UDESC

Projeto e Análise de Algoritmos

Cristiano Damiani Vasconcellos
cristiano.vasconcellos@udesc.br

UDESC

Bibliografia

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.

UDESC

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.

UDESC

Análise de Algoritmos

Contar o número de instruções que são executadas pelo algoritmo. Por exemplo: load, store, add, sub, div, mul, call, ret, cmp, jump, etc.

UDESC

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, etc.

UDESC

Programa

- Notação Assintótica.
- Complexidade de tempo e espaço.
- Relações de Recorrência.
- Somatórios.
- Divisão e Conquista.
- Algoritmos Gulosos.
- Programação Dinâmica.
- Tratabilidade (classes *P*, *NP* e *NP-Completo*).

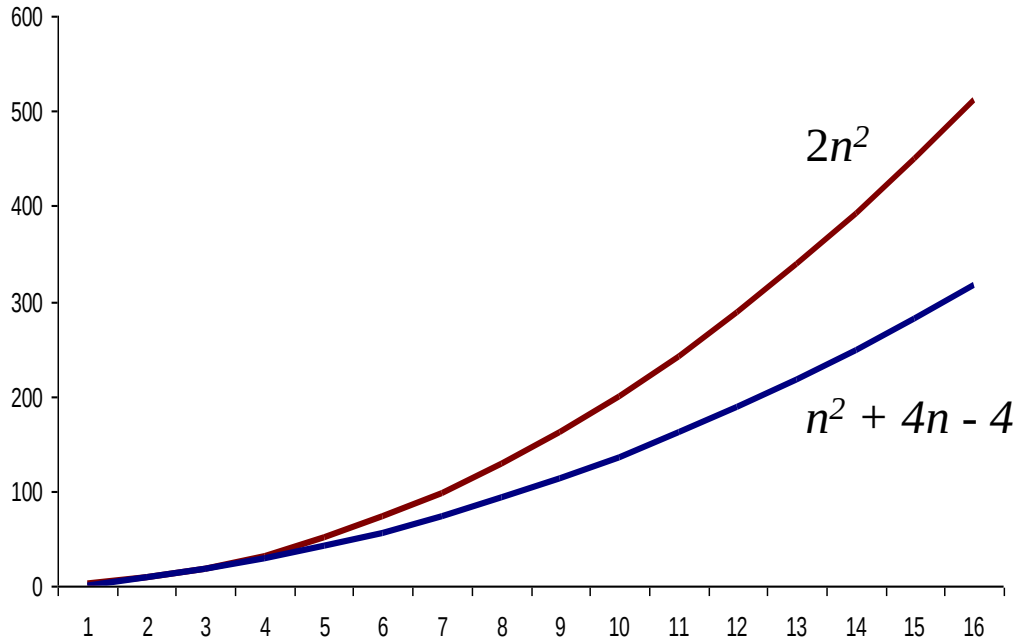
UDESC

Crescimento de Funções

Para simplificar o processo usamos uma abstração que ignora o custo de cada instrução, que é constante. Concentrando a análise no crescimento do tempo de execução (ou de outro recurso) em relação ao crescimento da entrada.

Notação Assintótica
(Notação O grande – Limite Superior)

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



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

UDESC

Algumas Operações com Notação *O*

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

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

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

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

UDESC

Complexidade de Tempo

```
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)
}
```

Em que situações ocorrem o melhor caso, pior caso e o caso médio?

Complexidade de Tempo

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:

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

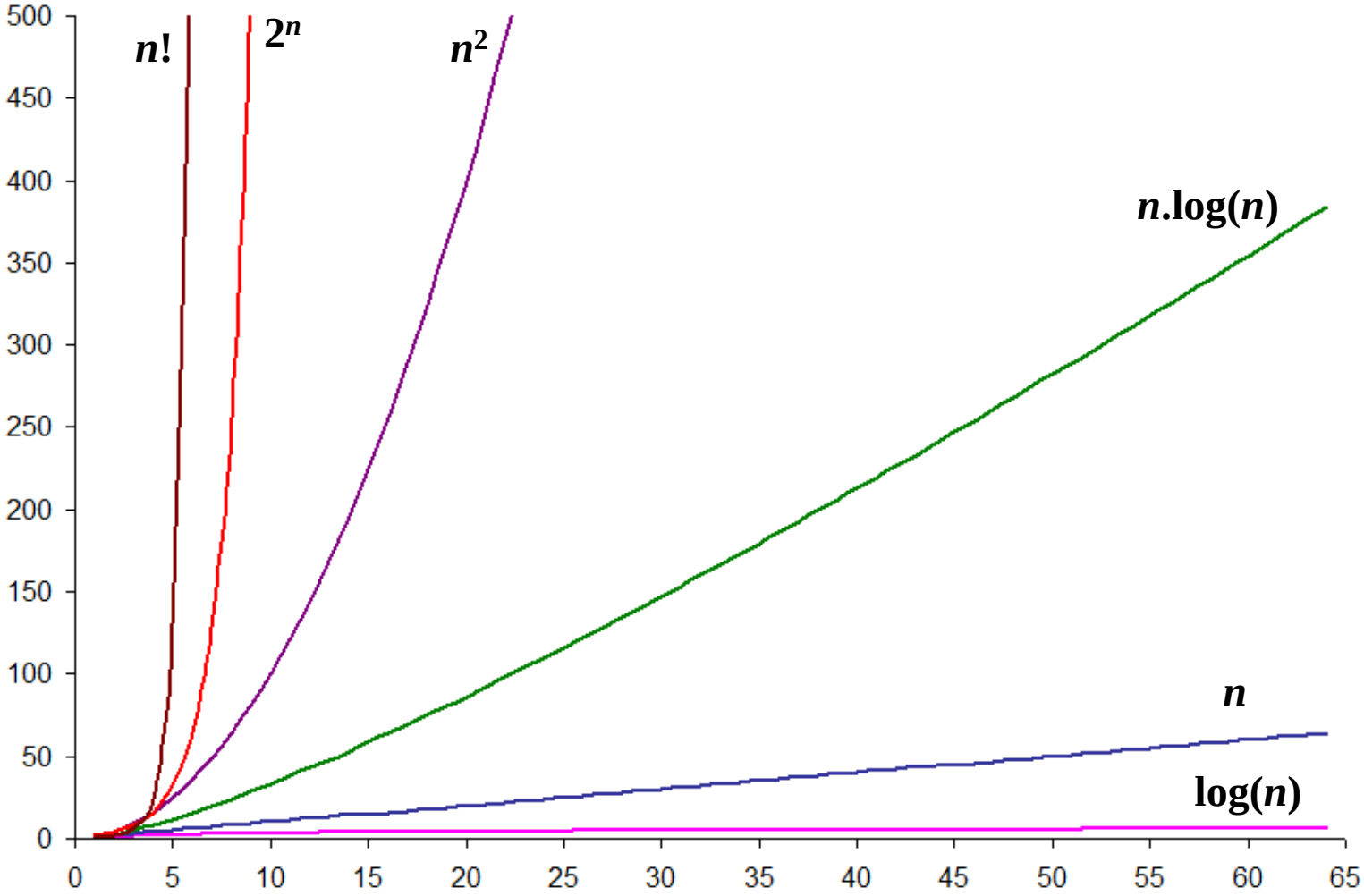
Considerando que a probabilidade de procurar qualquer registro é a mesma probabilidade, temos

$p_i = 1/n$, para todo i.

