Estruturas de Dados

Algoritmos

Prof. Marcos Caetano

(Material Base – Prof. Eduardo Alchieri)

Algoritmos
(definição)

- Sequência finita de instruções para executar uma tarefa
  - Bem definidas e não ambíguas
  - Executáveis com uma quantidade de esforço finita
  - Executáveis em um período de tempo finito
- Um algoritmo pode ser escrito de muitas formas
  - Exemplo: em alguma linguagem natural
  - Porém, estamos interessados em algoritmos especificados com alguna precisão por meio de formalisno matemático adequado
    - Exemplo, linguagem de programação

Soma de números sequenciais	Bolo
`int i, total = 0;` `for(i = 1; i <= 10; ++i)` `  total += i;`	tigela ← ingredientes **enquanto**(**tempo**() < 5min) **batedeira**(tigela) **untar**(forma) forma← tigela.conteudo **esperar**(40min)

Algoritmos
(definição)

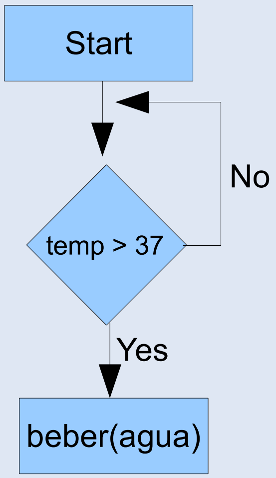
- Um algoritmo não representa, necessariamente, um programa de computador
  - **Algoritmo:** sequência de ações para resolver uma tarefa, ou ainda, um procedimento passo a passo para se chegar com sucesso a um fim;
  - **Programa:** sequência de instruções em código para executar uma operação em um computador
- Um algoritmo corretamente executado não resolve uma tarefa se:
  - For implementado incorretamente;
  - Não for apropriado para a tarefa;

Classificação de Algoritmos (término)

- Alguns autores restringem a definição de algoritmo para procedimentos que (eventualmente) terminam
  - Um algoritmo pode repetir um procedimento ou ação infinitamente
- **Se o tamanho do procedimento não é conhecido, não é possível determinar se ele terminará** (Marvin Minsky)
- Para algoritmos que não terminam, o sucesso não pode ser determinado pela interpretação da resposta e sim por condições impostas pelo próprio desenvolvedor do algoritmo durante sua execução
  - **Exemplo:** um algoritmo que nunca termina mas sempre mantém algo invariante;

Classificação de Algoritmos
(representação)

- Linguagem natural
  - Se estiver quente, beba água
- Pseudo-código
  - Se temperatura > 37ºC então beber(agua)
- Diagrama



- Linguagem de programação
  - If(temp > 37) then beber(agua)

Classificação de Algoritmos
(implementação)

- **Iterativo**: estruturas de repetições (laços, pilhas, etc.)
- **Recursivo**: invoca a si mesmo até que certa condição seja satisfeita

- **Serial**: cada instrução é executada em sequência
- **Paralela**: várias instruções executadas ao mesmo tempo

- **Determinístico**: decisão exata a cada passo
- **Probabilístico**: decisão provável em algum(s) passo(s)

- **Exato**: resposta exata
- **Aproximado**: resposta próxima a verdadeira solução

Classificação de Algoritmos
(implementação - recursividade)

- Um algoritmo que transforma um problema grande em problemas menores, cujas soluções requerem a aplicação dele mesmo, é chamado recursivo.

Fatorial iterativo

```
int factorial (int n) {
  int result = 1;
  while(n > 1) {
     result *= n;
    n -= 1;
  }
  return result ;
}
```

Fatorial recursivo

```
int factorial (int n) {
  if (n≤1)
    return 1;
  else
    return n* factorial (n−1);
}
```

- **Recursão vs Iteração**
  - A implementação iterativa tende a ser ligeiramente mais rápida na prática do que a implementação recursiva
  - Existem problemas cujas soluções são inerentemente recursivas (ex.: algoritmos de ordenação, buscas em árvores)

Classificação de Algoritmos
(implementação - recursividade)

- Existem dois tipos de recursividade
  - **Direta:** função invoca a si própria
  - **Indireta**: partindo de uma função f1, e através de uma cadeia de invocação de funções, f1 é novamente invocada
- Outro exemplo: Série de Fibonacci
  - 0,1,1,2,3,5,8,13,21,34,...
  - Função recursiva:

```
Fib(n) = n se n== 0 ou n==1
Fib(n) = Fib(n-2) + fib(n-1) se n >=2
```

Classificação de Algoritmos
(implementação - recursividade)

