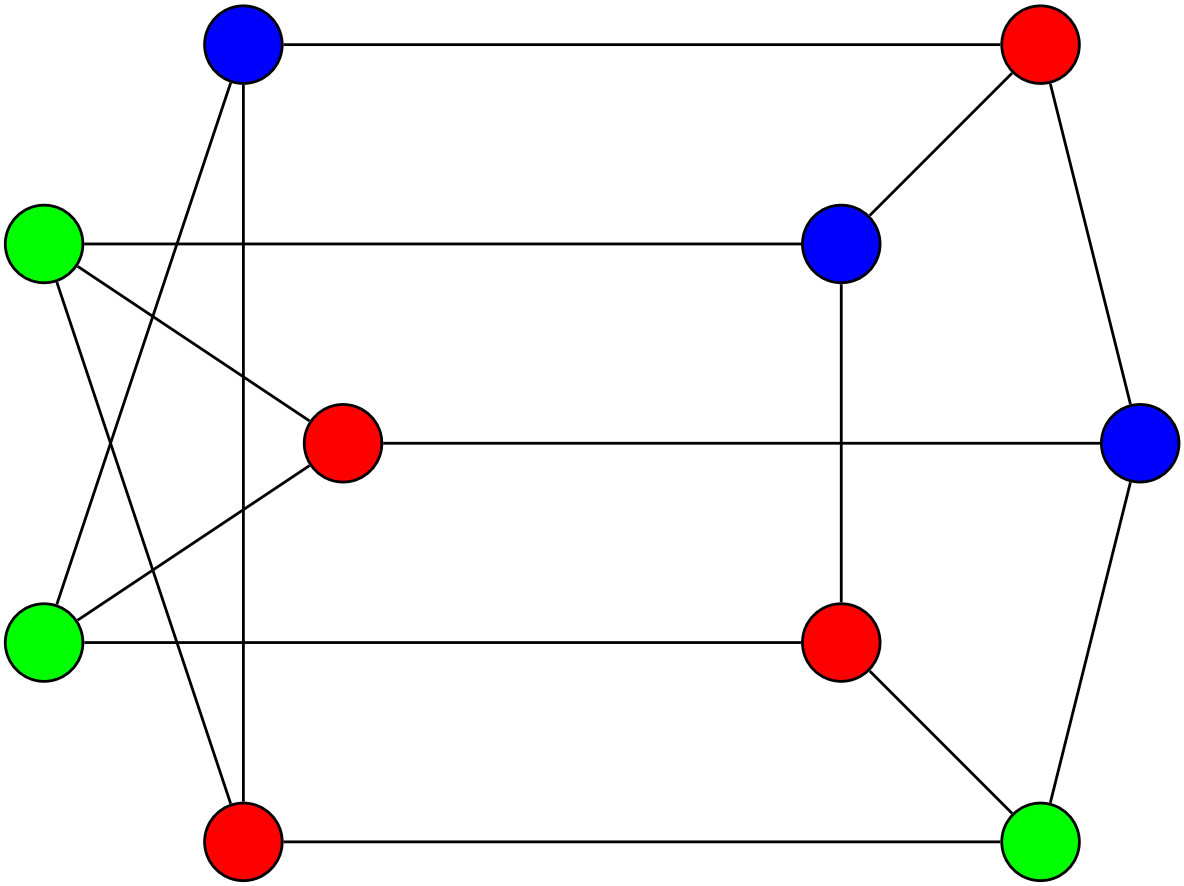
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC
Profa. Carla Negri Lintzmayer

Notas de aula da disciplina Teoria dos Grafos



Esse material é um compilado dos seguintes:

- Sedgewick, R.. *Algorithms in C, part 5: graph algorithms*. 3rd ed. Addison-Wesley. 2002.
- Bondy, J. A.; Murty, U. S. R.. *Graph Theory*. Graduate Texts in Mathematics. Springer. New York. 2008.
- Lintzmayer, C. N.; Mota, G. O.. *Notas de aulas - Análise de algoritmos e estruturas de dados*. Em construção.

