49 e 15 são primos relativos:

49 → 1,7,49

15 → 1,3,5,15

Propriedade

se $mdc\,(a\,,\,p) = 1$ e $mdc\,(b\,,\,p) = 1$ então $mdc\,(ab\,,\,p) = 1$

Fatoração

Fatoração Única

Um inteiro pode ser escrito como um produto da forma:

$$a = p_1^{e_1} \times p_2^{e_2} \times \ \ldots \times p_r^{e_r}$$

Exemplo

$$6.000 \quad = \quad 2^4 \times 3^1 \times 5^3 \quad = \quad 16 \times 3 \times 125$$

Teste de primalidade: Dado um número *n*, determinar se *n* é primo.

Fatoração de inteiros: Dado um número *n*, representar *n* através de seus fatores primos.

Fatoração

Algoritmo de Euclides

EUCLID (a , b)

se b = 0

então retorne a

senão retorne EUCLID(*b* , *a mod b*)

Complexidade:

Números pequenos: O (log b)

Números grandes (k bits): $O(k^2)$ (**mod*)

Fatoração

Algoritmo de Euclides

Calcular:

EUCLID(2 , 0)

EUCLID(99 ,78)

EUCLID (a , b)

se b = 0

então retorne a

senão retorne EUCLID(*b* , *a mod b*)

Complexidade:

Números pequenos: O (log b)

Números grandes (k bits): O (k^2) (**mod*)

Fatoração

Algoritmo de Euclides Extendido

Adaptar o algoritmo anterior para calcular x e y em:

$$d = mdc\ (a,b) = ax + by$$

EXT-EUCLID (a , b)

se b = 0

então retorne (a , 1 , 0)

(d’, x’, y’) = EXT-EUCLID(*b* , *a mod b*)

(d , x , y) = (d’, y’, x’ – a / b * y’)

retorne (d , x , y)

Divisão inteira

UDESC

Fatoração

Algoritmo de Euclides Extendido

Adaptar o algoritmo anterior para calcular x e y em:

$$d = mdc\,(a,b) = ax + by$$

EXT-EUCLID (a , b)

se b = 0

então retorne (a , 1 , 0)

(d', x', y') = EXT-EUCLID(*b* , *a mod b*)

(d , x , y) = (d', y', x' – a / b * y')

retorne (d , x , y)

Calcular:

EXT-EUCLID(4 , 0)

EXT-EUCLID(99 ,78)

Aritmética Modular

É um sistema para manipular faixas restritas de números inteiros.

Relação de congruência:

$a \equiv b \ (mod\ n) \Leftrightarrow a\ mod\ n$ dá resto $b \Leftrightarrow n\ divide\ (a - b)$.

$a \equiv b \ (mod\ n)$ se e somente se a mod $n = b$ mod n.

Exemplos:

$38 \equiv 14 \ (mod\ 12)$, 38 mod 12 = 14 mod 12

$-10 \equiv 38 \ (mod\ 12)$, -10 mod 12 = 38 mod 12

Aritmética Modular

Soluções para a equação $ax \equiv \mathrm{b} \pmod{n}$

Só há solução se: $mdc(a , n) \mid b$

$ax \equiv b \ (mod\ n)$ tem ***d*** soluções distintas

onde $d = mdc(a , n)$

MOD-LIN-SOLVER(a , b , n)

(d, x', y') = EXT-EUCLID(a, n)

se $d \mid b$

então $x_0 = x'(b / d) \bmod n$

para $i=0$ *a* d - 1 faça

imprimir($x_0 + i(n / d)) \bmod n$

senão imprimir "nenhuma solução"

Aritmética Modular

Soluções para a equação $ax \equiv \mathrm{b} \pmod{n}$

Só há solução se: $mdc(a, n) \mid b$

$ax \equiv b \ (mod\ n)$ tem ***d*** soluções distintas

onde $d = mdc(a, n)$

MOD-LIN-SOLVER(a , b , n)

(d, x', y') = EXT-EUCLID(a, n)

se d | *b*

então $x_0 = x'(b / d) \bmod n$

para i=0 a d - 1 faça

imprimir($x_0 + i(n / d)) \bmod n$

senão imprimir "nenhuma solução"

Calcular:

$14x \equiv 30 \pmod{100}$

MOD-LIN-SOLVER (14 , 30 , 100)

Aritmética Modular

Inverso multiplicativo modular:

O inverso multiplicativo modular de um inteiro $\boldsymbol{a}$ no módulo $\boldsymbol{m}$ é um inteiro $\boldsymbol{x}$ tal que:

$$ax \equiv 1 \ (mod\ m)$$

* Só existe se e somente se $\boldsymbol{a}$ e $\boldsymbol{m}$ são primos relativos.

$$17x \equiv 1 \text{ (mod 120)}$$

MOD-LIN-SOLVER (17 , 1 , 120)

Aritmética Modular

Teorema de Fermat:

Se p é primo então:

$$a^{p-1} \equiv 1 \ (mod\ p)$$

Contudo, a^{p-1} pode ser um número relativamente grande, e realizar a operação de módulo pode ser um problema.

Aritmética Modular

Exponenciação Modular:

Realizar a operação de elevação ao quadrado repetida e realizar o módulo sempre possível.

Exemplo: $2^{53} \ mod \ 101$

Criptografia RSA

Ideia: Permitir que comunicação entre dois participantes sem que um intruso possa entender as mensagens trocadas.

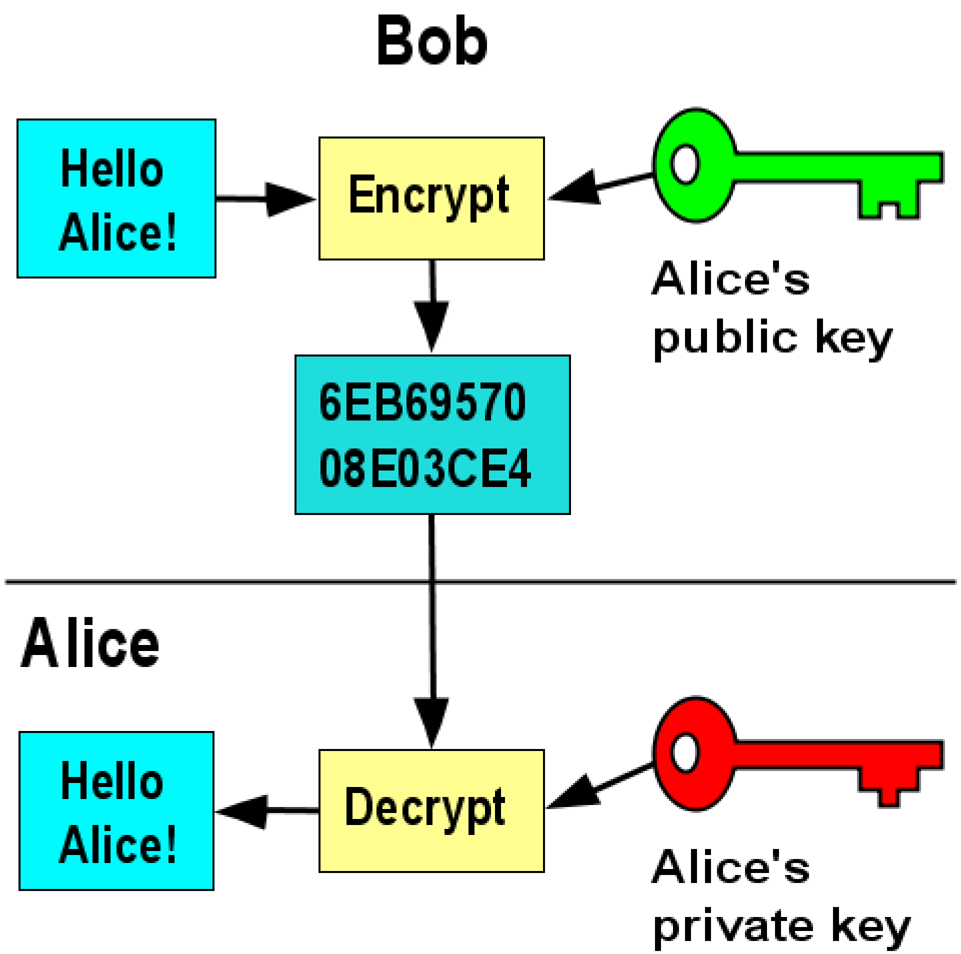
Baseia-se na facilidade em se encontrar números primos grandes e na dificuldade em fatorar o produto entre dois números primos grandes.

Em um sistema de criptografia de chave pública, cada participante possui:

+ uma **chave pública** (pública);

+ uma **chave privada** (secreta);

Criptografia RSA



Como funciona:

- Bob obtém a chave pública de Alice;

- Bob usa a chave para codificar a mensagem M: $C = P_A(M)$ e envia C;

- Alice recebe C e utiliza sua chave privada para recuperar a mensagem original M: $M = S_A(C)$

Criptografia RSA

Algoritmo:

- Selecionar dois números primos grandes ***p*** e ***q***, (512 bits, cada por exemplo) sendo $p \neq q$
- Calcular: $\boldsymbol{n = p * q}$
- Selecionar um inteiro ímpar pequeno ***e*** tal que ***e*** seja primo relativo de $(p - 1)(q - 1)$:

$$mdc\ (e, (p - 1)(q - 1)) = 1$$

Chave pública = (*e*, *n*)

Criptografia RSA

Algoritmo:

- Calcular ***d*** como o inverso modular de ***e***:

$$e*d \equiv 1 \bmod ((p-1)\,(q-1))$$

Chave privada = (*d*, *n*)

Criptografia RSA

Transforma um inteiro *M* (que representa um bloco de dados da mensagem) em um inteiro *C* (que representa um bloco da mensagem criptografada), usando a seguinte função:

$$C = M^e \bmod n$$

Criptografia RSA

A transformação da mensagem criptografada C na mensagem original é executada através da formula:

$$M = C^d \bmod n$$

Criptografia RSA (Exemplo)

Mensagem: “o” => 111 => 01101111

Para $M = 111$, $p = 11$ e $q = 13$ o valor de $n = 143$ e $(p-1) * (q-1) = 120$

Escolher arbitrariamente um valor para $e = 17$ [primo pequeno]

note que: mdc (17, 120) = 1 ***Chave pública* = (17, 143)**.

Como $17d \equiv 1 \pmod{120}$, podemos dizer que $17d = 1 + 120a$,

uma possível solução é $a = -1$ e $d = -7$ (Euclides Estendido),

outras soluções são $a = 16$ e $d = 113$, $a = 33$ e $d = 233$, …

Se -7 é o inverso modular de 17 mod 120 então todo inteiro congruente a -7 mod 120 é também o inverso modular de 17 mod 120: (…, -7, **113**, 233, 353, …).

***Chave privada* = (113, 143)**.

$$C = 111^{17} \bmod 143,\ C = 89.$$

$$M = 89^{113} \bmod 143,\ M = 111.$$

O Caráter Primo

Densidade de números primos

Distribuição dos primos: $\pi(n)$ → *nº de primos* $\leq n$

Exemplo: $\pi(10) = 4$ → *{2,3,5,7}*

Teorema dos números primos: $$\lim_{n\to\infty} \frac{\pi(n)}{n / \ln n} = 1$$

Para *n* grandes, *n* / ln *n* é uma boa aproximação para $\pi(n)$!

O Caráter Primo

Densidade de números primos

Com base no teorema apresentado podemos fazer uma estimativa de probabilidade para verificar se um número escolhido ao acaso é primo ou não como: $1 / \ln(n)$

Quantos números de 512 bits precisamos testar, em média, até encontrar um número primo?

$\ln(2^{512}) \approx 355$ números

$1 / 355 \approx 0{,}28\%$ (chance de encontrar um primo de 1ª)

O Caráter Primo

Densidade de números primos

Para testarmos o caráter primo de um número pequeno *n*, podemos testar verificando a divisibilidade por todos os números entre 2 e $\sqrt{n}$

Para inteiros pequenos: $\Theta(\sqrt{n})$
Para inteiros grandes com *k* bits: $\Theta(2^{k/2})$

O crivo de eratóstenes (algoritmo) é um dos mais conhecidos para criar uma lista de primos até um dado *n*

(ver gif: https://pt.wikipedia.org/wiki/Crivo_de_Erat%C3%B3stenes)

Teste do caráter pseudoprimo

Considerando novamente a equação modular:

$$a^{n-1} \equiv 1 \ (mod\ n)$$

O teorema de Fermat nos diz que se n é primo, então n satisfaz esta equação para qualquer escolha de a ($a \in Z^{+}{}_{n}$)

Se encontrarmos um a que não satisfaça a equação, então certamente n não é primo

Teste do caráter pseudoprimo

Ao testarmos se: $2^{n-1} \equiv 1 \pmod{n}$

caso falso: *n* certamente não é primo

caso verdade: ou *n* é primo, ou *n* é pseudoprimo de base 2

Mas, com que frequência há um falso positivo?

Raramente! Existem apenas 22 valores menores que 10.000:{341, 561, 645, 1105, ...)

Usando 512 bits a chance é de $1 / 10^{20}$

Usando 1024 bits a chance é de $1 / 10^{41}$

Teste do caráter pseudoprimo

Teste Aleatório do Caráter primo de Miller-Rabin

- Experimentar diversos valores como base:

  Melhora a confiabilidade, mas existem números "traiçoeiros" e extremamente raros que dão falso positivo para diferentes bases (números de Carmichael)

- Observar raiz quadrada não trivial de 1 módulo n:

  $x^2 \equiv 1 \pmod{n}$ e x não é 1 ou -1 (ex: x=6, n=35)

MILLER-RABIN(n, s)

    para j=1 a s

        n = RANDOM(1, n-1)

        se (WITNESS(a, n))

        então retorne falso [número composto, certamente]

    retorne verdade [número <u>quase</u> certamente primo]

Trabalho RSA

- Escreva um programa que realize:
  - ❖ a codificação (criptografia RSA) de uma mensagem (String);
  - ❖ a decodificação de uma mensagem criptografada usando a chave privada;
  - ❖ a decodificação de uma mensagem criptografada através de um ataque de força bruta usando a chave pública;

- Quantificar o tempo que leva para decodificar uma mensagem por força bruta
- Pode-se usar as bibliotecas (importadas ou da linguagem) para manipular números grandes (BigInt)

Trabalho RSA

- Funções a serem implementadas:

  getPrimoPequeno( ) : int

    crivoEratóstenes ( )

  primoProvavel(bits : int) : BigInt

    testePrimalidade(n : BigInt) : boolean

  inversoModular(a : BigInt , b : BigInt) : BigInt

    gcdExt (a : BigInt , b : BigInt) : Trio

  expModular(num : BigInt, exp : int, n : BigInt) : BigInt

UDESC

Abordagens para Resolução de Problemas

Abordagens para Resolução de Problemas

Muitos problemas podem ser resolvidos de diversas maneiras diferentes.