- Fib(4) = Fib(2) + Fib(3) =

  Fib(0) + Fib(1) + Fib(3) =

  0 + Fib(1) + Fib (3) =

  0 + 1 + Fib(3) =

  1 + Fib(3) =

  1 + Fib(1) + Fib(2) =

  1 + 1 + Fib(2) =

  1 + 1 + Fib(0) + Fib(1) =

  1 + 1 + 0 + Fib(1) =

  1 + 1 + 0 + 1 =

  1 + 1 + 1 =

  1 + 2 =

  3

Custo de Algoritmos
(análise)

- Um algoritmo deve:
  - Funcionar corretamente
  - Executar o mais rápido possível
  - Utilizar a memória da melhor forma possível
- A fim de sabermos mais sobre um algoritmo, podemos analisá-lo
  - Precisamos estudar as suas especificações e tirar conclusões sobre como a sua implementação (o programa) irá se comportar em geral.

Custo de Algoritmos
(análise)

- **Mas o que podemos analisar de um algoritmo?**
  - O **tempo de processamento** de um programa como função de seus dados de entrada;
  - O **espaço de memória máximo** ou total requerido para os dados do programa;
  - O comprimento total do código do programa;
  - Se o programa chega corretamente ao resultado desejado;
  - A complexidade do programa
    - Facilidade em ler, entender e modificar
  - A robustez do programa;
    - Exemplo: como ele lida com entradas errôneas ou inesperadas

Custo de Algoritmos
(análise)

- Estaremos particularmente interessados em analisar o **tempo de execução** e o **espaço de memória utilizado**
- **Como comparar algoritmos em função do custo de tempo?**
  - Computadores diferentes podem funcionar em frequências diferentes
    - Ainda, diferente hardware (processador, memória, disco, etc.), diferente SO, etc.
  - Compiladores podem otimizar o código antes da execução
  - Um algoritmo pode ser escrito diferente, de acordo com a linguagem de programação utilizada
  - Além disso, uma análise detalhada, considerando todos estes fatores, seria difícil, demorada e pouco significativa
    - Tecnologias mudam rapidamente

Custo de Algoritmos
(análise)

- Podemos medir o custo de tempo **contando quantas operações são realizadas pelo algoritmo**
  - Atribuições, comparações, operações aritméticas, instruções de retorno, etc.
- **Cada operação demora o mesmo tempo ?**
  - Não, mas podemos simplificar nossa análise
  - Exemplo: i = i + 1
    - **Análise detalhada:** 2 x tempo de recuperar uma variável (i e 1) + 1 x tempo da soma + 1 x tempo para armazenar o valor na variável (i)
    - **Análise simplificada:** 4 operações

Custo de Algoritmos
(análise)

- Na análise do algoritmo abaixo, como saberemos quantas vezes o loop é executado ?

```
busca_linear(vetor, chave) {
    i = 0;
    enquanto(i < tamanho(vetor)) {
        se(vetor[i] == chave)
            retorna i;
        ++i;
    }
    retorna -1;
}
```

- Os dados de entrada determinarão quantas vezes o loop é executado
    - Como não faz sentido analisar um algoritmo para apenas um determinado conjunto de entradas e é impossível fazer esta análise para todas as entradas possíveis, consideraremos apenas dois cenários: **o pior caso e o melhor caso**

Custo de Algoritmos
(análise)

- Pior caso

`busca_linear(vetor, chave) {`	
`    i = 0;`	2
`    enquanto(i < tamanho(vetor)) {`	3*(n+1) //sendo n o tamanho do vetor, já calculado
`        se(vetor[i] == chave)`	3*(n) //considerando o acesso a vetor[i] como uma única operação
`            retorna i;`	0
`        ++i;`	4*(n)
`    }`	
`    retorna -1;`	1
`}`	

Total = 10n + 6

Custo de Algoritmos
(análise)

- Pior caso - Simplificado

`busca_linear(vetor, chave) {`	
`i = 0;`	1
`enquanto(i < tamanho(vetor)) {`	n+1 //sendo n o tamanho do vetor
`se(vetor[i] == chave)`	n
`retorna i;`	0
`++i;`	n
`}`	
`retorna −1;`	1
`}`	

Total = 3n + 3

Custo de Algoritmos
(análise)

- Melhor caso

```
busca_linear(vetor, chave) {
    i = 0;                              2
    enquanto(i < tamanho(vetor)) {      3*(1)
        se(vetor[i] == chave)           3*(1) //considerando o acesso a vetor[i] como uma única operação
            retorna i;                  1
        ++i;                            0
    }
    retorna −1;                         0
}
```