5 Aula 6: breve introdução à análise de algoritmos

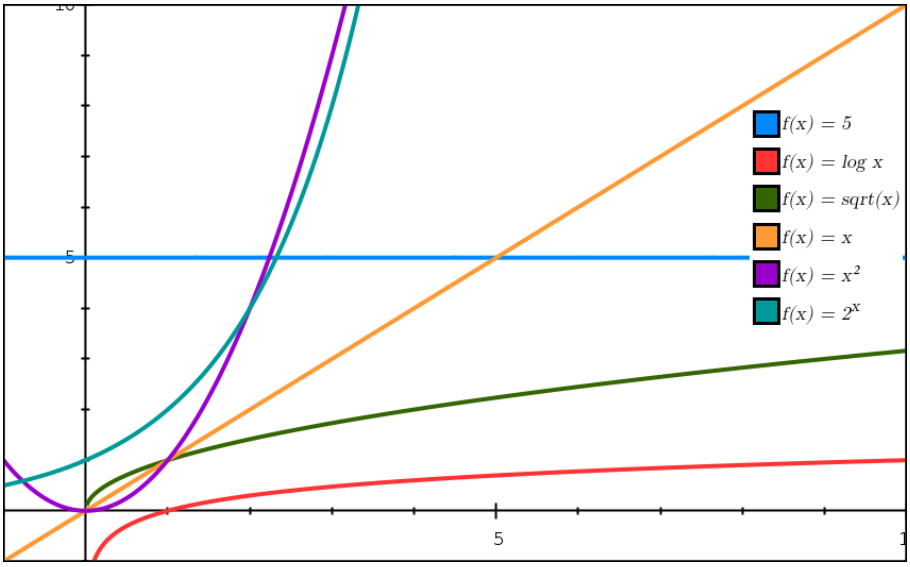
Mais detalhes sobre esses tópicos: http://professor.ufabc.edu.br/~carla.negri/cursos/materiais/Livro-Analise.de.Algoritmos.pdf

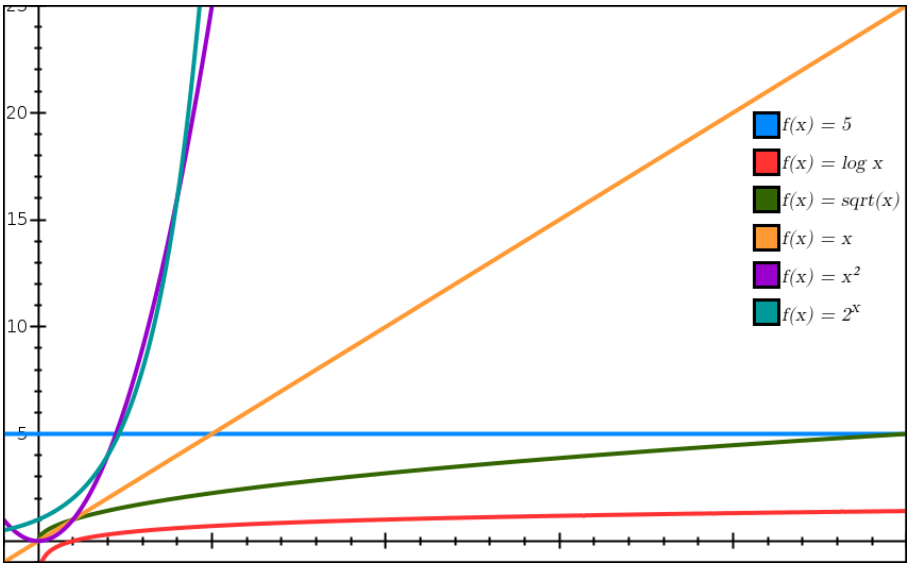
- A *análise de algoritmos* nos permite *prever* o comportamento do algoritmo sem que seja necessário implementá-lo em um dispositivo específico (programa)
- Podemos analisar:
  - desempenho (tempo e memória)
  - correção (se funciona para qualquer entrada)
- Para desempenho, usamos um conceito independente de hardware e linguagem
- *Tempo de execução:* é o número de passos básicos executados descritos como uma função do tamanho da entrada
  - *passo básico:* operações aritméticas ($+$, $-$, $\times$, $\div$, mod , $\lfloor\ \rfloor$, $\lceil\ \rceil$), relacionais ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$), lógicas ($\vee$, $\wedge$, $\neg$), de movimento (atribuição), e controle de fluxo (if-else, for, while) sobre números pequenos (32 ou 64 bits)
  - *tamanho da entrada:* tamanho da representação (vetores = qtd. de elementos, inteiros = qtd. de bits, grafos = número de vértices e número de arestas)
- Exemplo:

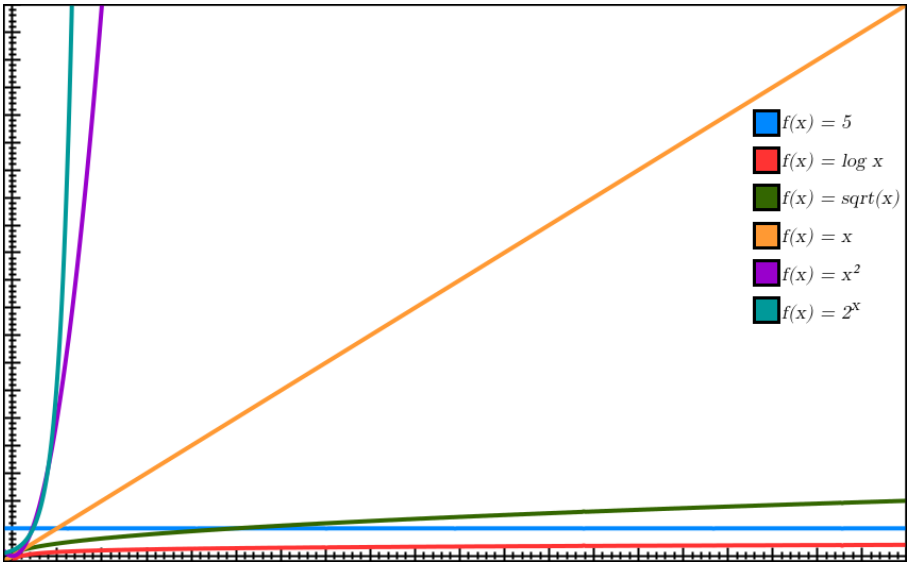
```
1: Função SomaPar(A, n)
2:     total = 0
3:     Para i = 1 até n faça
4:         Se A[i] é par então
5:             total = total + A[i]
6:     Devolve total
```

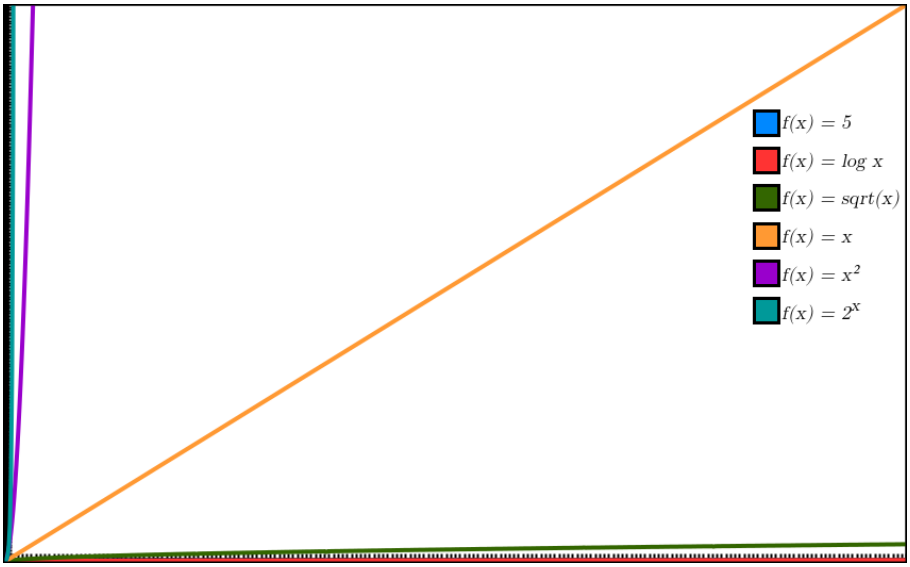
```
1: Função SomaParR(A, n)
2:     Se n == 0 então
3:         Devolve 0
4:     parcial = SomaParR(A, n − 1)
5:     Se A[n] é par então
6:         Devolve parcial + A[n]
7:     Devolve parcial
```

- Uma vez que conseguimos descrever o tempo de execução de um algoritmo como uma função no tamanho da entrada, podemos compará-lo com outros algoritmos que resolvam o mesmo problema
- Vamos focar no *tempo de pior caso*, que é o maior tempo de execução possível que o algoritmo leva para executar uma instância de um dado tamanho n
  - Assim, sempre teremos um *limitante superior* no tempo de execução de qualquer instância
- Um algoritmo é *eficiente* se seu tempo de execução no pior caso puder ser descrito por uma função que é limitada superiormente por uma função polinomial no tamanho da entrada
- A comparação é feita por meio da *ordem de crescimento* das funções
  - c, $\log n$, n^c, $n^c \log n$, 2^n, $n!$