Contudo, dependendo da abordagem escolhida, uma solução pode ser melhor ou pior do que outra em questões de tempo ou uso de memória.

Abordagens para Resolução de Problemas

Podemos classificar os principais métodos de resolução de problemas em relação a abordagem utilizada:

- Indução Matemática – Fórmula
- Divisão e Conquista (*divide and conquer*)
- Algoritmos Gulosos (*greedy*)
- Algoritmos de Tentativa e Erro (*backtracking*)
- Programação dinâmica (*dynamic programming*)
- Algoritmos de aproximação (*approximation*)

Indução Matemática – Fórmula

Abordagem mais simples possível.

O problema pode ser reduzido a uma fórmula ou a um conjunto de fórmulas matemáticas.

Exemplo: descobrir as raízes de uma equação de segundo grau.

Algoritmos de Divisão-e-Conquista

(*Divide and Conquer*)

Já estudamos esta abordagem.

Desmembrar o problema original em vários subproblemas semelhantes (menores), resolver os subproblemas (executando o mesmo processo recursivamente) e combinar as soluções.

Exemplos: ordenar um vetor de elementos (Merge-Sort, Quick-Sort)

Algoritmos Gulosos
(*Greedy Algorithms*)

Uma técnica para resolver problemas de otimização: minimizar ou maximizar um determinado valor.

Um algoritmo guloso sempre faz a escolha que parece ser a melhor em um determinado momento, esperando que essa melhor escolha local leve ao melhor resultado do problema como um todo.

Uma técnica simples, mas que na maioria das vezes não leva ao resultado ótimo.

Algoritmos Gulosos
(*Greedy Algorithms*)

Alguns problemas que podem ser resolvidos com algoritmos gulosos:

- Problema do melhor troco - canônico
- Problema da Árvore Geradora Mínima
- Escalonamento de tarefas
- Codificação de Huffman
- Problema da Mochila Fracionada

Problema do Melhor Troco - canônico

Considere que o sistema econômico de um local utiliza unidades monetárias de valor: M_0, M_1, M_2, ... M_n;

Qual é o menor número de unidades monetárias que representam um determinado valor V?

Exemplo: Moedas disponíveis: 1, 5, 10, 25, 50, 100

Valor do troco: 3.82

Qual a menor quantidade de moedas a serem entregues e quais são elas?

Problema do Melhor Troco - canônico

Para sistemas monetários **canônicos** como os usualmente utilizados, podemos adotar a abordagem de escolher o maior número possível das maiores moedas:

Exemplo: Moedas disponíveis: 1, 5, 10, 25, 50, 100

Valor do troco: $ 3.82

382 / 100 = 3 e sobram $ 82	3 moedas de 100
82 / 50 = 1 e sobram 32	1 moeda de 50
32 / 25 = 1 e sobram 7	1 moeda de 25
7 / 5 = 1 e sobram 2	1 moeda de 5
2 / 1 = 2 e sobram 0	2 moedas de 1

Problema do Melhor Troco - canônico

CUIDADO! Para sistemas monetários **não canônicos** a abordagem apresentada pode não resultar no melhor troco possível:

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

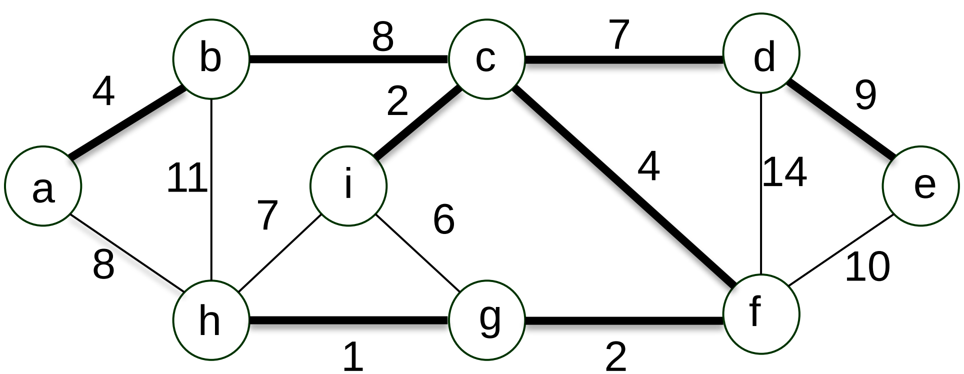
Valor do troco: $ 0.06

Qual é o número mínimo de moedas para o troco?

Problema da Árvore Geradora Mínima

Dado um grafo com seus respectivos vértices e arestas, determinar qual a árvore que contém todos os vértices deste grafo com o menor custo possível.

Escolhemos sempre a melhor aresta dos nós já inclusos (Prim)



Escalonamento de Tarefas

Dada uma lista de tarefas a serem executadas com um horário de início e um horário de término, determinar qual a quantidade máxima de atividades que podem ser executadas

Exemplo: Um auditório só pode ser utilizado para um evento por vez. Em um dia com muitos eventos, deseja-se determinar qual é o maior número de eventos que podem ser realizados no auditório, e quais são eles (OBS: pode haver mais de uma solução).

Evento	1	2	3	4	5	6	7	8	9	10	11
Inicio	3	8	5	1	6	12	0	8	5	2	3
término	5	11	7	4	10	14	6	12	9	13	8

Escalonamento de Tarefas

Solução gulosa: ordenamos os eventos pelo horário de término (em ordem crescente) e sempre que possível pegamos o evento com menor horário de término.

Evento	4	1	7	3	11	9	5	2	8	10	6
Inicio	1	3	0	5	3	5	6	8	8	2	12
término	4	5	6	7	8	9	10	11	12	13	14

Codificação de Huffman

Um método de compressão de dados. Esse método usa a frequência com que cada símbolo ocorre, em uma coleção de dados, para determinar um código de tamanho variável para o símbolo.

Caractere	Frequência	Código (fixo)	Código (variável)
a	45	000	0
b	13	001	101
c	12	010	100
d	16	011	111
e	9	100	1101
f	5	101	1100

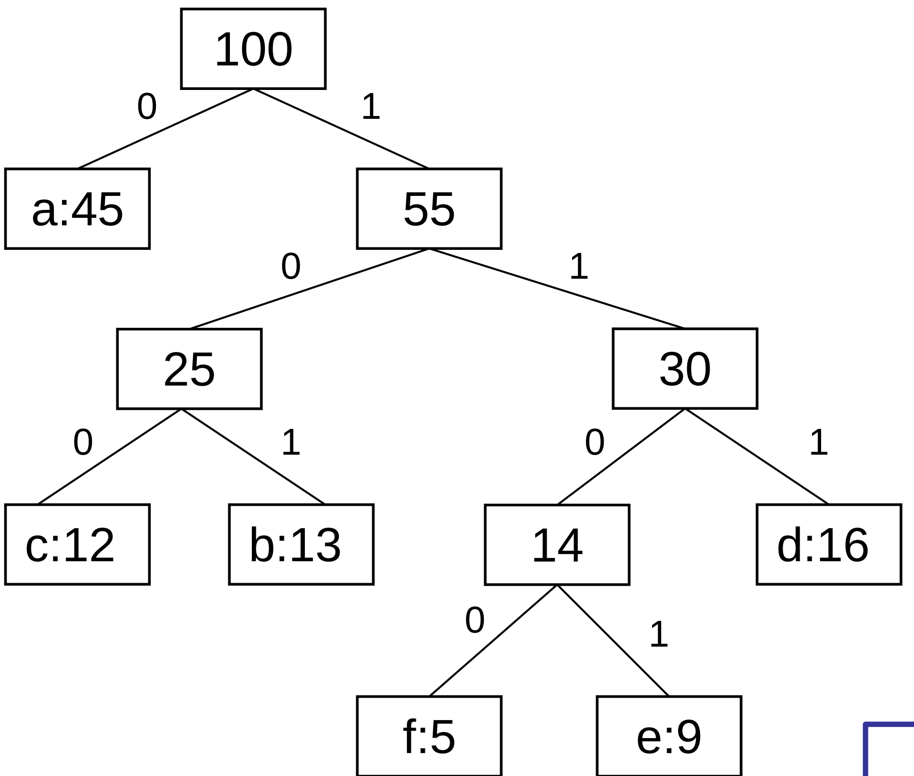
Exemplo:

... faca ...

... 101000010000 ...

... 110001000 ...

Codificação de Huffman (Exemplo)



Exemplo:

... faca ...

... 101000010000 ...

... 110001000 ...

Codificamos usando a notação:
Filhos a esquerda: 0
Filhos a direita: 1
Até chegar a letra desejada

Codificação de Huffman

Huffman(C)
$n \leftarrow |C|$
$Q \leftarrow C$
para $i \leftarrow 1$ **até** n - 1
$z \leftarrow$ AlocaNo()
$x \leftarrow z.esq \leftarrow$ Extrair-Minimo(Q)
$y \leftarrow z.dir \leftarrow$ Extrair-Minimo(Q)
$z.valor = x.valor + y.valor$
Inserir(Q, z)
retorne Q

Testando o algoritmo:
Como ficaria a codificação para a string:

"testando o teste"

Construir a árvore e o mapa de codificação

Importante:

Note que para decodificar uma string, é necessário conhecer a árvore que foi usada para gerá-la!
Ou seja, espaço utilizado: nova string (menor) + árvore

Problema da Mochila Fracionada

Dark Version: Um ladrão está assaltando uma loja e possui uma mochila que pode carregar até **P** kg sem arrebentar. Sabendo os pesos e valores dos itens, qual é o maior valor possível que o ladrão conseguirá roubar?

Na versão da mochila fracionada os elementos podem ser pegos em fração (apenas parte do objeto, com proporção linear entre peso e valor)

Exemplo: Mochila com capacidade 9 kg

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Problema da Mochila Fracionada

Solução Gulosa: Ordenamos o vetor de itens em relação a proporção: [valor / peso]

Objetos	**1**	**3**	**4**	**6**	**5**	**2**
Peso	1	2	5	9	7	3
Valor	3	4	7	12	9	2
Proporção	3	2	1.4	1.33	1.29	0.67

Tentativa e Erro
(*Backtracking*)

Outra técnica para resolver problemas de otimização

Testa todas as possíveis soluções até encontrar a(s) melhor(es) solução(ões) para um problema, utilizando as ideias de busca em profundidade ou busca em largura

Pode utilizar validações para diminuir (de maneira não muito significativa) o escopo de busca [ex: desconsiderar soluções parciais piores do que uma solução final já encontrada]

Geralmente não adequada para instâncias muito grandes

Tentativa e Erro
(*Backtracking*)

Alguns problemas clássicos que são resolvidos (também) com *backtracking* (instâncias pequenas):

- Problema do passeio do cavalo
- Puzzle 3x3
- Sokoban
- Caixeiro Viajante

Passeio do Cavalo

Partindo de uma posição inicial, qual é o menor número de movimentos que um cavalo deve realizar para chegar a qualquer uma das casas de um tabuleiro?

LEMBRETE: o cavalo (*knight*) anda apenas em movimentos 'L'

Exemplo: tab. 5x5   c->2,1

- Busca em Profundidade...
- Busca em Largura...

Tentativa e Erro
(*Backtracking*)

Solução usando Busca em Profundidade:

[+] usa menos memória (apenas tabuleiro)

[–] repete muitos movimentos (mais lento)

Solução usando Busca em Largura:

[+] não repete movimentos (mais rápido)

[–] usa mais memória (para salvar as posições)

em alguns casos se torna inviável

Puzzle 3x3

Montar a imagem original reorganizando as peças, podendo movimentar as peças usando a única lacuna.

Como resolver computacionalmente?

Puzzle 3x3

Como resolver computacionalmente?

- Abstrair que os movimentos se baseiam na lacuna: ela pode ir para cima, baixo, direita e esquerda;
- Representar o 'puzzle' como uma estrutura de dados;
- Armazenar a quantidade de movimentos (mínima) necessárias para alcançar uma dada posição.
- Descobrir o mínimo para chegar à posição final.

Puzzle 3x3

Exemplo:

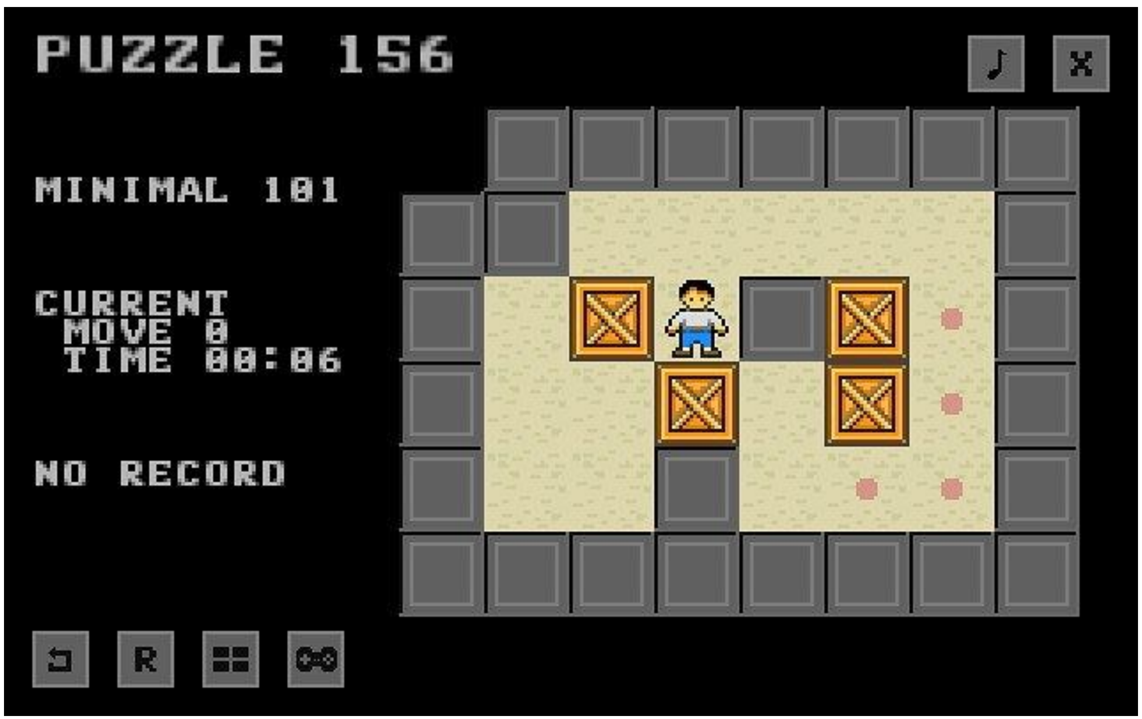
Peças são representadas por números entre 1 e 8

Lacuna é representada pelo número 0

1	**0**	**3**
4	2	5
7	8	6

Sokoban

Levar as caixas para pontos objetivos, em qualquer ordem, com o menor número de movimentos possível.