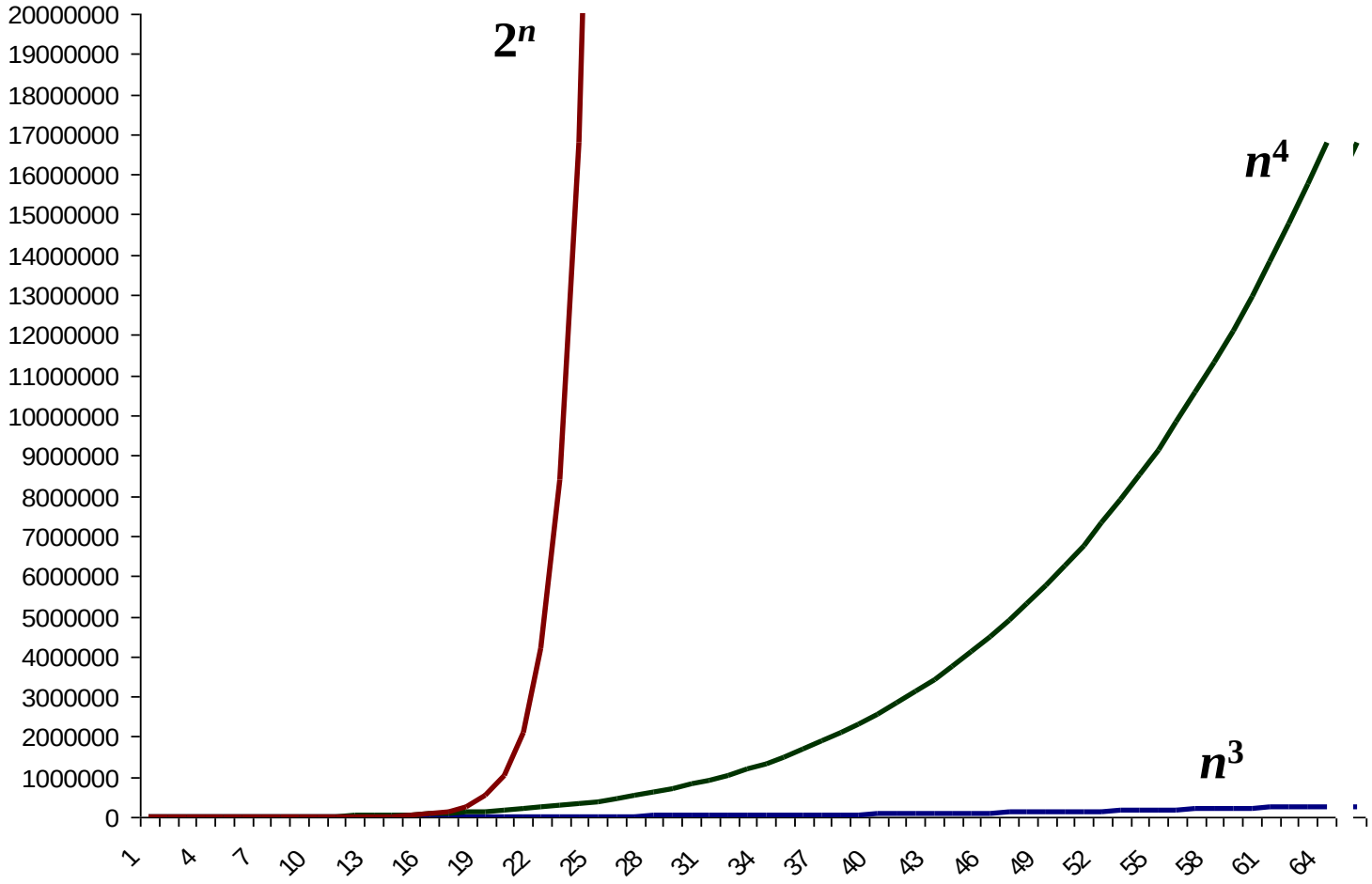
$$f(n) = \frac{1}{n}(1+2+...+n) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{(n+1)}{2}$$

UDESC

Crescimento de Funções



Crescimento de Funções



UDESC

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 $0 < \varepsilon < 1 < c$.

UDESC

Exemplo (Ordenação por Inserção)

```
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;
    }
}
```

UDESC

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;
            }

        }
    }
}
```

UDESC

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);
}
```

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

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

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

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

...

$T(n/2^{l-1}) = T(n/2^l) + 1$

$$\frac{n}{2^l} = 1$$

$$n = 2^l$$

$$l = log_2\, n$$

$O(log\ n)$

$$T(n) = \underbrace{1 + 1 + \ldots + 1}_{log_2\, n\ vezes} + T(1)$$

Mudança de base:

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

UDESC

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;
}
```

UDESC

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.

UDESC

Exemplo (Merge Sort)

```
int mergeSort(int *v, int p, int r)
{
    int q;
    if (p<r)
    {
        q = (p+r)/2;
        mergeSort(v, p, q);
        mergeSort(v, q+1, r);
        merge(v, p, q, r);
    }
}
void mSort(int *v, int n)
{
    mergeSort(v, 0, n - 1);
}
```

Exemplo (Merge Sort)

```
int mergeSort(int *v, int p, int r)
{
    int q;
    if (p<r)
    {
        q = (p+r)/2;
        mergeSort(v, p, q);
        mergeSort(v, q+1, r);
        merge(v, p, q, r);
    }
}
void mSort(int *v, int n)
{
    mergeSort(v, 0, n-1);
}
```

Relação de Recorrência:

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

$T(1) = O(1)$

UDESC

Dividir e Conquistar

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

UDESC

Quick Sort

```
void quickSort(int *v, int e, int d)
{
        int q;
        if (e < d)
        {
                q = particao(v, e, d);
                quickSort(v, e, q);
                quickSort(v, q + 1, d);
        }
}
```

UDESC

Quick Sort