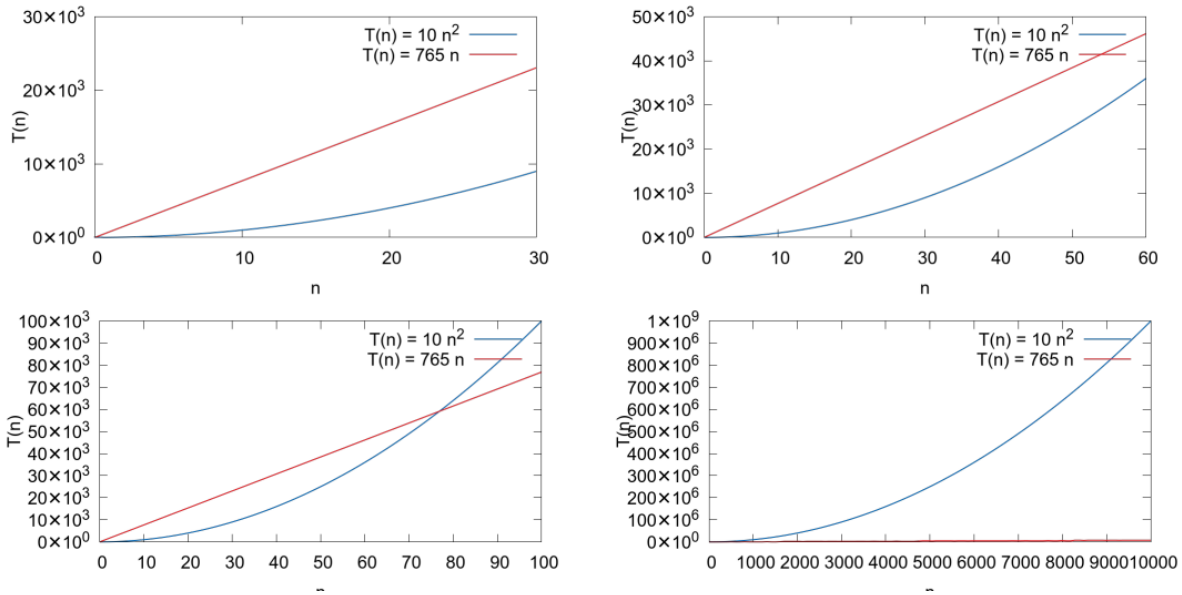




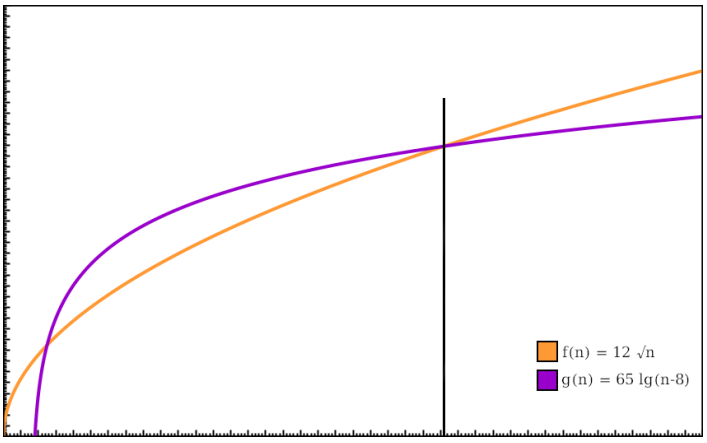
- Para correção, usamos indução:
  - no número de iterações (laços): "no início de qualquer iteração vale que . . . "
  - no tamanho da entrada (algoritmos recursivos): "uma chamada recursiva devolve . . . "

5.1 Notação assintótica

- Estamos interessados no que acontece com $f(n)$ quando n cresce indefinidamente
  - Termos de menor ordem não importam
  - Constantes não importam



- A *notação assintótica* permite justamente isso
- Seja n um inteiro positivo e sejam $f(n)$ e $g(n)$:
  - Dizemos que $f(n) = O(g(n))$ se existem constantes positivas c e n_0 tais que $f(n) \leq cg(n)$ para todo $n \geq n_0$.
  - Dizemos que $f(n) = \Omega(g(n))$ se existem constantes positivas c e n_0 tais que $f(n) \geq cg(n)$ para todo $n \geq n_0$.
  - Dizemos que $f(n) = \Theta(g(n))$ se existem constantes positivas c, d e n_0 tais que $cg(n) \leq f(n) \leq dg(n)$ para todo $n \geq n_0$.
  - Equivalente: Dizemos que $f(n) = \Theta(g(n))$ se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$.
- Exemplo:



Se $f(n) = 12\sqrt{n}$ e $g(n) = 65\log_2(n-8)$, então $f(n) = \Omega(g(n))$ e $g(n) = O(f(n))$.

De fato,

$$\begin{aligned} g(n) &= 65\log_2(n-8) \\ &< 65\log_2(n) \\ &\leq 65\sqrt{n} \\ &= 65\frac{12}{12}\sqrt{n} \\ &= \frac{65}{12}f(n) \end{aligned}$$

- Observações:
  - Se $f(n) = O(n^c)$ para uma constante positiva c, então $f(n) = O(n^d)$ para qualquer $d > c$
  - Se $f(n)$ é um polinômio de ordem c, então $f(n) = O(n^c)$
  - $f(n) = O(g(n) + h(n))$ se e somente se $f(n) = O(g(n)) + O(h(n))$
  - Se $f(n) = O(g(n) + h(n))$ e $g(n) = O(h(n))$, então $f(n) = O(h(n))$
  - Usamos $O(1)$, $\Omega(1)$, $\Theta(1)$ para indicar funções constantes