Quais valores representam um estado neste problema?

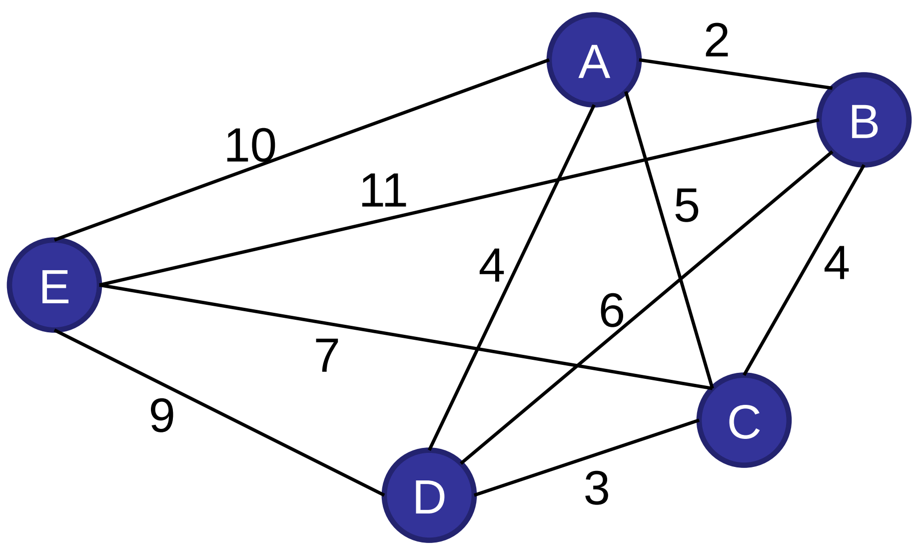
Sokoban

Posição do jogador (x,y)	(1,2) e (2,2)
Posição das caixas (x,y)	[1][2][2][2] array

Problema do Caixeiro Viajante

Um vendedor deseja visitar n cidades e retornar a cidade de origem. Dado um grafo não orientado completo com n vértices, onde existe um custo $c(i, j)$ (associado a cada aresta) para viajar da cidade i a cidade j.

Qual o trajeto com custo mínimo?



Programação Dinâmica
(*Dynamic Programming*)

Uma abordagem auxiliar para reduzir a computação em problemas de otimização.

Consiste em dividir o problema original em subproblemas mais simples, resolvê-los, armazenar os resultados em memória e consultar estes resultados posteriormente a fim de reduzir o espaço de busca.

Resultados são calculados apenas uma vez (nunca são recalculados por outra iteração)

Programação Dinâmica
(*Dynamic Programming*)

Alguns problemas que podem ser resolvidos com programação dinâmica:

- Fibonacci
- Problema do Menor Troco não canônico
- Problema da Mochila Binária (0/1)
- Maior Subsequência Comum
- Multiplicação de Cadeias de Matrizes

Fibonacci

O modelo recursivo de Fibonacci pode ser extremamente ruim dependendo da implementação:

```
int fib(int n){
    if( n<=1 ) return n;
    return fib(n-1) + fib(n-2);
}
```

Fibonacci

Para entender a programação dinâmica, talvez o problema da sequencia de Fibonacci seja o exemplo mais simples para demonstrar o seu uso.

A ideia é utilizar uma estrutura auxiliar para armazenar os resultados e utilizá-los posteriormente

Fibonacci

Algoritmo utilizando programação dinâmica:

```
int fibonacci[50];
void init(){
    for(int i=0; i<50; i++)
        fibonacci[i] = -1;
}
int fib(int n){
    if( fibonacci[n]!=-1 )
        return fibonacci[n];
    if( n<=1 )
        return n;
    fibonacci[n] = fib(n-1) + fib(n-2);
    return fibonacci[n];
}
```

Fibonacci

Abordagem *top-down* (*Memorization*)

Fib(n)

  se $n \leq 1$ então

    retorne n

  senão

    se F[n] está indefinido

      F[n] $\leftarrow$ Fib($n - 1$) + Fib($n - 2$)

    retorne F[n]

Fibonacci

Abordagem *bottom-up*

Fib(n)

  $F[0] = 0$

  $F[1] = 1$

  para $i \leftarrow 2$ até n

    $F[i] = F[i-2] + F[i-1]$

  retorne $F[n]$

Problema do Melhor Troco não canônico

Como visto anteriormente, para sistemas monetários **não canônicos** a abordagem gulosa pode não resultar no melhor resultado!

Nesse caso, podemos usar programação dinâmica como um método eficiente para encontrar o melhor troco:

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

Valor do troco: $ 0.06

Qual é o número mínimo de moedas para o troco?

Problema do Melhor Troco não canônico

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

Valor do troco: $ 0.06

Considere as moedas como um vetor: M = { 1, 3, 4 }

Inicialmente assumimos que o troco para $ 0.00 precisa de zero moedas

$$T(0) = 0$$

E seguimos iterativamente até o troco desejado usando a fórmula:

$$T(n) = \min(\ 1+T(n-M[0]),\ 1+T(n-M[1]),\ 1+T(n-M[2])\ )$$

Problema do Melhor Troco não canônico

Exemplo Trivial: Moedas disponíveis: 1, 3, 4

Valor do troco: $ 0.06

$$T(0) = 0$$

$$T(n) = \min(\ 1+T(n-M[0]) , 1+T(n-M[1]) , 1+T(n-M[2]) \ )$$

$$T(1) = \min(\ 1+T(1-1) , 1+\cancel{T(1-3)} , 1+\cancel{T(1-4)} \ ) = 1$$

$$T(2) = \min(\ 1+T(2-1) , 1+\cancel{T(2-3)} , 1+\cancel{T(2-4)} \ ) = 2$$

$$T(3) = \min(\ 1+T(3-1) , 1+T(3-3) , 1+\cancel{T(3-4)} \ ) = 1$$

(...)

Complexidade de tempo => O (n * |M|)

Complexidade de espaço => O (n)

Problema da Mochila Binária (0/1)

Dark Version: Um ladrão está assaltando uma loja e possui uma mochila que pode carregar até **P** kg sem arrebentar. Sabendo os pesos e valores dos itens, qual é o maior valor possível que o ladrão conseguirá roubar?

- Na versão da mochila binária os elementos só podem ser pegos como um todo (não podemos pegar metade de um item – ou pega, ou não pega).

- **Exemplo**: Mochila com capacidade 9 kg

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Problema da Mochila Binária (0/1)

Solução Matricial

Montamos uma matriz para armazenar as melhores soluções intermediárias e as usamos para gerar as próximas soluções;

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Problema da Mochila Binária (0/1)

Solução Matricial

Objetos	1	2	3	4	5	6
Peso	1	3	2	5	7	9
Valor	3	2	4	7	9	12

Item / Capacidade	0	1	2	3	4	5	6	7	8	9
1	0	3	3	3	3	3	3	3	3	3
2	0	3	3	3	5	5	5	5	5	5
3	0	3	4	7	7	7	9	9	9	9
4	0	3	4	7	7	7	10	11	14	14
5	0	3	4	7	7	7	10	11	14	14
6	0	3	4	7	7	7	10	11	14	**14**

```
m[item][k] = min( m[item-1][k], valor + m[item-1][k-peso] )
```

Problema da Mochila Binária (0/1)

Solução Matricial

Complexidade de tempo => O (peso * qtd-itens)

Complexidade de espaço => O (peso * qtd-itens)

Maior Subsequência Comum

Longest Common Subsequence

Dadas duas strings S e T, qual é maior subsequencia (sequencia não necessariamente contígua), da esquerda para a direita, entre estas strings?

- **Exemplo**: S = ABAZDC

  T = BACBAD

  “ABAD”

Resolução similar a abordagem matricial para resolver o problema da mochila.

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

Dadas uma sequencia de *n* matrizes, qual é a melhor ordem para multiplicá-las a fim de realizar menos multiplicações.

OBS: a multiplicação entre matrizes é <u>associativa</u>, logo (AB)(CD) = A(B(CD)) = (AB)C)D = (A(BC))D = A((BC)D)

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$ $\quad$ $M_1 * M_2 = 2 * 3 * 6 = 36$

$M_2 = (3,6)$ $\quad$ $M_2 * M_3 = 3 * 6 * 4 = 72$

$M_3 = (6,4)$ $\quad$ $M_3 * M_4 = 6 * 4 * 5 = 120$

$M_4 = (4,5)$

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$

$M_2 = (3,6)$

$M_3 = (6,4)$

$M_4 = (4,5)$

$M_1 * M_2 = 2 * 3 * 6 = 36$

$M_2 * M_3 = 3 * 6 * 4 = 72$

$M_3 * M_4 = 6 * 4 * 5 = 120$

MATRIZ	1	2	3	4
1	0	36		
2		0	72	
3			0	120
4				0

MATRIZ	1	2	3	4
1	0	1		
2		0	2	
3			0	3
4				0

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$ $\quad (M_1 * M_2)\, M_3 = 36 + 2 * 6 * 4 = 84$

$M_2 = (3,6)$ $\quad M_1\, (M_2 * M_3) = 2 * 3 * 4 + 72 = 96$

$M_3 = (6,4)$ $\quad (M_2 * M_3)\, M_4 = 72 + 3 * 4 * 5 = 132$

$M_4 = (4,5)$ $\quad M_2\, (M_3 * M_4) = 3 * 6 * 5 + 120 = 210$

MATRIZ	1	2	3	4
1	0	36	84	
2		0	72	132
3			0	120
4				0

MATRIZ	1	2	3	4
1	0	1	1	
2		0	2	2
3			0	3
4				0

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

- **Exemplo**: São dadas 4 matrizes com os seguintes tamanhos:

$M_1 = (2,3)$ $(M_1 M_2 M_3) M_4 = 84 + 2 * 4 * 5 = 124$

$M_2 = (3,6)$ $(M_1 M_2) * (M_3 M_4) = 36+2*6*5+120 = 216$

$M_3 = (6,4)$ $M_1 (M_2 M_3 M_4) = 2 * 3 * 5 + 132 = 162$

$M_4 = (4,5)$

MATRIZ	1	2	3	4
1	0	36	84	124
2		0	72	132
3			0	120
4				0

MATRIZ	1	2	3	4
1	0	1	1	1
2		0	2	2
3			0	3
4				0

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

PSEUDOCÓDIGO

```
MATRIX-CHAIN-MULT(   v[n]  )
    para  i de 1 a n:                    // zera a primeira linha
        m[ i , i ] = 0
    para  c de 2 a n:                    // para todas as linhas posteriores
        para  i de 1 a (n - c + 1):      // para todos os elem. da diagonal
            j = i + c - 1
            m[ i , j ] = ∞
            para  k de i a (j - 1):      // para todas as permutações
                q = m[ i , k ] + m[ k+1 , j ] + v[ i-1 ] * v[ k ] * v[ j ]
                se  q < m[ i , j ]:
                    m[ i , j ] = q
                    s[ i, j] = k
```

Multiplicação de Cadeias de Matrizes

Matrix Chain Multiplication

PSEUDOCÓDIGO

```
PRINT-EQUACAO (  s[n][n] , i , j  )
    se i == j
        print  "M" + i
    senão
        print  "("
        PRINT-EQUACAO(   s , i , s[ i , j ] )
        PRINT-EQUACAO(   s , s[ i , j ] + 1 , j )
        print  ")"
```

MATRIZ	1	2	3	4
1	0	1	1	1
2		0	2	2
3			0	3
4				0

```
P-E (1,4)
    P-E(1,1) => "M1"
    P-E(2,4)
        P-E(2,2) => "M2"
        P-E(3,4)
            P-E(3,3) => "M3"
            P-E(4,4) => "M4"
```

(M1 (M2 (M3 M4)))

Algoritmos de Aproximação

(*Approximation Algorithms*)

Existem alguns problemas os quais não se conhece uma solução eficiente. Dizemos que estes problemas são "difíceis" ou intratáveis pois, para instâncias grandes, o tempo de processamento seria inviável.

Nestas situações é comum remover a exigência de procurar pela solução ótima e passamos a procurar por uma solução próxima da ótima (solução boa / razoável)

Algoritmos de Aproximação
(*Approximation Algorithms*)

Existem duas abordagens principais:

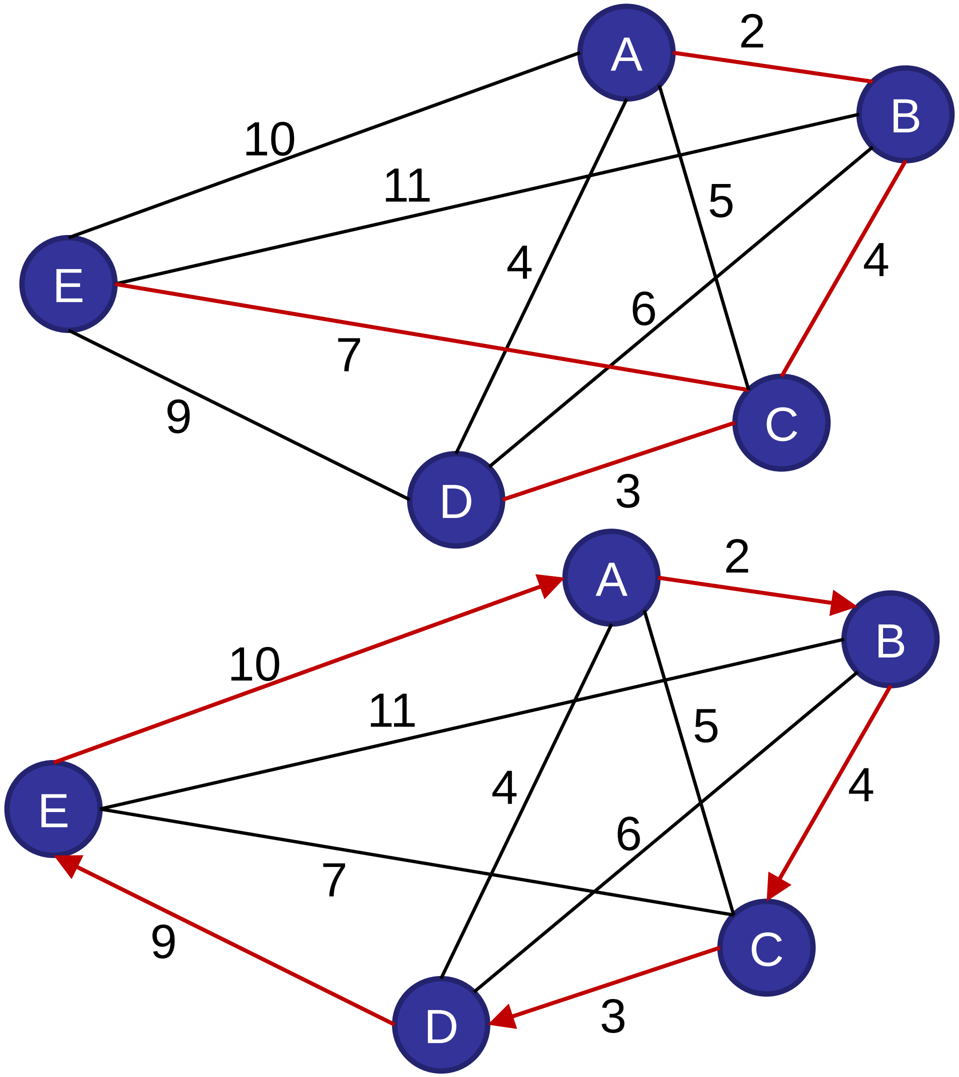
- **Heurística**: algoritmo que buscam soluções próximas a solução ótima. Utilizam alguma informação (ou intuição) sobre a instância do problema para resolvê-lo de forma eficiente. Podem ser genéricas (servem p/ vários problemas).