```
int particao(int *v, int e, int d)
{
   int i, j, pivo, aux;
   pivo = v[e];
   i = e - 1;  j = d + 1;
   while (i < j)
   {
       do  j--;   while (v[j] > pivo);
       do  i++; while (v[i] < pivo);
       if (i < j)
           swap (&v[i], &v[j]);
    }
    return j;
}
```

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 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

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

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

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

$$\log_c (a/b) = \log_c a - \log_c b$$

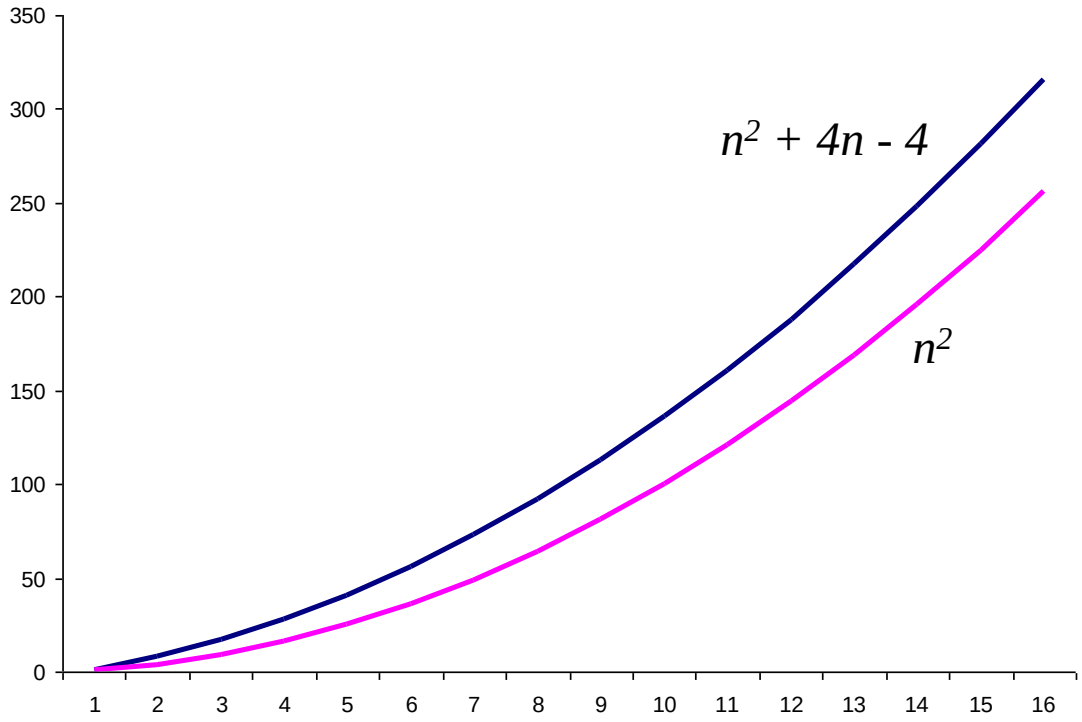
$$\log_b a^c = c \log_b a$$

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

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

Notação Assintótica
Limite Inferior (Notação Ω)

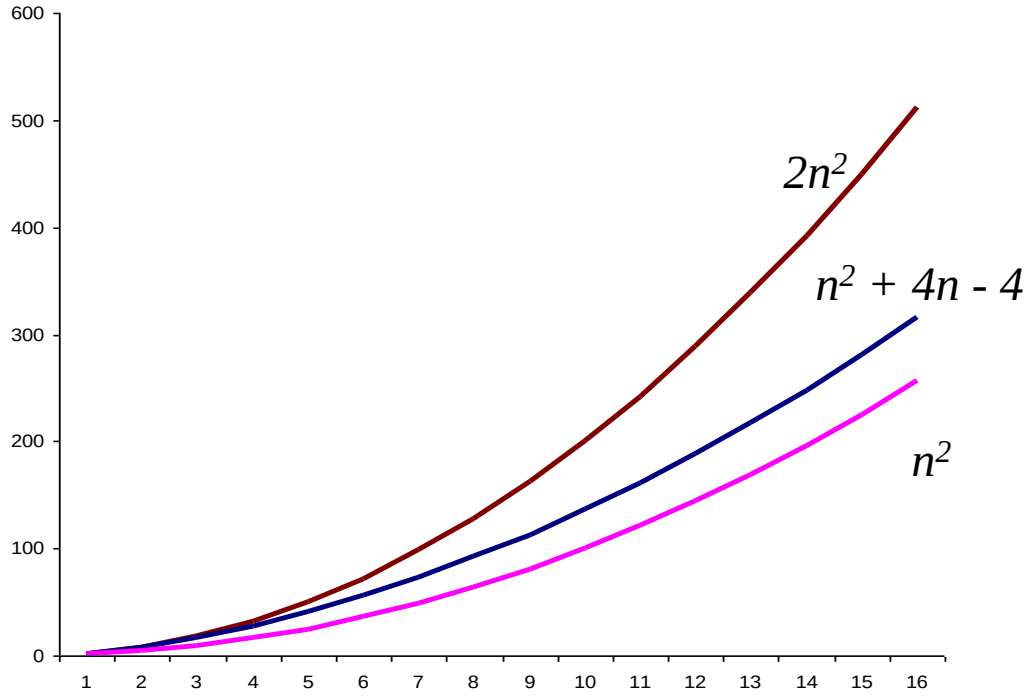
Uma função *f(n)* é o limite inferior de outra função *g(n)* se existem duas constantes positivas *c* e n_0 tais que, para $n > n_0$, temos $|g(n)| \geq c.|f(n)|$, $g(n) = \Omega(f(n))$.