- **Algoritmo Aproximado**: algoritmo que resolve um problema de forma eficiente e ainda garante a qualidade da solução. É necessário provar que a solução está próxima da ótima. Geralmente únicos para cada problema.

Heurísticas

- **Algoritmos Bio-inspirados**: algoritmo baseados no comportamento de seres vivos:
  - **Algoritmos genéticos**: gera soluções, escolhe as melhores (seleção natural) e gera um novo ciclo, oferecendo espaço para mutações (alterações nos componentes das soluções).
  - **Colônia de formigas**: modela a geração de soluções baseado no comportamento das formigas (como elas sempre encontram um doce?!) .

Algoritmos de Aproximação
Caixeiro Viajante



Dado o grafo $G = (V, A)$ e o custo c:

1. Selecione um vértice $r \in V$ para ser o vértice *raiz*.
2. Obtenha a ávore geradora mínima a partir de r.
3. Faça H ser a lista de vertices ordenados de acordo com a primeira visita, considerando um percurso em pré-ordem, retornando a r.

Se a função custo satisfaz a desigualdade de triângulos: $c(u, w) < c(u, v) + c(v, w)$

$$c(H) < 2c(\acute{o}timo)$$

Exercícios

Analisar e resolver os seguintes problemas:

- https://www.urionlinejudge.com.br/judge/pt/problems/view/1034
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1286
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1288
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1310
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1351
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1458
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1602
- https://www.urionlinejudge.com.br/judge/pt/problems/view/1976
- https://www.urionlinejudge.com.br/judge/pt/problems/view/2032

Estudo da Tratabilidade de Problemas Computacionais

Problemas tratáveis e intratáveis

Problemas tratáveis: resolvidos por algoritmos que executam em tempo polinomial.

Problemas intratáveis: não se conhece algoritmos que os resolvam em tempo polinomial.

$$1 \prec \log\log n \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}$$

Problemas tratáveis e intratáveis

Problemas tratáveis: resolvidos por algoritmos que executam em tempo polinomial.

Problemas intratáveis: não se conhece algoritmos que os resolvam em tempo polinomial.

$$\boxed{1 \prec \log\log n \prec \log n \prec n^{\varepsilon} \prec n^{c}} \prec \boxed{n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}}$$

Categorias de Problemas

Problemas de Otimização: Cada solução possível tem um valor associado e desejamos encontrar a solução com melhor valor.

Problemas de Decisão: Problemas que tem resposta sim ou não.

Problemas de Decisão são possivelmente “mais fáceis” do que problemas de Otimização, mas com certeza “não mais difíceis”!

Exemplo:

- Qual é o menor caminho entre os vértices *a* e *b* de um grafo?
- Existe um caminho de no máximo k arestas entre *a* e *b*?

Algoritmos Não Deterministas

Capaz de escolher uma entre várias alternativas possíveis a cada passo. A alternativa escolhida será sempre a alternativa que leva a conclusão esperada, caso essa alternativa exista.

Autômato não determinista

```
int pesq(Estr *v, int n, int ch){
    int i;
    for (i = 0; i < n; i++)
        if (v[i].chave == ch)
            return i;
    return -1;
}
```

```
int pesq(Estr *v, int n, int ch){
    int i;
    i = escolheND(0, n - 1);
    if (v[i].chave == ch)
        return i;
    return -1;
}
```

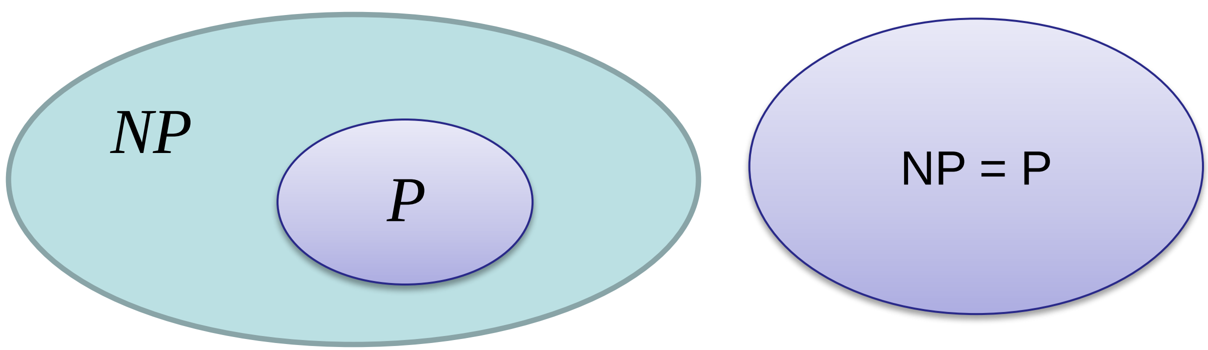
Classes de Problemas *P* e *NP*

Classe de Problemas *P*: Problemas que podem ser resolvido (por algoritmos deterministas) em tempo polinomial.

Classe de Problemas *NP*: Problemas que podem ser resolvidos por algoritmos não deterministas em tempo polinomial. Ou problemas que a solução pode ser verificada em tempo polinomial.

Pergunta do milhão: P=NP ou P ≠NP?

Possíveis relações entre as classes:



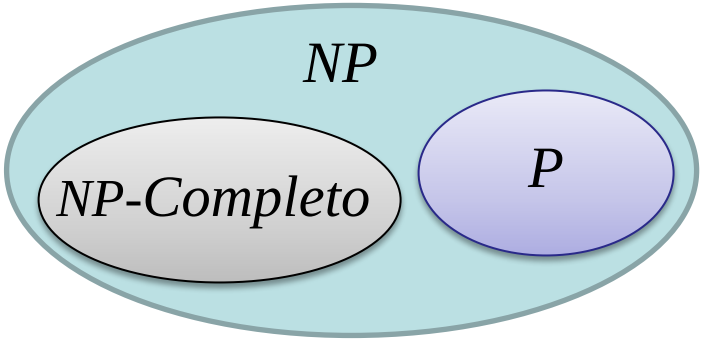
Classes de Problemas *NP-Completo*

Classe de Problemas *NP-Completo*: É um problema NP e é tão difícil quanto qualquer outro problema NP.

Se um problema NP-Completo pode ser resolvido em tempo polinomial, então todo problema NP-Completo pode ser resolvido em tempo polinomial e, portanto, P = NP

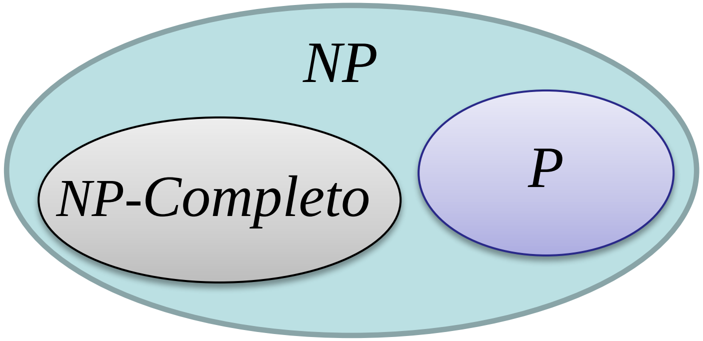
Acredita-se que a relação correta seja P ≠ NP

Por quê?



Classes de Problemas *NP-Completo*

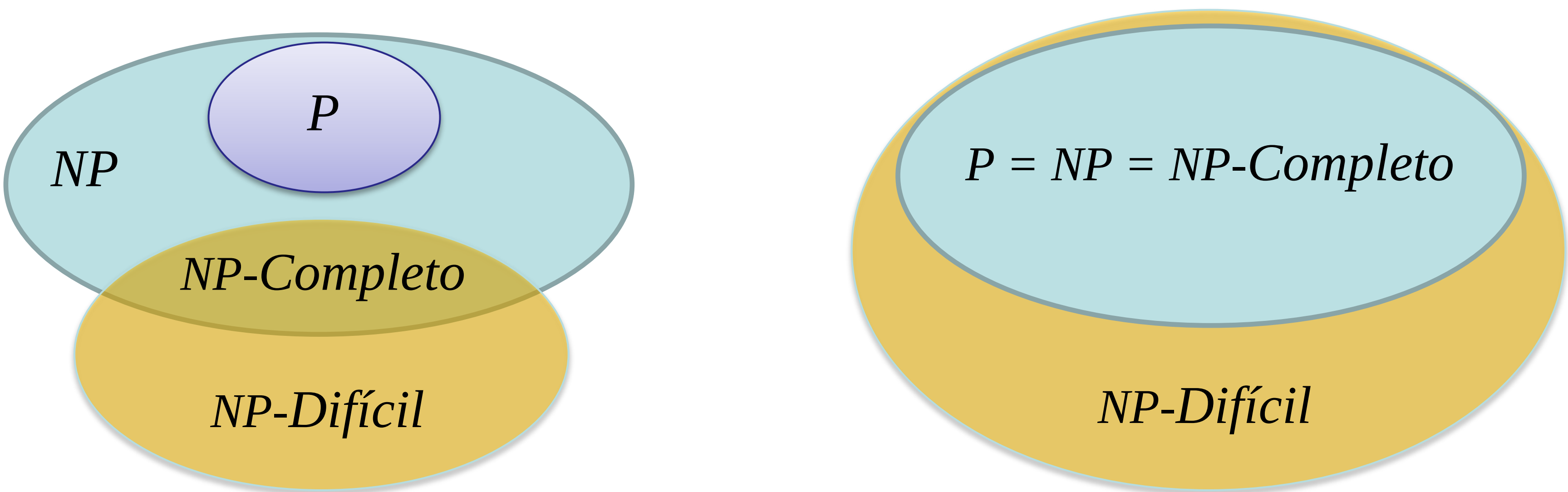
E existe um problema que não pode ser resolvido em tempo polinomial por uma algoritmo não determinista?



NP-Hard

Sim, existem! Eles são classificados como problemas **NP-*Hard*** ou **NP-Difíceis**!

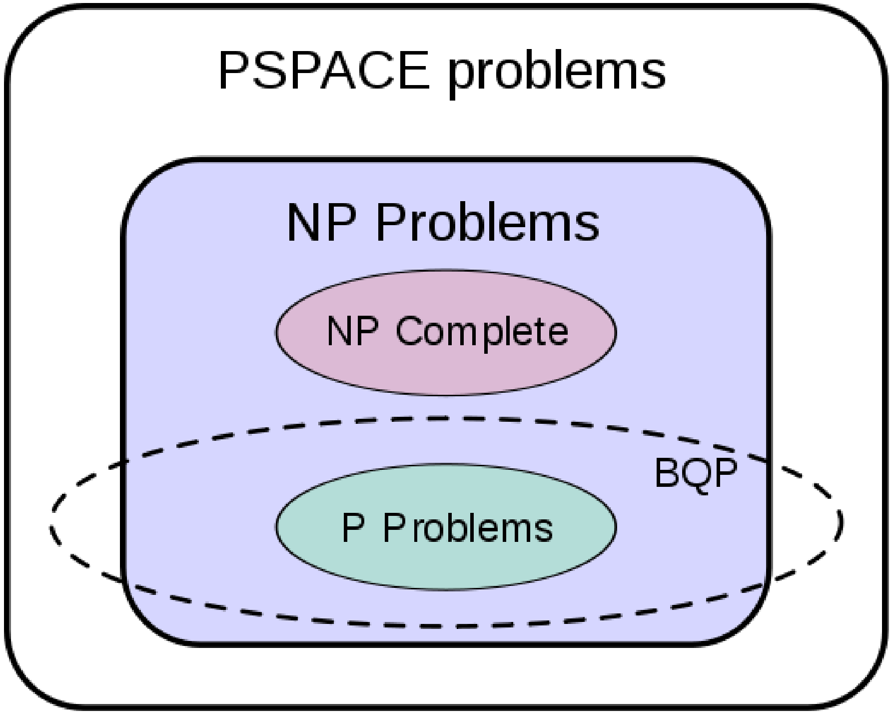
Nesta classe podemos encontrar problemas intratáveis mas decidíveis e problemas indecidíveis (ex: Problema da Parada).



Computadores Quânticos

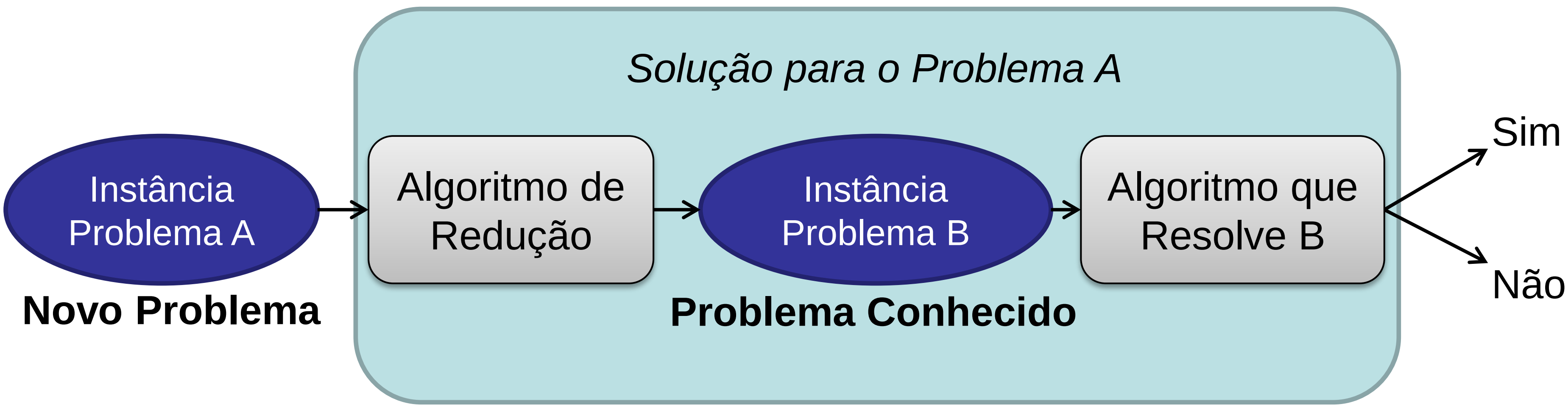
- Servem para problemas que podem ser resolvidos com chutes aleatorizados: fatoração (quebrar o RSA), logaritmo discretos e simulações de física quântica;

- Utilizam **qubits**;
- Podem reduzir o tempo em até 2^{qubits};
- **BQP**: problemas que podem ser resolvidos em tempo polinomial com alta probabilidade;
- Dúvidas e incertezas.



Redução de Problemas

Redução para problemas de decisão



A $\leq_p$ B

Se o "**Algoritmo de Redução**" e o "**Algoritmo que Resolve B**" forem algoritmos polinomiais, então podemos concluir algo sobre a solução do Problema A?

Redução de Problemas

Relação entre Redução e Problemas NP-Completos:

Queremos representar a dificuldade dos problemas NP-Completos e criar uma relação entre eles.

Cenário 1

Problema A

Problema B

Algoritmo de Redução

Problema Intratável: Não existe solução em tempo polinomial

Problema: Não sabemos se é tratável ou intratável

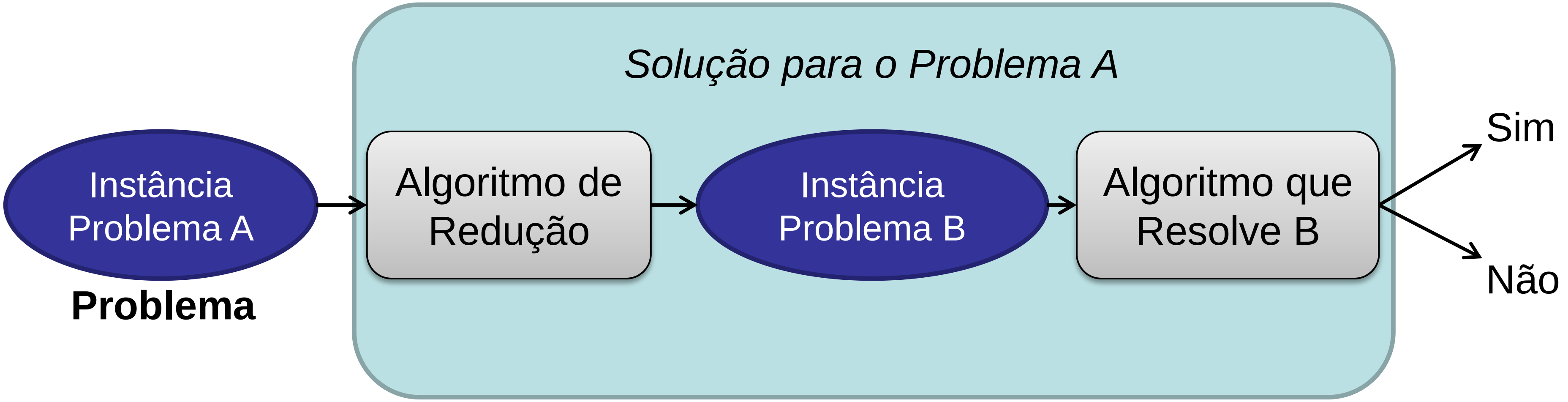
Existe um algoritmo de Redução de A para B em tempo polinomial

QUEREMOS DESCOBRIR

Redução de Problemas

Relação entre Redução e Problemas NP-Completos:

Vamos considerar que o Problema B tenha uma solução polinomial (ou seja, é tratável). Se isso for verdade:



Encontramos uma contradição: **Problema A** teria de ser polinomial!
Logo, **Problema B** não poderia ter uma solução polinomial

Redução de Problemas

Relação entre Redução e Problemas NP-Completos:

Queremos representar a dificuldade dos problemas NP-Completos e criar uma relação entre eles.

Cenário 2

Problema A

Problema B

Algoritmo de Redução

Problema Tratável: Existe solução em tempo polinomial

Problema: Não sabemos se é tratável ou intratável

Existe um algoritmo de Redução de A para B em tempo polinomial

QUEREMOS DESCOBRIR

NP-Completo

Um problema X é ***NP-Completo*** se:

1. O problema deve ser NP: $X \in NP$

   a) *Conseguir um algoritmo não determinista que resolva o problema em tempo polinomial*

   b) *Conseguir um algoritmo determinista que verifica se uma resposta é verdadeira ou não (**certificado**)*

2. Fazer a redução deste problema para qualquer outro problema NP-Completo: $X' \leq_p X$ para todo $X' \in NP$

NP-Completo

Teorema de Cook: SAT é um problema NP-Completo e está em *P* se e somente se *P* = *NP*

É possível reduzir qualquer máquina de Turing não determinista (MTND) no problema SAT em tempo polinomial.

$$\text{MTND} \;\; \leq_p \;\; \text{SAT}$$

Não vamos fazer essa redução pois ela é mais longa
(acreditem no teorema! É possível!)

(SAT) Satisfazibilidade de Fórmulas Booleanas

O problema da ***Satisfazibilidade de fórmulas booleanas*** consiste em determinar se existe uma atribuição de valores booleanos, para as varáveis que ocorrem na fórmula, de tal forma que o resultado seja ***verdadeiro***.

Um ***literal*** é uma variável proposicional ou sua negação.

Exemplo:

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

Problema de Decisão: existe uma combinação de valores para x_1 e x_2 que satisfazem esta equação?

(SAT) Satisfazibilidade de Fórmulas Booleanas

Classificando SAT como NP-Completo:

Passo 1: Algoritmo de certificado (determinista e polinomial)

Passo 2: MTND $\leq_p$ SAT

```
bool certificado( bool *sol ){
   return sol[1] &&
         (sol[2] || !sol[1]) &&
         (!sol[2] || !sol[3]) &&
         (!sol[1] || sol[2] || sol[3]);
}
```

Complexidade?

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

!! Um algoritmo de certificado genérico usaria um for para iterar sobre cada literal ou operador da expressão !!

Neste caso qual seria a complexidade do algoritmo?

(SAT) Satisfazibilidade de Fórmulas Booleanas

Classificando SAT como NP-Completo:

Passo 1: Algoritmo de certificado (passed)

Passo 2: MTND $\leq_p$ SAT (passed)

Logo, provamos que SAT pertence ao conjunto de problemas NP-Completo!

Forma Normal Conjuntiva

Uma formula booleana está na ***Forma Normal Conjuntiva*** (***CNF***) se é expressa por um grupo cláusulas AND, cada uma das quais formada por OR entre literais.

Uma fórmula booleana esta na ***k-CNF*** se cada cláusula possui exatamente k literais:

Exemplo 2-CNF:

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

3-CNF-SAT

Problema: verificar se uma fórmula booleana na 3-CNF é satisfazível.

3-CNF-SAT é *NP-Completo*?

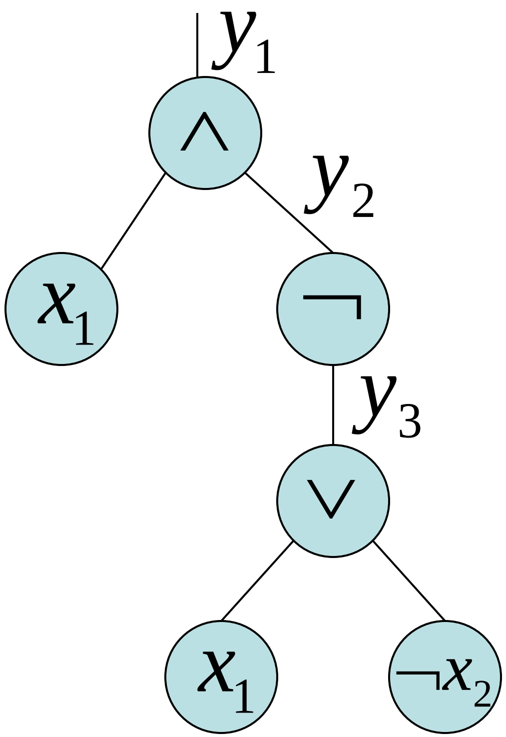
- **Passo 1**: 3-CNF-SAT $\in NP$.
- **Passo 2**: SAT $\leq_p$ 3-CNF-SAT.

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

SAT $\leq_p$ 3-CNF-SAT

Dada uma fórmula booleana: $\phi = x_1 \wedge \neg(x_1 \vee \neg x_2)$

SAT

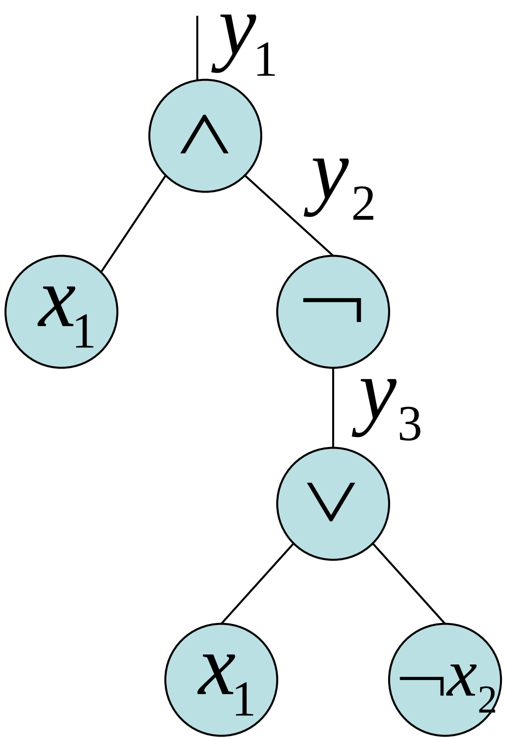


REDUÇÃO

1. Construir uma árvore que represente à fórmula.
2. Introduzir uma variável y_i para a raiz e a saída de cada no interno.

SAT $\leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$



3. Reescrevemos a fórmula original como conjunções entre a variável raiz e as cláusulas que descrevem as operações de cada nó.

Introduz **uma** variável e **uma** cláusula para cada operador.

SAT $\leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

4. Para cada ϕ_i' construir uma tabela verdade, usando as entradas que tornam $\neg\phi_i'$ verdade, construir uma forma normal disjuntiva para cada ϕ_i'.

SAT $\leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (\underbrace{y_1 \leftrightarrow (x_1 \wedge y_2)}) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

y_1	x_1	y_2	$y_1 \leftrightarrow (x_1 \wedge y_2)$
V	V	V	V
V	V	F	F
V	F	V	F
V	F	F	F
F	V	V	F
F	V	F	V
F	F	V	V
F	F	F	V

$$\begin{aligned}\neg\phi_2'' = & (y_1 \wedge x_1 \wedge \neg y_2) \\ & \vee (y_1 \wedge \neg x_1 \wedge y_2) \\ & \vee (y_1 \wedge \neg x_1 \wedge \neg y_2) \\ & \vee (\neg y_1 \wedge x_1 \wedge y_2)\end{aligned}$$

Cada cláusula de ϕ' introduz no máximo 8 cláusulas em ϕ'', pois cada cláusula de ϕ' possui no máximo 3 variáveis.

SAT $\leq_p$ 3-CNF-SAT

$$\neg\phi_2'' = (y_1 \wedge x_1 \wedge \neg y_2) \vee (y_1 \wedge \neg x_1 \wedge y_2) \vee$$
$$(y_1 \wedge \neg x_1 \wedge \neg y_2) \vee (\neg y_1 \wedge x_1 \wedge y_2)$$

Converter a fórmula para a CNF usando as leis de De Morgan:

$$\phi_2'' = (\neg y_1 \vee \neg x_1 \vee y_2) \wedge (\neg y_1 \vee x_1 \vee \neg y_2) \wedge$$
$$(\neg y_1 \vee x_1 \vee y_2) \wedge (y_1 \vee \neg x_1 \vee \neg y_2)$$

SAT $\leq_p$ 3-CNF-SAT

O último passo faz com que cada cláusula tenha exatamente 3 literais, para isso usamos duas novas variáveis p e q. Para cada cláusula C_i em ϕ'' :

1. Se C_i tem 3 literais, simplesmente inclua C_i.
2. Se C_i tem 2 literais, $C_i = (l_1 \vee l_2)$, inclua:

$$(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$$

3. Se C_i tem 1 literal, l_1, inclua:

$$(l_1 \vee p \vee q) \wedge (l_1 \vee \neg p \vee \neg q) \wedge (l_1 \vee p \vee \neg q) \wedge (l_1 \vee \neg p \vee q)$$

Introduz no máximo **4** cláusulas por cláusula em ϕ''.

SAT $\leq_p$ 3-CNF-SAT

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

$$\phi_1''' = (y_1 \vee p \vee q) \wedge (y_1 \vee \neg p \vee \neg q) \wedge (y_1 \vee p \vee \neg q) \wedge (y_1 \vee \neg p \vee q)$$

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (x_1 \wedge y_2)) \wedge (y_2 \leftrightarrow \neg y_3) \wedge (y_3 \leftrightarrow (x_1 \vee \neg x_2))$$

$$(y_1 \vee p \vee q) \wedge (y_1 \vee \neg p \vee \neg q) \wedge (y_1 \vee p \vee \neg q) \wedge (y_1 \vee \neg p \vee q) \wedge$$
$$(\neg y_1 \vee \neg x_1 \vee y_2) \wedge (\neg y_1 \vee x_1 \vee \neg y_2) \wedge (\neg y_1 \vee x_1 \vee y_2) \wedge (y_1 \vee \neg x_1 \vee \neg y_2)$$

3-CNF-SAT

Problema: verificar se uma fórmula booleana na 3-CNF é satisfazível.