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

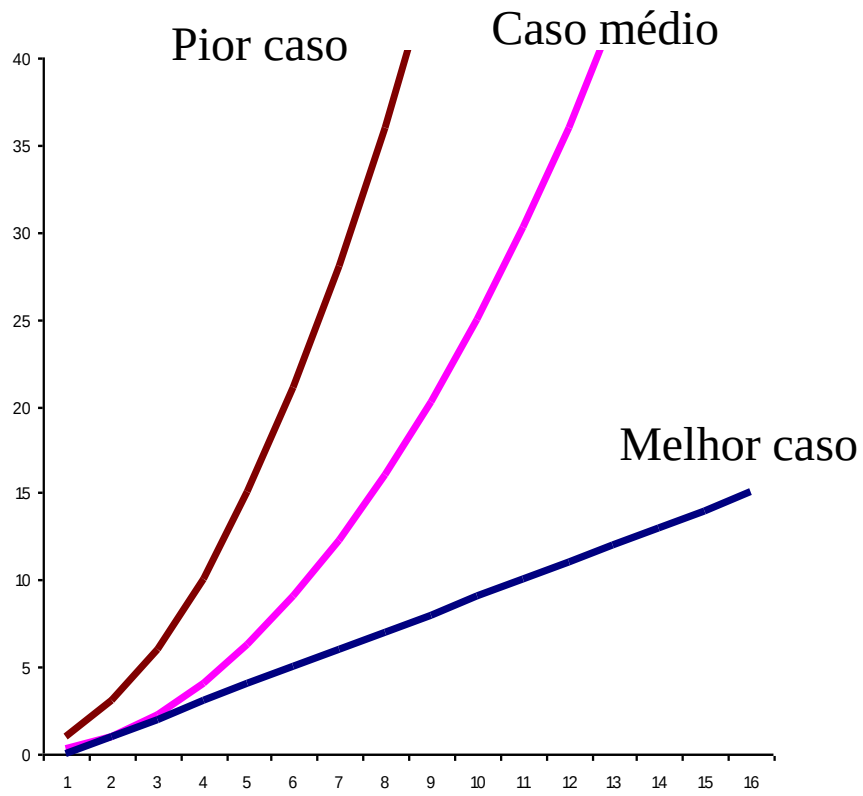
Notação Assintótica
Limite Firme (Notação Θ)

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



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

Ordenação por Inserção



Pior Caso: $(n^2\text{-}n)/2$

$\underline{\boldsymbol{O(n^2)}}$, $\Theta(n^2)$, $\Omega(n^2)$.

Caso Médio: $n^2/4$

$O(n^2)$, $\Theta(n^2)$, $\Omega(n^2)$.

Melhor Caso: n-1

$O(n)$, $\Theta(n)$, $\underline{\boldsymbol{\Omega(n)}}$.

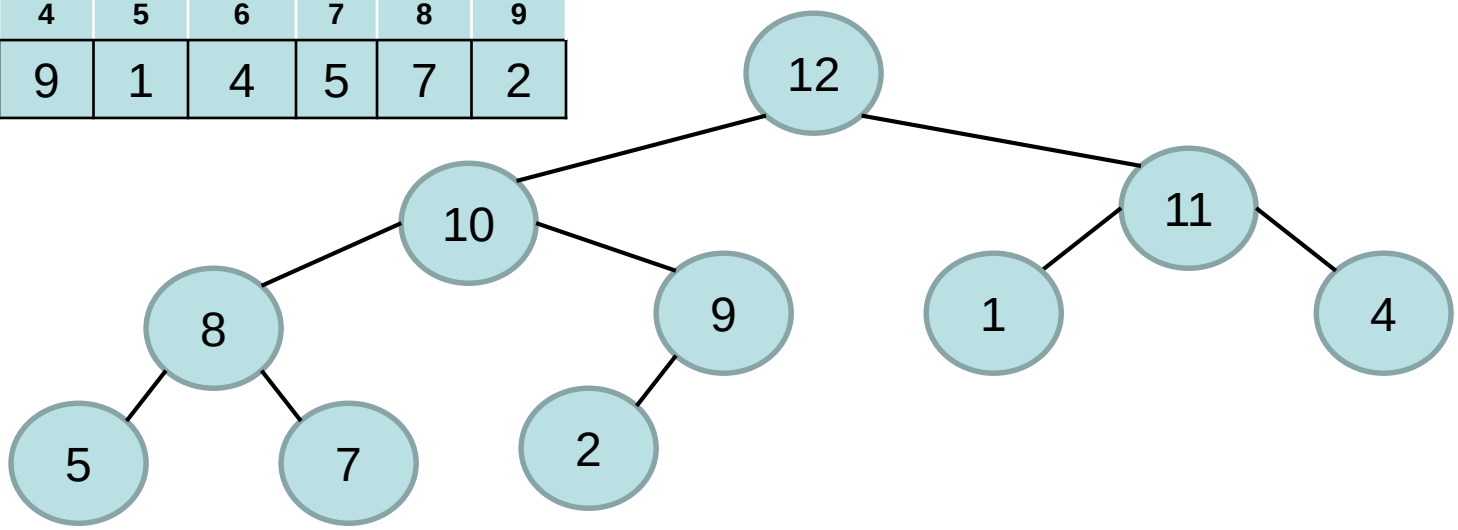
Heap Binário

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 subárvore serão sempre menores que o elemento armazenado na raiz. Essa árvore é completa todos seus níveis, com a possível exceção do nível mais baixo.

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

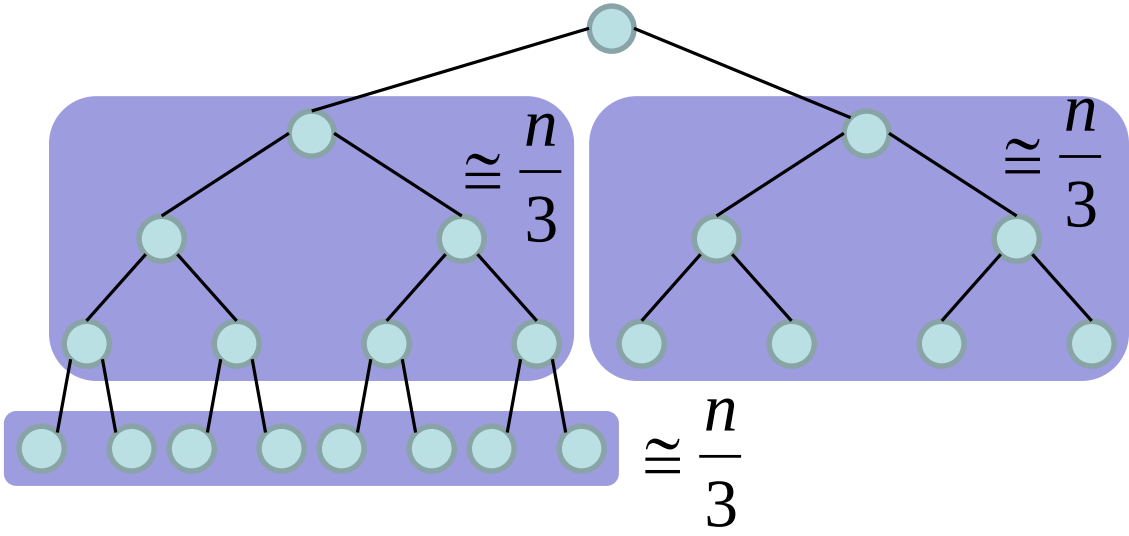
```
int esquerda (int i)
{return (2 * i + 1);}
```

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



Heap Binário