3-CNF-SAT é *NP-Completo*? SIM

- **Passo 1**: 3-CNF-SAT $\in NP$.
- **Passo 2**: SAT $\leq_p$ 3-CNF-SAT.

$$\boxed{(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)}$$

$$\downarrow$$

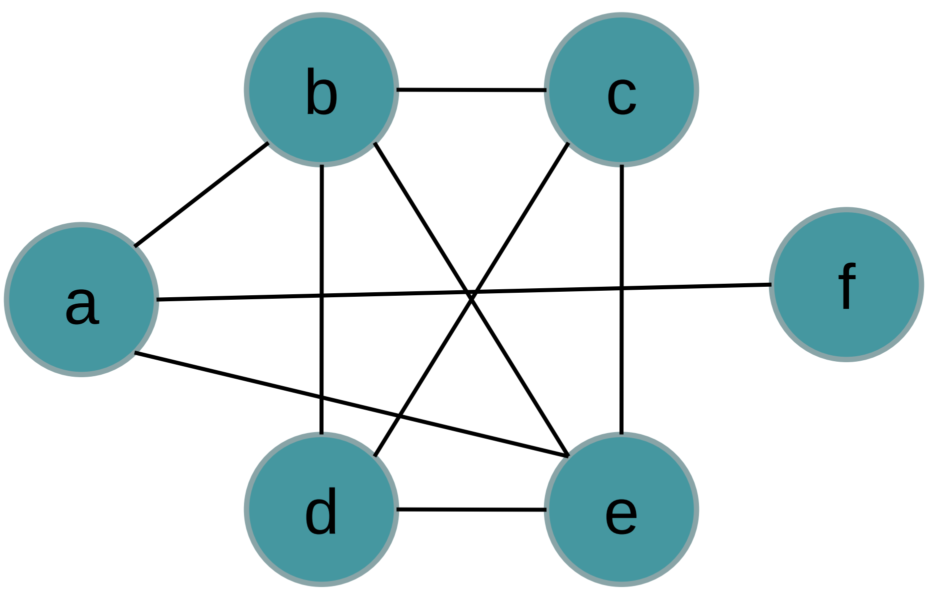
$$\boxed{\begin{array}{l}(y_1 \vee p \vee q) \wedge (y_1 \vee \neg p \vee \neg q) \wedge (y_1 \vee p \vee \neg q) \wedge (y_1 \vee \neg p \vee q) \wedge \\ (\neg y_1 \vee \neg x_1 \vee y_2) \wedge (\neg y_1 \vee x_1 \vee \neg y_2) \wedge (\neg y_1 \vee x_1 \vee y_2) \wedge (y_1 \vee \neg x_1 \vee \neg y_2) \wedge \ldots\end{array}}$$

CLIQUE

Um *Clique* em um grafo não direcionado $G = (V, A)$ é um subconjunto de vértices $V' \subseteq V$, onde cada vértice está conectado por uma aresta. Ou seja, um subgrafo completo.

Versão de otimização: Encontrar o maior *Clique* possível.

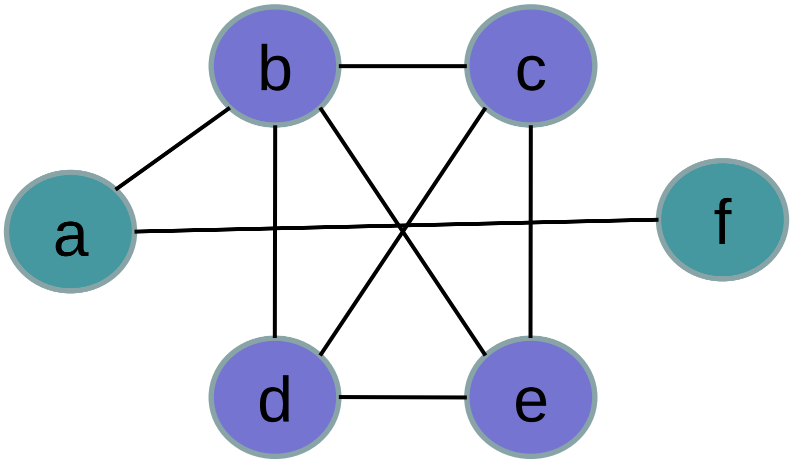
Versão de decisão: Existe um *Clique* de tamanho $\geq k$?



CLIQUE

CLIQUE é *NP-Completo*?

- **Passo 1**: CLIQUE $\in$ *NP*.
- **Passo 2**: 3-CNF-SAT $\leq_p$ CLIQUE.



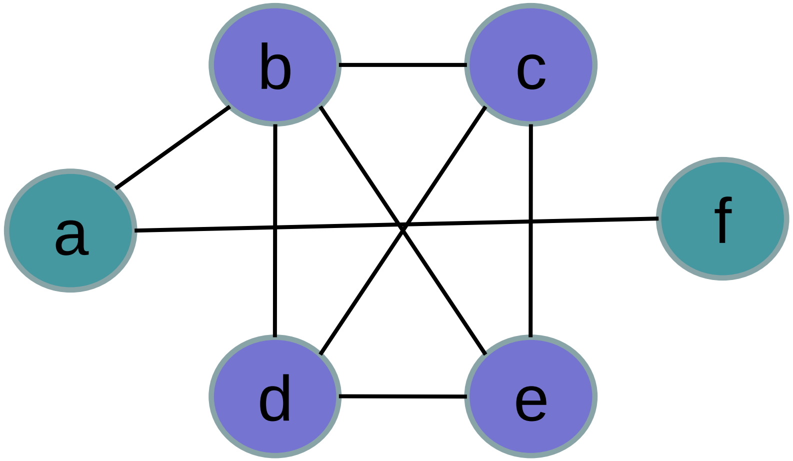
CLIQUE

Passo 1: Clique $\in$ *NP*

V = { a, b, c, d, e, f }

A = { (a,b), (a,f), (b,c), (b,d), (b,e), (c,d), (c,e), (d,e) }

V' = { b, c, d, e }



Dado um grafo $G = (V, A)$, a solução (**certificado**) V' e k, verificar se $|V| \geq k$ em tempo polinomial

Para cada $u \in V'$
Para cada $v \in V'$
Se $u \neq v$ então verificar se $(u, v) \in A$

Complexidade?

3-CNF-SAT $\leq_p$ CLIQUE

- **Passo 2**: 3-CNF-SAT $\leq_p$ CLIQUE.

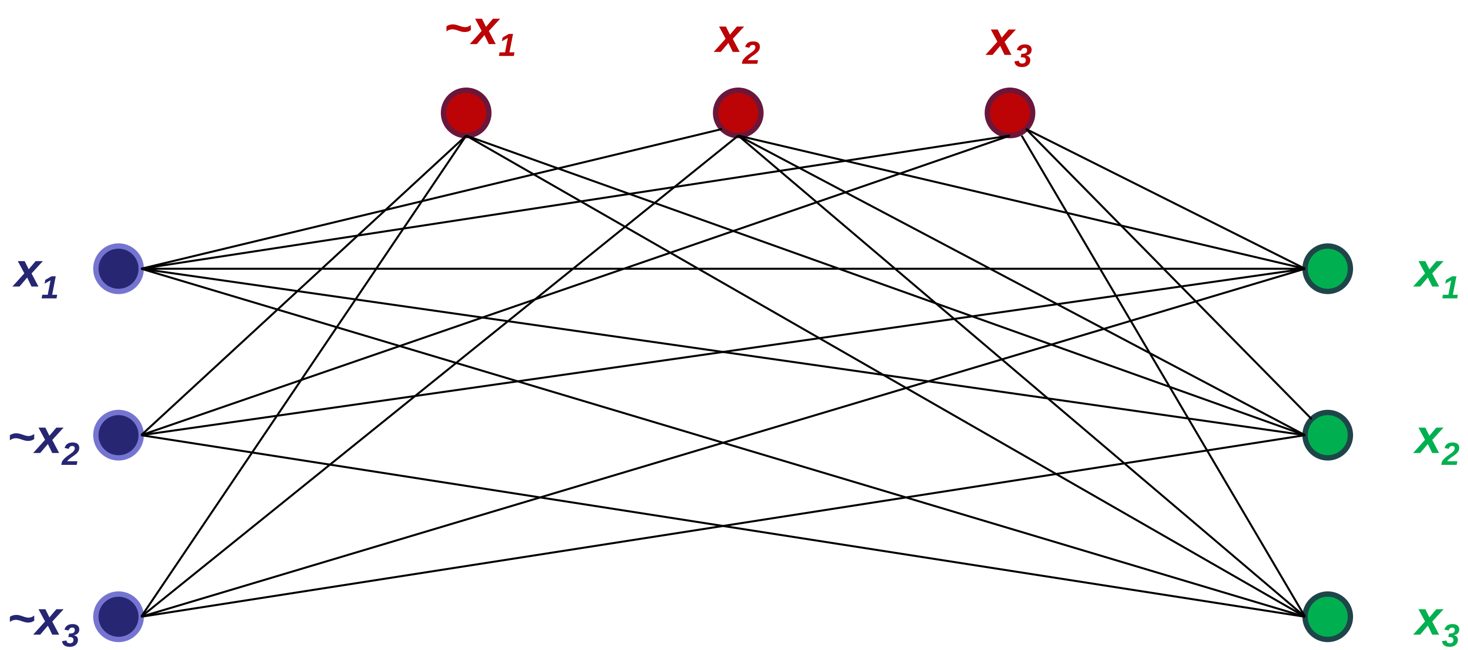
Dada uma instancia ϕ do problema 3-CNF-SAT converteremos esta para um grafo G que terá 3k vértices, onde k é o número de cláusulas de ϕ.

- ***u*** e ***v*** são vértices que correspondem a literais em diferentes cláusulas;
- Todos os vértices são ligados por arestas, com exceção:
  - se ***u*** e ***v*** pertencem a mesma cláusula, então não há ligação;
  - se ***u*** corresponde a um literal ***x*** e ***v*** corresponde ao literal **~*x*,** então não há ligação entre esses dois vértices;

3-CNF-SAT $\leq_p$ CLIQUE

- **Passo 2**: 3-CNF-SAT $\leq_p$ CLIQUE.

$$\phi = (x_1 \vee \sim x_2 \vee \sim x_3) \wedge (\sim x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

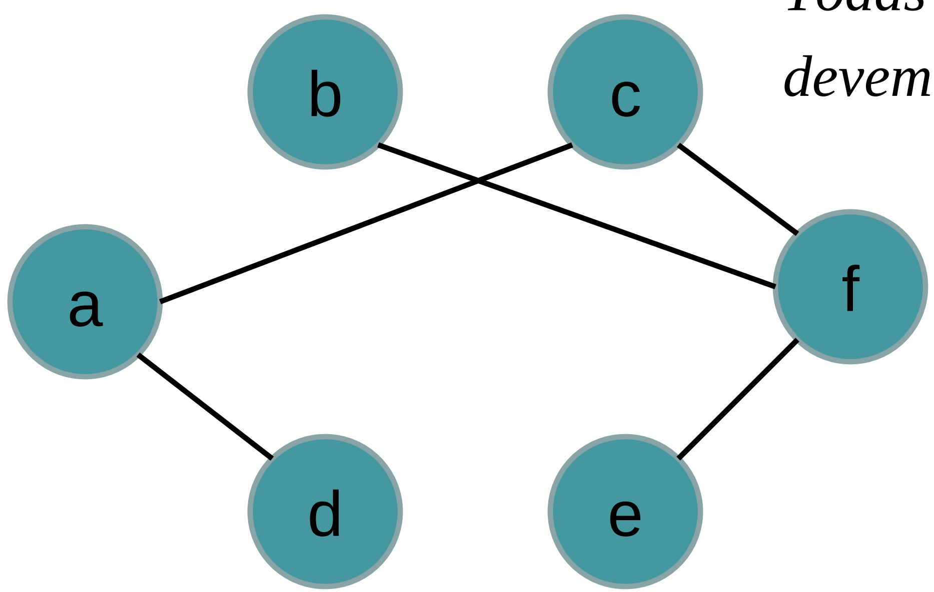


ϕ é satisfazível $\leftrightarrow$ G possui um clique $\geq$ k

Cobertura de Vértices
(VERTEX-COVER)

Uma *Cobertura de Vértices* de um grafo não orientado $G = (V, A)$ é um subconjunto $V' \subseteq V$ tal que se $(u, v) \in A$, então $u \in V'$ ou $v \in V'$.

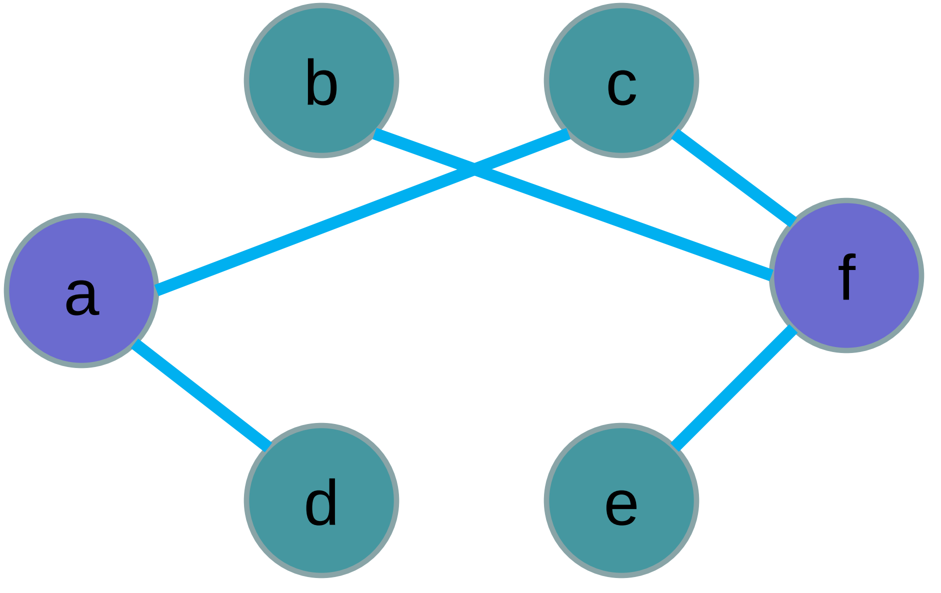
Todas as arestas devem ser observadas!



Cobertura de Vértices
(VERTEX-COVER)

Versão de otimização: Encontrar menor Cobertura de Vértices.

Versão de decisão: Existe uma cobertura de tamanho *k*?



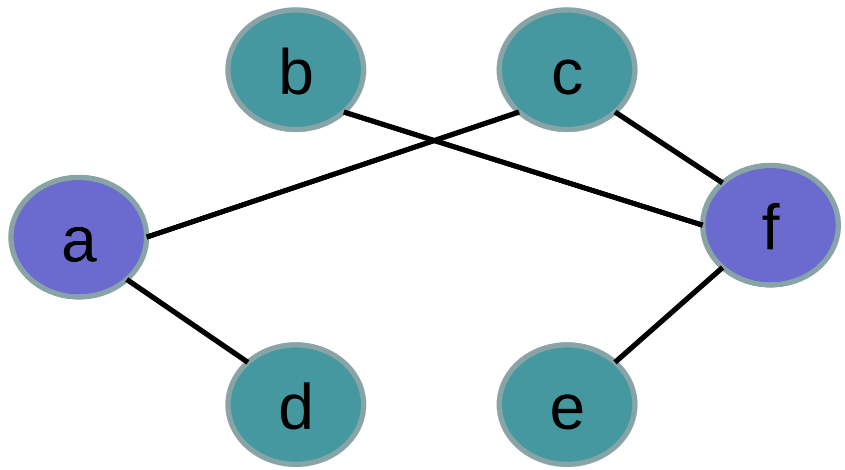
Cobertura de Vértices
(VERTEX-COVER)

Passo 1: Cobertura de Vértices $\in$ NP.