```
void heapify (int *a, int n, int i)
{
        int e, d, maior;
        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);
        }
}
```



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

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

UDESC

Heap Binário

```
int heapExtract(int *a, int n)
{
        int maior;

        maior = a[0];
        a[0] = a[n - 1];
        heapify(a, n - 1, 0);
        return maior;
}

void buildHeap(int *a, int n)
{
        int i;

        for (i = (n-1)/2; i >= 0; i--)
                heapify(a, n, i);
}
```

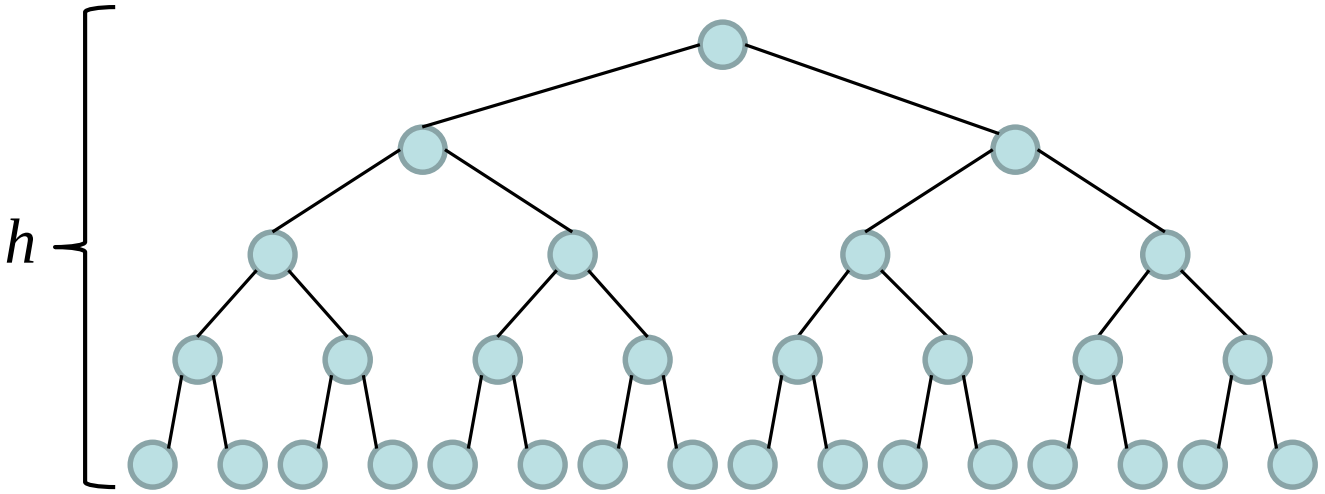
Heap Binário

```
void buildHeap(int *a, int n)
{
        int i;

        for (i = (n-1)/2; i >= 0; i--)
                heapify(a, n, i);
}
```

$$T(n) = \sum_{i=0}^{\lfloor \log_2 n \rfloor} (h-i)2^i$$

$$= O(n)$$



UDESC

Heap Sort

```
void heapSort(int *a, int n)
{
    int i;

    buildHeap(a, n);
    for (i = n - 1; i > 0; i--)
    {
        swap(&a[0], &a[i]);
        heapify(a, i, 0);
    }
}
```

Algoritmos com Inteiros

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

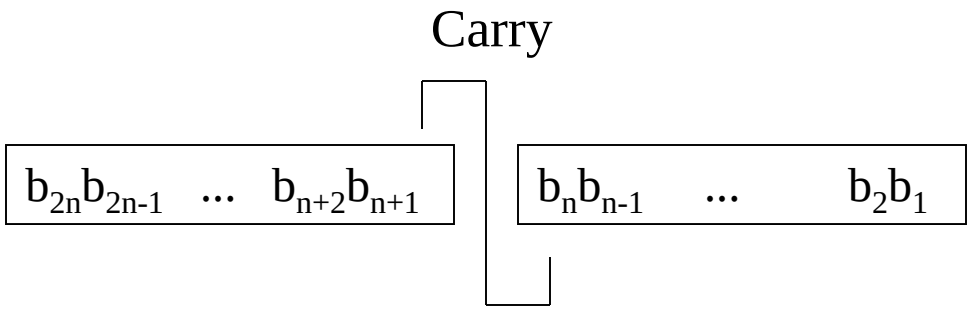
Mas quando essas operações envolvem números cujo o tamanho, em número de bits, é muito maior que a palavra do processador (atualmente 32 ou 64 bits)?

Algoritmos com Inteiros

A adição e multiplicação de grandes números em computadores é bastante similar a operações com bits.

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

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



Podemos considerar que a adição de grandes números tem complexidade $O(n)$, onde n é o número de bits.

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(n).
        y--;
    }
    return r;
}
```

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

Sendo n o número de bits, qual o valor de y no pior caso?

Algoritmos com Inteiros (Multiplicação)

```
  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)
-----------
10000010 (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
}
```

UDESC

Algoritmos com Inteiros (Multiplicação)