V = { a, b, c, d, e, f }

A = { (a,c), (a,d), (b,f), (c,f), (f,e) }

V’ = { a, f }



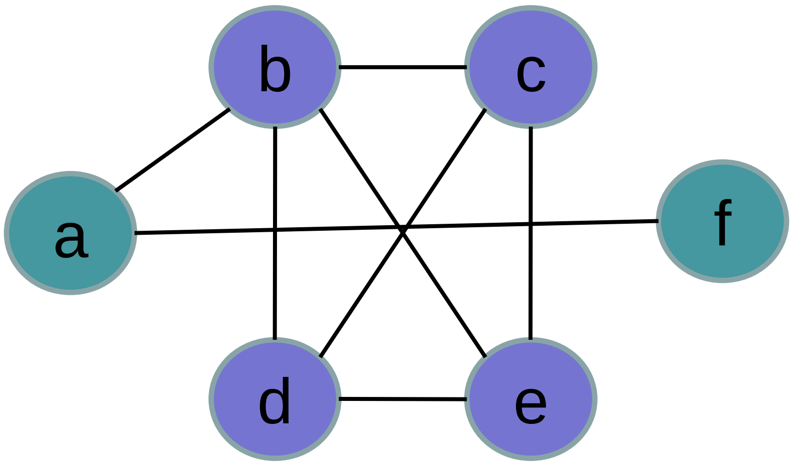
Dado um grafo $G = (V, A)$ e a solução (**certificado**) V'
verificar se $|V| \geq k$ em tempo polinomial

Para cada $(u, v) \in A$
Verificar se $u \in V'$ ou $v \in V'$

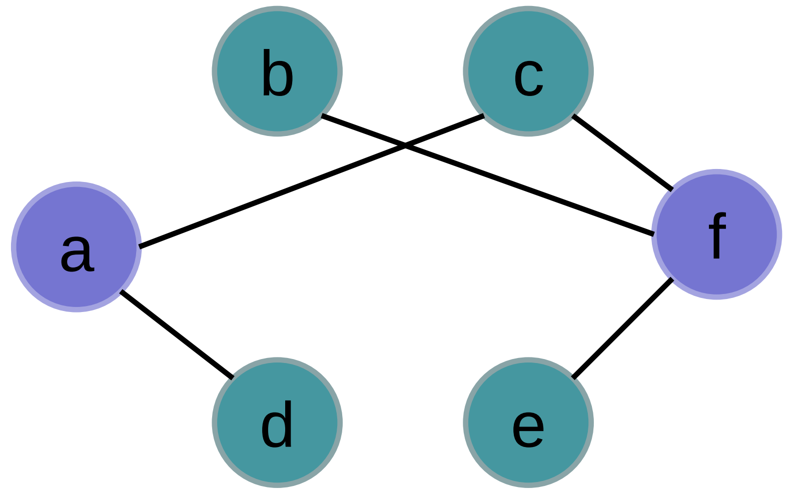
Complexidade?

CLIQUE $\leq_p$ VERTEX-COVER

- **Passo 2**: CLIQUE $\leq_p$ VERTEX-COVER



CLIQUE
Entrada (G, k), onde $G = (V, A)$

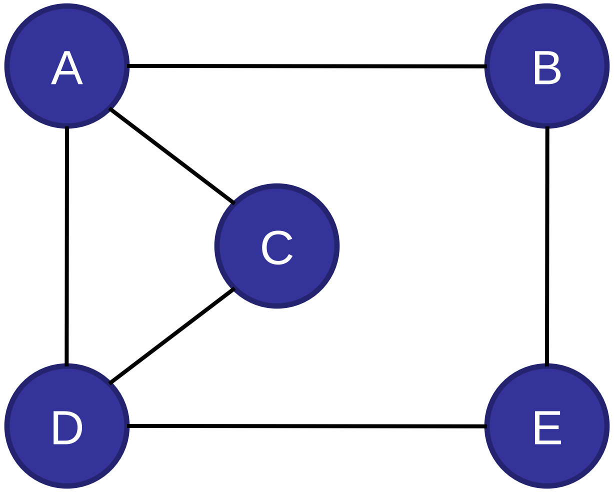


VERTEX-COVER
Entrada $(\overline{G}, |V| - k)$

Ciclo Hamiltoniano

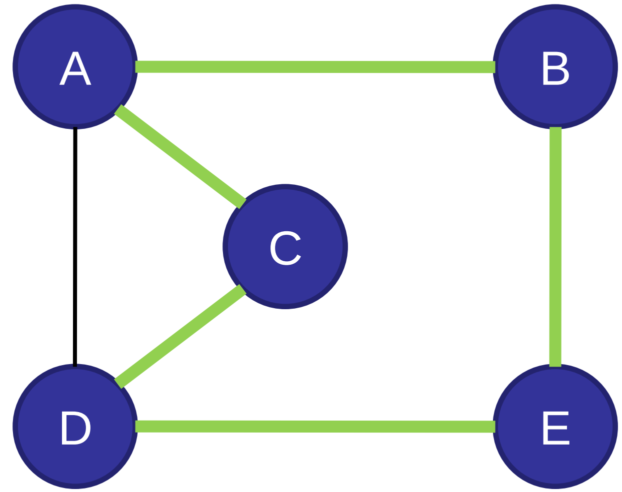
Um ***Ciclo Hamiltoniano*** em um grafo não orientado é um caminho que passa por cada vértice do grafo exatamente uma vez e retorna ao vértice inicial.

Versão de decisão: um grafo *G* possui um ciclo Hamiltoniano?



Ciclo Hamiltoniano

Passo 1: Ciclo Hamiltoniano $\in$ *NP*



V = { a, b, c, d, e }

A = { (a,b), (a,c), (a,d), (b,e), (c,e), (d,e) }

V' = { a, b, e, d, c }

Dado um grafo $G = (V, A)$ e a solução (**certificado**) V' verificar se V' é um ciclo Hamiltoniano em tempo polinomial

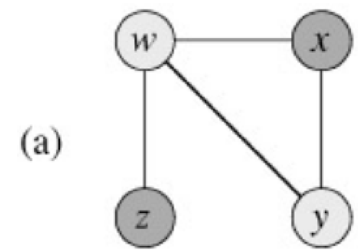
```
Para cada v ∈ V: viz[v] = não marcado
Para cada v’ ∈ V’:
    Se viz[v’]==marcado: retorne falso
    Senão: viz[v’] = marcado
Se todo x ∈ viz está marcado: retorne verdade
Senão: retorne falso
```

Complexidade?

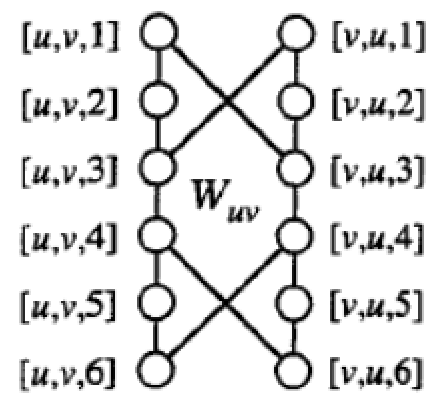
Ciclo Hamiltoniano

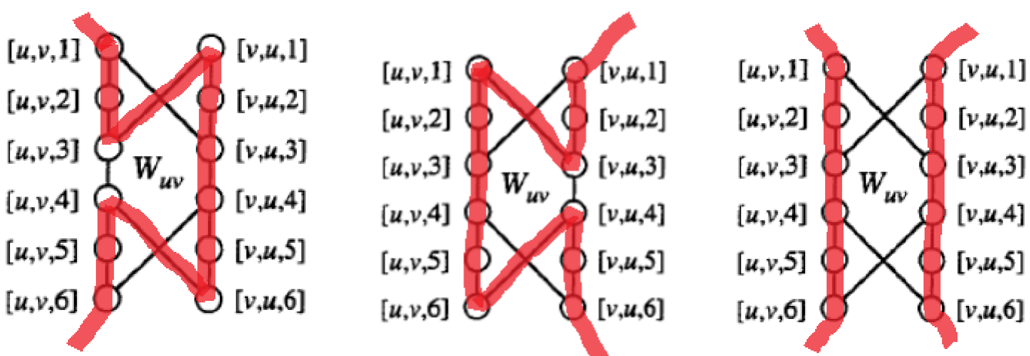
- **Passo 2**: VERTEX-COVER $\leq_p$ CICLO HAMILTONIANO

Dado um grafo instância do problema de Cobertura de vértices G = (V , E), devemos:



- criar ***k*** vértices seletores, onde ***k*** é o número de vértices que pertencem a solução da cobertura;
- criar ***E*** dispositivos, totalizando ***E* * 12** novos vértices e ***E * 14*** arestas;





Ciclo Hamiltoniano

- **Passo 2**: VERTEX-COVER $\leq_p$ CICLO HAMILTONIANO

- criar uma lista com as adjacências de cada nó (para formar um caminho entre todas as coberturas de um vértices):

  u: $u_1, u_2, \ldots u_{grau(u)}$

  v: $v_1, v_2, \ldots v_{grau(v)}$

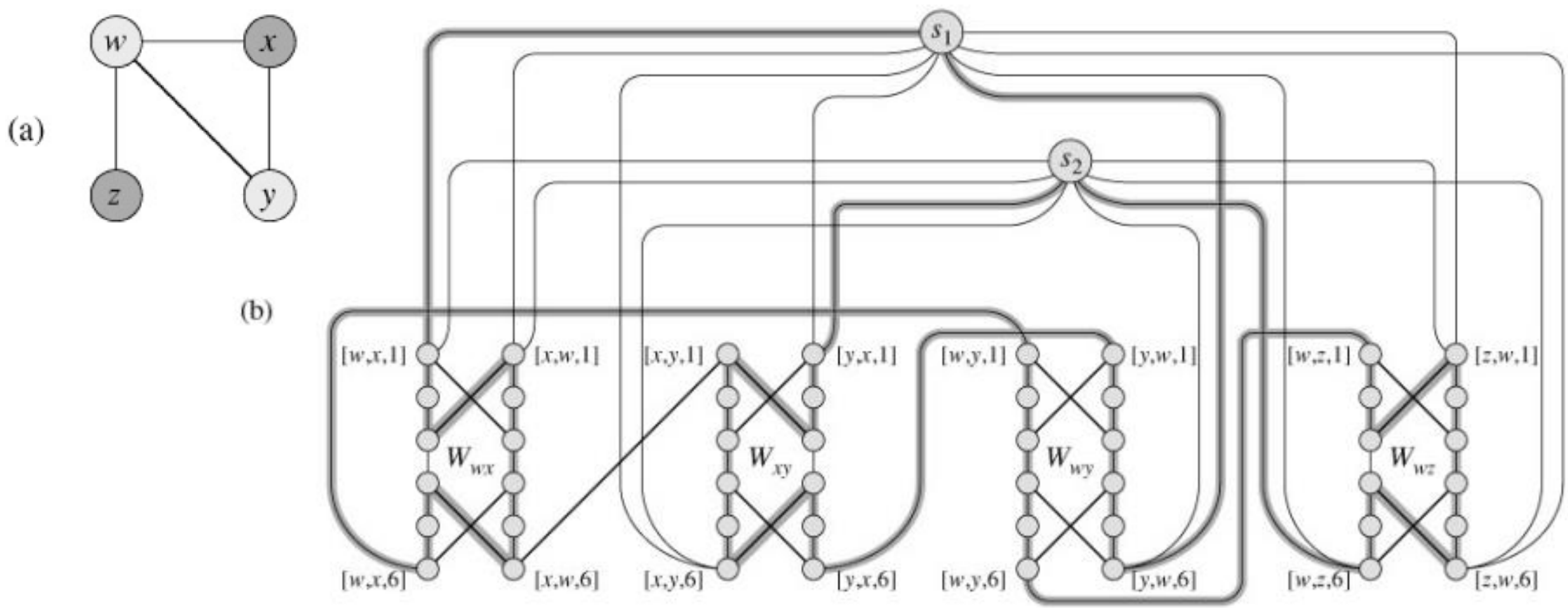
  - adicionar arestas para unir pares de dispositivos:
    $\{([u,u_i,6],[u,u_{i+1},1]), \ldots \}$
  - criar arestas para unir o primeiro $[u, u_1, 1]$ e o último vértice $[u, u_{grau(u)}, 6]$ de cada um desses caminhos a cada vértice seletor.

    $\{(sj, [u, u_1, 1]) : u \in V \text{ e } 1 \leq j \leq k\}$

    $\{(sj, [u, u_{grau(u)}, 6]) : u \in V \text{ e } 1 \leq j \leq k\}$

Ciclo Hamiltoniano

- **Passo 2**: VERTEX-COVER $\leq_n$ CICLO HAMILTONIANO



s1 → W_{wx} → W_{wy}* → W_{wz} → s2 → W_{yx} → W_{yw}* → s1

O caminho 3 entre dispositivos (*) só ocorre em arestas compartilhadas por vértices que fazem parte da solução da cobertura de vértices

Ciclo Hamiltoniano

- **Passo 2**: VERTEX-COVER $\leq_p$ CICLO HAMILTONIANO

Importante: note que o novo grafo G' = (V', E')

$| V' | = 12\ |E| + k$

$| V' | \leq 12\ |E| + |V|$

Instância cresceu apenas em tamanho polinomial

$| E' | = 14\ |E| + (\ 2|E| - |V|\ ) + (2k\ |V|)$

$| E' | = 16\ |E| + (\ 2k - 1\ )\ |V|$

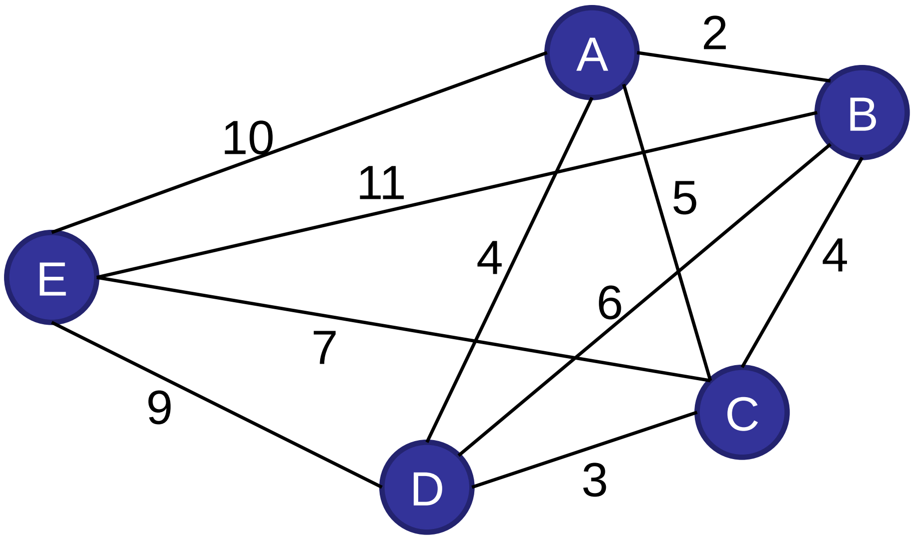
$| E' | \leq 16\ |E| + (\ 2|V| - 1\ )\ |V|$

Problema do Caixeiro Viajante

Um vendedor deseja visitar n cidades e retornar a cidade de origem. Dado um grafo não orientado completo com n vértices, onde existe um custo $c(i, j)$ (associado a cada aresta) para viajar da cidade i a cidade j.

Otimização: Qual é o menor caminho para o vendedor?

Decisão: Existe um caminho para o vendedor com custo máximo igual a k?



Problema do Caixeiro Viajante

Passo 1: Caixeiro Viajante $\in$ *NP*

Dado um grafo $G = (V, A)$, a solução (**certificado**) V' e o custo máximo k, verificar se V' é um caminho válido do Caixeiro com custo menor ou igual a k em tempo polinomial

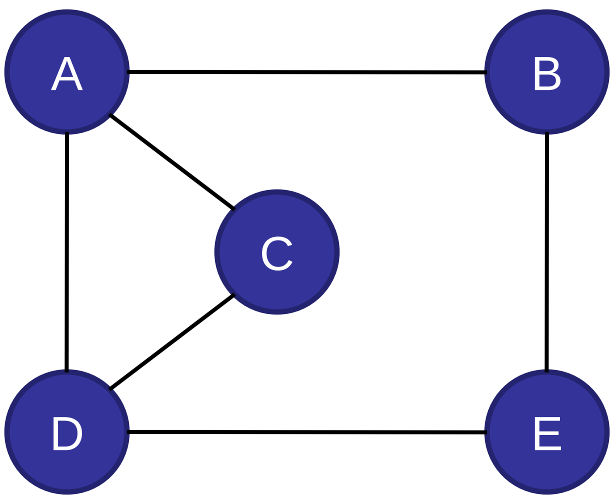
```
Para cada v ∈ V: viz[v] = não marcado
custo = 0, n = |V'|
Se V'[1] ≠ V'[n]: retorne 0
Para i=1 até n-1:
     Se viz[ V[i] ]==marcado: retorne 0
     Senão: viz[ V[i] ] = marcado
            custo = c[ V[i] ][ V[i+1] ]
Se todo x ∈ viz está marcado E custo<=k: retorne 1
Senão: retorne 0
```

Complexidade?

Redução do Problema do Ciclo Hamiltoniano ao Problema do Caixeiro Viajante

- **Passo 2**: CICLO HAMILTON $\leq_p$ CAIXEIRO

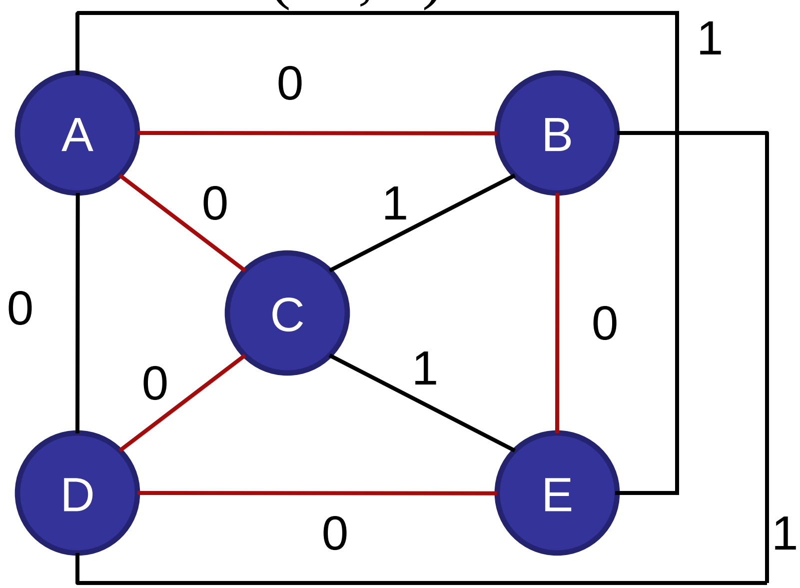
G



$\leq_p$

Redução em tempo polinomial

$(G', 0)$



para cada vértice i

para cada vértice j

se $(i, j) \in H$ então $c(i, j) \leftarrow 0$

senão $c(i, j) \leftarrow 1$

SUBSET-SUM

Dado um conjunto finito de inteiros positivos *S* e um inteiro $t > 0$, determinar se existe um subconjunto $S' \subseteq S$ onde o somatório dos elementos de *S'* é igual a *t*.

$$\sum_{i=1}^{n} s_i' = t$$

Exemplo: S = {1, 2, 7, 14, 49, 98, 343, 686, 2.409, 2.793, 16.808, 17.206, 17.705, 117.993 }

t = 138.457

SUBSET-SUM

Exemplo:

S = {1, 2, 7, 14, 49, 98, 343, 686, 2.409, 2.793, 16.808, 17.206, 17.705, 117.993 }

t = 138.457

S’ = { 1, 2 , 7, 98, 343, 686, 2.409, 17.206, 117.705 }

SUBSET-SUM

Passo 1: Subset-Sum $\in$ *NP*

Dado um conjunto de números inteiros S, o valor t objetivo e a solução (**certificado**) S', verificar se S' é uma solução do problema em tempo polinomial.

```
soma = 0
Para cada s' ∈ S':
      Se s' ∉ S: retorne 0
      soma = soma + s'
Se soma == t: retorne 1
Senão: retorne 0
```

Complexidade?

Passo 2: 3-CNF-SAT $\leq_p$ SUBSET-SUM

Dada uma fórmula ϕ instância de 3-CNF-SAT, devemos:

- Criar dois números para cada variável x_i *em* ϕ: v_i *e* v'_i
- Criar dois números para cada cláusula C_j em ϕ: s_j e s'_j

Cada número criado terá **n + k** dígitos, onde n é o número de variáveis e k é o número de cláusulas

O valor **t** terá um valor 1 para cada dígito identificado por variável e 4 em cada dígito identificado por uma cláusula

SUBSET-SUM

Passo 2: 3-CNF-SAT $\leq_p$ SUBSET-SUM

- Para cada variável v_i *e* v'_i colocamos o valor 1 no dígito identificado por x_i e 0 nos outros dígitos;
- Se o literal x_i aparece na cláusula C_j, então o dígito identificado por C_j em v_i contém valor 1;
- Se o literal $\sim x_i$ aparece na cláusula C_j, então o dígito identificado por C_j em v_i contém valor 0;
- Para cada s_j e s'_j colocamos valor 0 em todos os dígitos, com duas exceções:

  em s_j colocamos 1 no dígito C_j

  em s'_j colocamos 2 no dígito C_j

3-CNF-SAT $\leq_p$ SUBSET-SUM

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

$$\downarrow$$

S = { 1, 2, 10, 20, 100, 111, 1.000, 1.011, 10.001,10.010 }

t = 11144

S' = { 10001, 1011, 111, 20, 1 }
{ v_1,v_2,v'_3,s'_1,s_2 }

X_1 = V, X_2 = V, X_3 = F

	x_1	x_2	x_3	C_1	C_2
v_1	1	0	0	0	1
v'_1	1	0	0	1	0
v_2	0	1	0	1	1
v'_2	0	1	0	0	0
v_3	0	0	1	0	0
v'_3	0	0	1	1	1
s_1	0	0	0	1	0
s'_1	0	0	0	2	0
s_2	0	0	0	0	1
s'_2	0	0	0	0	2
t	1	1	1	4	4

Passo 2: 3-CNF-SAT $\leq_p$ SUBSET-SUM

Note que a maior soma de cada coluna (dígito) é no máximo 6.

Assim, para esta conversão devemos usar uma base ≥ 7.

No exemplo usamos números na base 10.

A redução de 3-CNF-SAT para SUBSET-SUM acontece em tempo polinomial, **MAS**, o valor **t** objetivo do SUBSET-SUM aumenta de forma exponencial em relação ao número de variáveis e cláusulas da fórmula 3-CNF-SAT.

Programação Dinâmica (*Subset-Sum*)

Dado um conjunto de inteiros positivos, representados como um arranjo $S[1..n]$, e um inteiro t, existe algum subconjunto de S tal que a soma de seus elementos seja t.

$$SubsetS(i,t) = \begin{cases} Verdade & \text{se } t = 0 \\ Falsidade & \text{se } t < 0 \vee i > n \\ SubsetS(i+1,t) \vee SubsetS(i+1, t - x[i]) & \end{cases}$$

Programação Dinâmica
(*Subset-Sum*)

Exemplo: $x = \{2, 3, 5\}$ e $t = 8$.

$$SubsetS(i,t) = \begin{cases} Verdade & \text{se } t = 0 \\ Falsidade & \text{se } t < 0 \vee i > n \\ SubsetS(i+1,t) \vee SubsetS(i+1,t-x[i]) & \end{cases}$$

Programação Dinâmica (*Subset-Sum*)

SubsetSum ($x[1..n]$, t)

$S[n + 1, 0] \leftarrow$ Verdade

para $j \leftarrow 1$ até t

$S[n + 1, j] \leftarrow$ Falsidade

para $i \leftarrow n$ até 1

$S[i, 0] \leftarrow$ Verdade

para $j \leftarrow 1$ até $x[i] - 1$

$S[i, j] \leftarrow S[i + 1, j]$

para $j \leftarrow x[i]$ até t

$S[i, j] \leftarrow S[i + 1, j] \vee S[i + 1, j - x[i]]$

retorne $S[1,t]$

Programação Dinâmica (*Subset-Sum*)

Exemplo: $x = \{1, 3, 5, 7\}$ e $t = 10$.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	V	V	F	V	V	V	V	V	V	V	V
2	V	F	F	V	F	V	F	V	V	F	V
3	V	F	F	F	F	V	F	V	F	F	F
4	V	F	F	F	F	F	F	V	F	F	F
5	V	F	F	F	F	F	F	F	F	F	F

Algoritmos que Executam em Tempo Pseudo-Polinomial

Usando programação dinâmica podemos implementar um algoritmo pseudo-polinomial com complexidade $O(nt)$, onde ***n*** é o número de elementos no conjunto e ***t*** o valor do somatório que se deseja alcançar.

Se restringirmos nossa atenção a instâncias do problema onde o valor de t é limitado por um polinômio existe uma solução eficiente.

Algoritmos que Executam em Tempo Pseudo-Polinomial

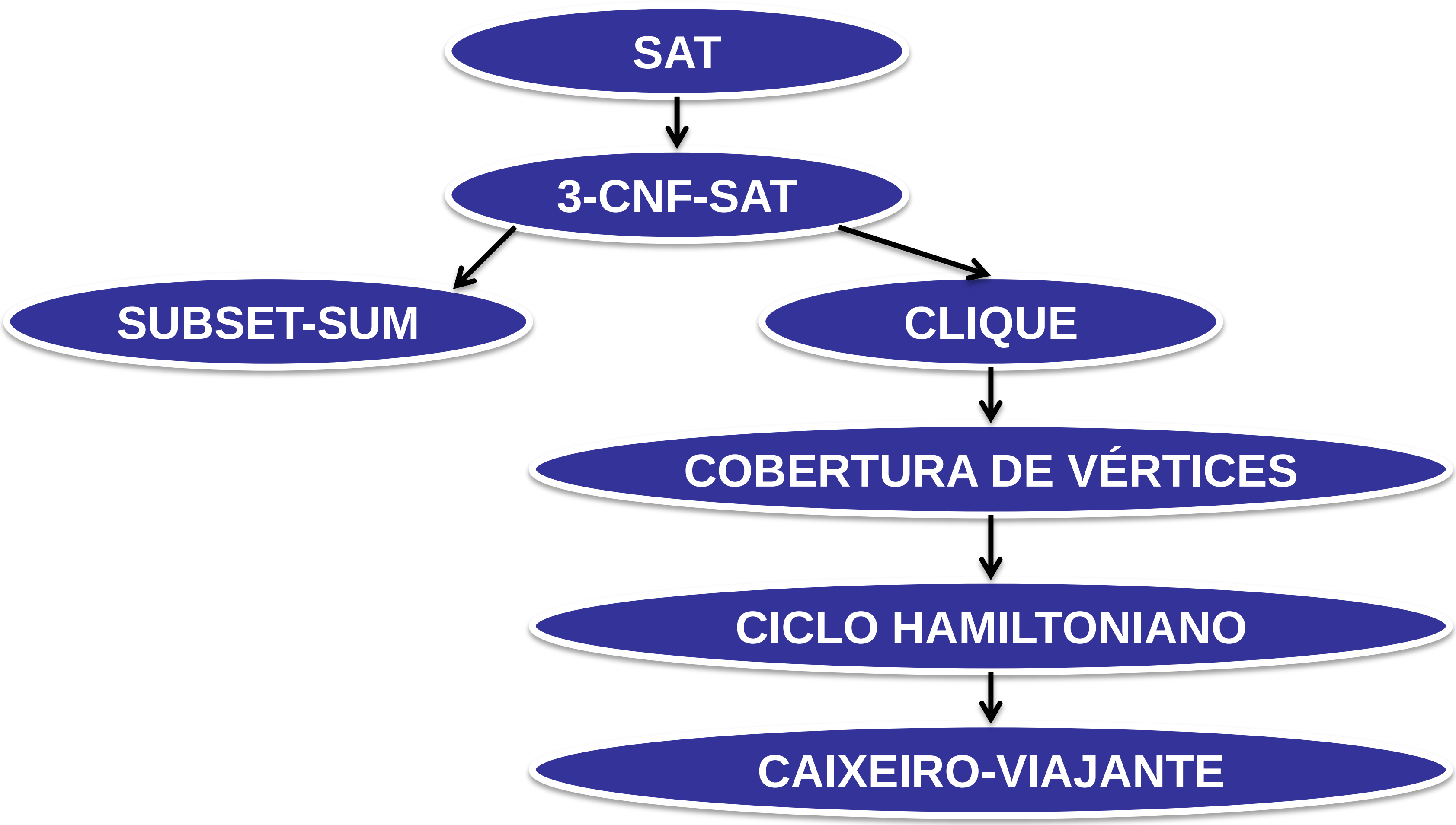
Essa restrição pode ser bastante razoável na prática:

- Problemas onde é impossível a ocorrência de números muito grandes (*e.g.* problemas de escalonamento).
- Problemas onde o tamanho do número possa ser restrito ao tamanho da palavra do processador.

Note que esse não é o caso da redução do 3-CNF-SAT ao SUBSET-SUM, onde o valor de t cresce exponencialmente ao número de variáveis e cláusulas presentes na fórmula booleana.

Reduções

Resumindo, quais reduções de problemas foram feitas:



Exercícios

Lista de Exercícios

Referências

Algoritmos. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein. Campus.

Algorithms. Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. McGraw Hill.

Concrete Mathematics: A Foundation for Computer Science (2nd Edition). Ronald L. Graham, Donald E. Knuth, Oren Patashnik. Addison Wesley.

M. R. Garey and D. S. Johnson. 1978. *"Strong" NP-Completeness Results: Motivation, Examples, and Implications*. J. ACM 25, 3 (July 1978)