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

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

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

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

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) // onde n é o número de bits dos inteiros x e y.
{
    bigInt xl, xr, yl, yr, p1, p2, p3, p4;
    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 à esquerda e mais à 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 + p3) << (n/2) + p4;
}
```

Números Primos

Uma número natural é primo se possui exatamente dois divisores: o número 1 e ele mesmo. Todo número composto pode representado pela multiplicação de seus fatores primos.

Primos relativos:

Dois inteiros são chamados de primos relativos se o único inteiro positivo que divide os dois é 1. Por exemplo, 49 e 15 são primos relativos.

UDESC

Números Primos

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

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

Aritmética Modular

É um sistema para manipular faixas restritas de números inteiros.

Relação de congruência:

$a \equiv b \pmod{m} \Leftrightarrow m$ *divide* $(a - b)$.

$a \equiv b \pmod{m}$ se e somente se $a \bmod m = b \bmod m$.

Exemplos:

$38 \equiv 14 \pmod{12}$, $38 \bmod 12 = 14 \bmod 12$

$-10 \equiv 38 \pmod{12}$, $-10 \bmod 12 = 38 \bmod 12$

UDESC

Aritmética Modular

O ***inverso multiplicativo modular*** de um inteiro a no módulo m é um inteiro x tal que: $ax \equiv 1 \pmod{m}$.

O **inverso multiplicativo** de a no módulo m existe se e somente se a e m são primos relativos.

O inverso multiplicativo de a no módulo m é um inteiro x que satisfaz a equação: $ax = 1 + my$.

Exemplo: Qualquer número congruente à 5 (no módulo 12) é o inverso multiplicativo de 5 no módulo 12: {…, -7, 5, 17, 29, …}.

UDESC

Criptográfia 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:

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

Sendo $n = pq$, onde p e q são números primos, e e um primo relativo de $(p - 1)(q - 1)$:

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

Chave pública = (e, n).

UDESC

Criptográfia RSA

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

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

Onde d é o inverso modular de *e*:

$$ed \equiv 1 \bmod ((p-1)(q-1))$$

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

Criptográfia RSA (Exemplo)

Para $M = 92$, $p = 11$ e $q = 13$ o valor de $n = 143$.

Escolher arbitráriamente um valor para $e = 17$, 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 $d = -7$ (ver algoritmo de Euclides Estendido), outras soluções são $a = 16$ e $d = 113$, $a = 33$ e $d = 233$, …

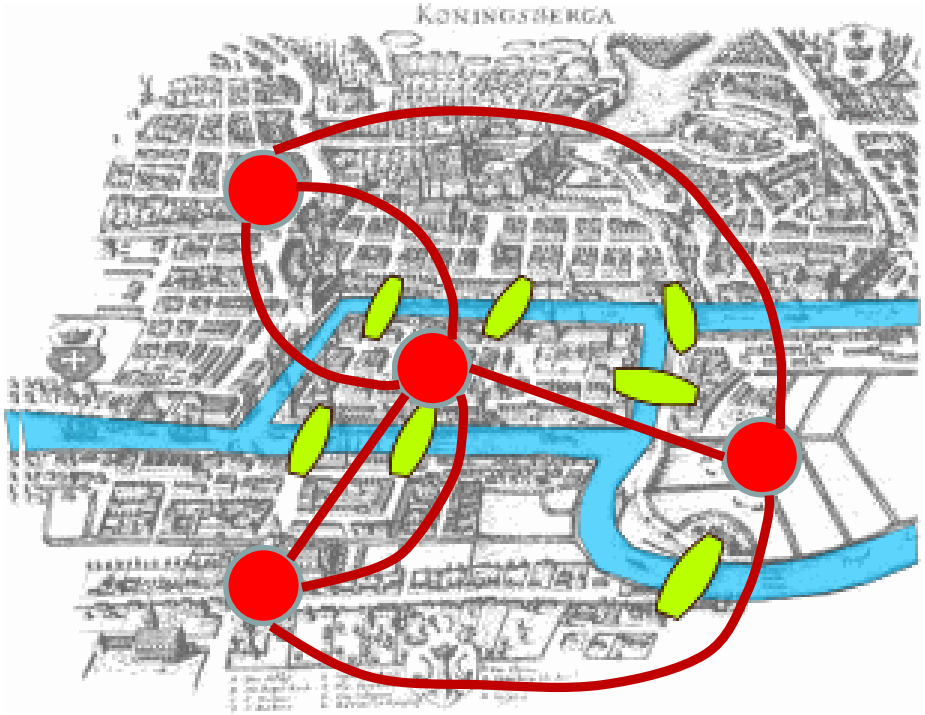
Se -7 é o inverso modular de 17 mod 120 então todo inteiro congruente a -7 mod 120 é também o inverso modular de 17 mod 120: (…, -7, **113**, 233, 353, …). ***Chave privada* = (113, 143)**.

$$C = 92^{17} \bmod 143,\ C = 27.$$

$$M = 27^{113} \bmod 143,\ M = 92.$$

Grafos

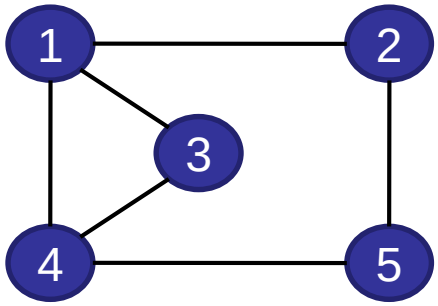
As ideias iniciais sobre grafos foram introduzidas no século XVIII pelo matemático suíço **Leonhard Euler**. Ele usou para resolver o famoso problema das **Sete Pontes de Königsberg.**



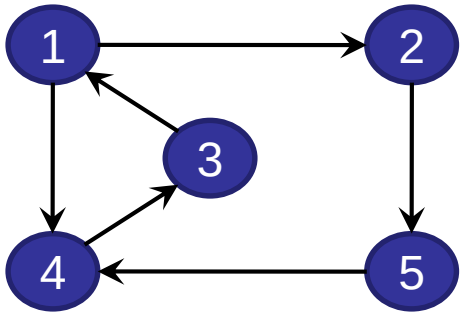
Discutia-se a possibilidade de cruzar as sete pontes sem repetir nenhuma.

Grafos

Um **grafo simples** consiste em um conjunto não vazio de vértices e um conjunto de pares não ordenados chamados de arestas.



Um **grafo direcionado (ou orientado)** consistem em um conjunto não vazio de vértices e um conjunto de arestas que são pares ordenados.

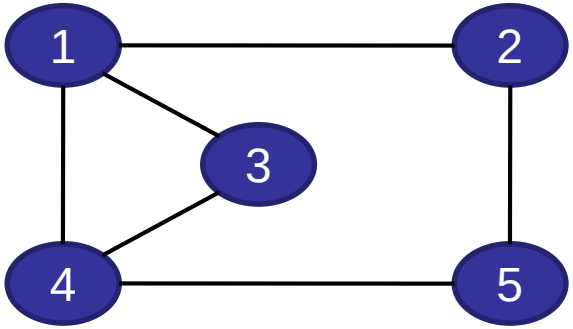


Dois vértices u e v de um grafo $G = (V, A)$ são ditos **adjacentes** se $(u, v) \in A$.

Grafos simples não apresentam *loops*.

Representações de Grafos

São estruturas discretas constituídas de vértices e arestas que conectam os vértices. $G = (V, A)$.



Vértices: $V = \{1, 2, 3, 4, 5\}$

Arestas: $A = \{(1,2),(1,3),(1,4),(2,5),(3,4),(4,5)\}$

Lista de Adjacência

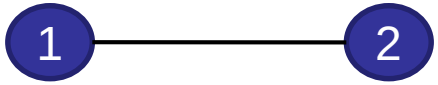
1	→ [2, 3, 4]
2	→ [1, 5]
3	→ [1, 4]
4	→ [1, 3, 5]
5	→ [2, 4]

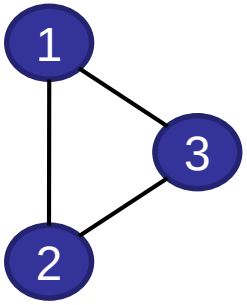
Matriz de adjacência

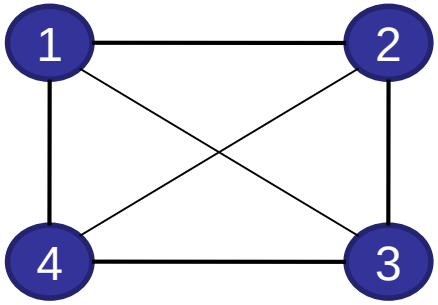
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Grafos

Um grafo simples é dito **completo** se contém uma aresta ligando cada um dos seus vértices.



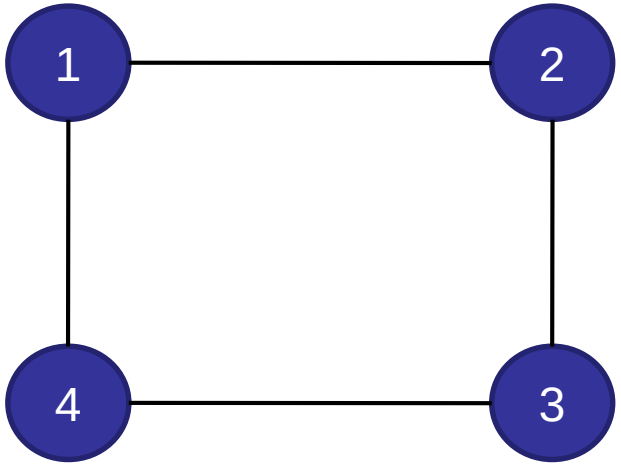


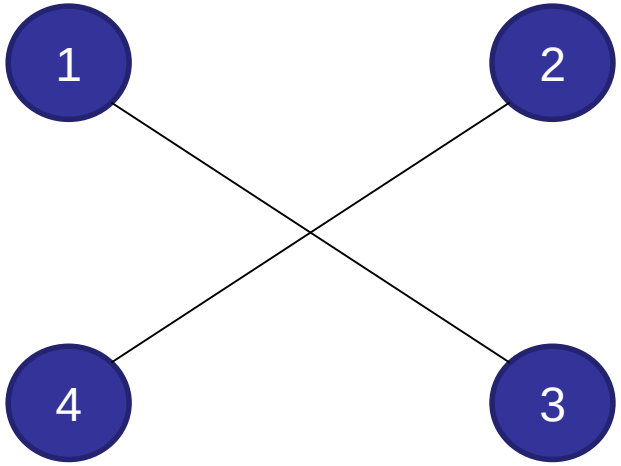


Complemento de um Grafos

O **complemento** de um grafo simples $G = (V, A)$ é denotado por $\bar{G}$, $\bar{G} = (V, \bar{A})$:

Sendo $\bar{A} = \{(u, v) | u \in \mathrm{V}, v \in \mathrm{V}, u \neq v, (u, v) \notin A\}$

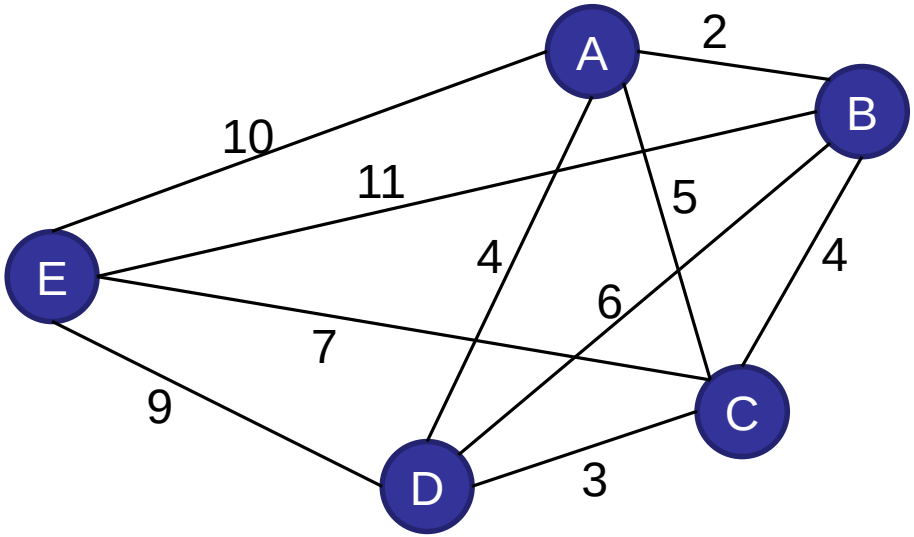




Grafos Ponderados

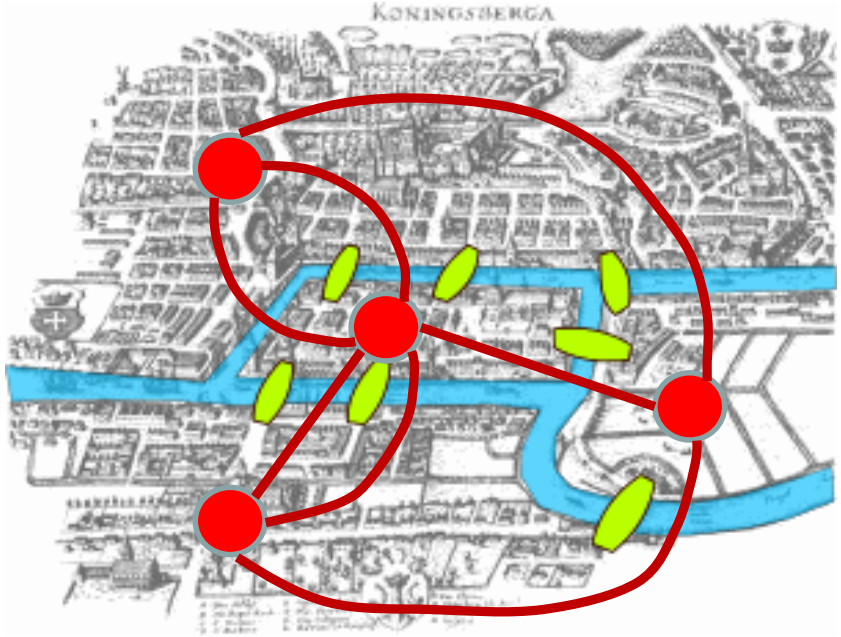
Um **grafo ponderado** (ou **valorado**) possui um custo associado a cada uma das suas arestas, esse custo é normalmente representado como uma função:

$c : A \rightarrow \mathbb{R}$



Multigrafos

Multigrafos são grafos que permitem arestas paralelas.



Pseudografos são multigrafos que permitem *loops*.

Grafo Euleriano

Caminho Euleriano é um caminho que visita cada aresta uma única vez.

Ciclo Eulerinano é um caminho euleriano que termina no ponto de partida. Um grafo que possui um ciclo euleriano é chamado **grafo euleriano**, para possuir esse tipo de ciclo o grafo deve ser conexo e todos seus vértices têm que possuir grau par.

O **grau** de um vértice é o número de arestas que incidem sobre o vértice.

UDESC

Algoritmos Gulosos
(*Greedy Algorithms*)

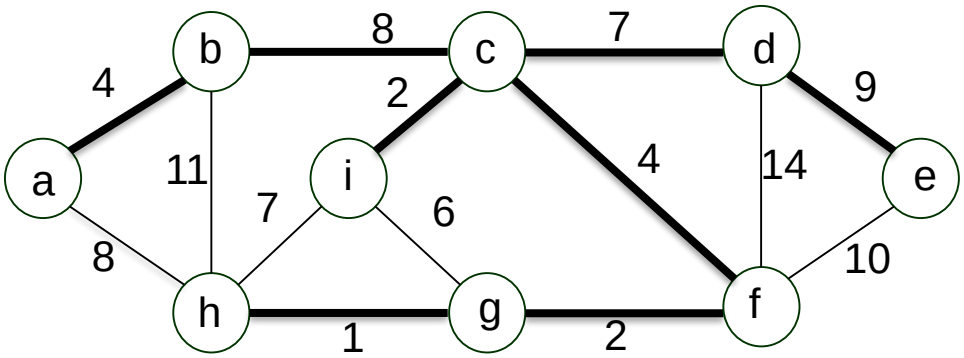
Uma técnica para resolver problemas de otimização, um algoritmo guloso sempre faz a escolha que parece ser a melhor em um determinado momento, esperando que essa melhor escolha local leve ao melhor resultado do problema como um todo.

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

Árvore Geradora Mínima

(*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.



Árvore Geradora Mínima

(*Minimum Spanning Tree*)

MST-PRIM(*G*,*w*, *r*) // *G – grafo*, *w – custos*, *r – vértice inicial*

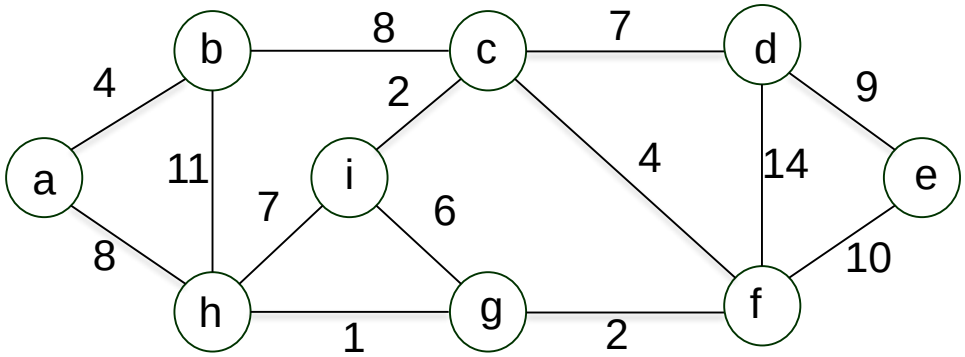
para cada $u \in G.V$

$u.key = \infty$

$u.\pi$ = **null**

$r.key = 0$

$Q = G.V$



Enquanto $Q \neq \emptyset$ // *Executa* $|V|$ *vezes*

u = Extrai-Minimo(Q) // $O(\log |V|)$ *heap binário*

para cada $v \in G.Adj[u]$ // *Executa* $|A|$ *vezes*

se $v \in Q$ **e** $w(u, v) < v.key$

$v.\pi = u$

$v.key = w(u, v)$ // *propriedade de heap* $O(\log |V|)$

Árvore geradora mínima = $\{(v, v.\pi) \mid v \in V - \{r\}\}$

Codificação de Huffman

É 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

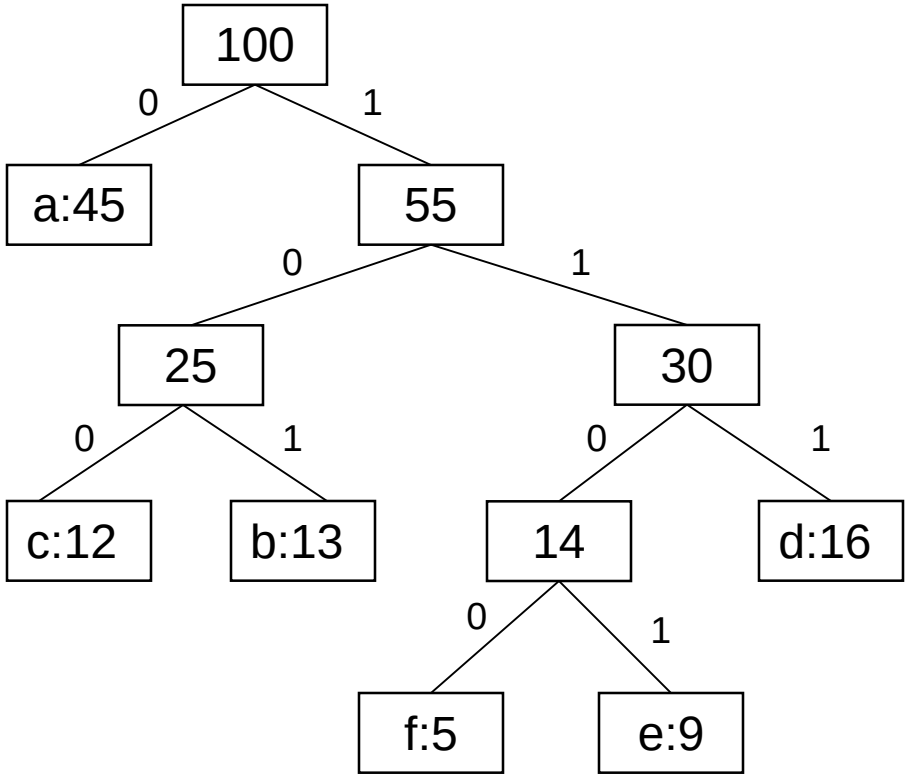
Exemplo:

... faca ...

... 101000010000 ...

... 110001000 ...

Codificação de Huffman (Exemplo)



Exemplo:

... faca ...

... 101000010000 ...

... 110001000 ...

Codificação de Huffman

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