Total = 9

Custo de Algoritmos
(análise)

- Melhor caso - Simplificado

busca_linear(vetor, chave) {	
i = 0;	1
enquanto(i < tamanho(vetor)) {	1
se(vetor[i] == chave)	1
retorna i;	1
++i;	0
}	
retorna −1;	0
}	Total = 4

Custo de Algoritmos
(análise)

- Diferentes algoritmos podem realizar a mesma tarefa com instruções diferentes a um custo maior ou menor

- **Problema:** ordenar os números da mega-sena

```
for(int i = 1; i < 60; ++i) {
  i = menor(numeros);
  adicionar( lista_ord , i);
  remover(numeros, i);
}
```

```
lista_ord = numeros;
while(!em_ordem(lista_ord))
  lista_ord = embaralhar(numeros);
```

- Qual dos dois é melhor? Sempre?

Classificação de Algoritmos
(complexidade)

- **Porque analisar um algoritmo?**
  - Para avaliar sua performance e comparar diferentes algoritmos;
- **Mas analisar o que?**
  - Tempo de execução;
  - Uso de memória;
  - Pior caso, caso típico (dependendo das entradas), melhor caso;
- IMPORTANTE
  - Análise de algoritmos compara algoritmos e não programas

Classificação de Algoritmos
(complexidade)

- **Análise de um algoritmo particular**
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico ?
  - Características de devem ser investigadas:
    - **Tempo:** análise do número de vezes que cada parte do algoritmo deve ser executada
    - **Espaço**: estudo da quantidade de memória necessária
- **Análise de uma classe de algoritmos**
  - Qual é o algoritmo de menor custo possível para resolver um problema particular ?
    - Toda uma família de algoritmos é investigada (busca pelo melhor possível);
  - Coloca-se limites para a complexidade computacional dos algoritmos pertencentes à classe;

Classificação de Algoritmos
(complexidade)

- **Custo de um algoritmo**
  - Ao determinar o menor custo possível para resolver problemas de uma classe, tem-se a medida da dificuldade inerente para resolver o problema;
  - **Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é considerado ótimo**;
- Podem existir vários algoritmos para resolver o mesmo problema;
  - É possível compará-los e escolher o mais adequado;
  - **Se há vários algoritmos com custo de execução dentro da mesma ordem de grandeza**, pode-se considerar tanto os custos reais como os custos não aparentes, como: alocação de memória, carga de arquivos, etc.

Classificação de Algoritmos
(complexidade)

- **Análise pela execução:**
  - A eficiência do programa depende da linguagem (compilada ou interpretada)
  - Depende do sistema operacional
  - Depende do hardware (quantidade de memória, velocidade do processador, etc.)

  - **Útil nas comparações entre programas em máquinas específicas pois, além dos custos do algoritmo, são comparados os custos não aparentes (alocação de memória, carga de arquivos, etc.)**

Classificação de Algoritmos
(complexidade)

- **Análise pelo modelo matemático:**
  - Não depende do computador nem da implementação;
  - O custo das operações mais signicativas deve ser especificado, e algumas operações são desprezadas;
  - É possível analisar a complexidade do algoritmo dependendo dos dados de entrada;
  - **Útil nas comparações entre algoritmos distintos que resolvem um mesmo problema;**

Classificação de Algoritmos
(complexidade)

- **Exemplo:** descobrir o maior número em uma lista;
  - Considerando somente atribuições como operações relevantes e que todas as atribuições possuem o mesmo custo;

Algoritmo	Tempo	Espaço
$lista = \mathbf{list}();$	$T \cdot 1$	$S \cdot (n+1)$
$maior = -\infty;$	$T \cdot 1$	$S \cdot 1$
$\mathbf{for}(i = 0; i < n; ++i)$	$T \cdot (n+1)$	$S \cdot 2$
$\quad maior = \mathbf{max}(maior, lista[i]);$	$T \cdot n$	$S \cdot 0$
	$g(n) = 2n + 3$	$g(n) = n + 4$

  - Como veremos adiante, conforme **n** aumenta, o valor de **n** passa a determinar o custo destas funções;
    - Podemos dizer que ambas possuem um custo O(n)

Classificação de Algoritmos (complexidade)

- **Qual o custo do seguinte algoritmo de soma?**
  - Considerando somente atribuições como operações relevantes e que todas as atribuições possuem o mesmo custo;

```
soma = 0
for(i = 0; i < n; ++i)
  soma += i;
```

  - **Custo tempo:** g(n) = 2 + 2n
    - 2 na inicialização e 2 por repetição
  - **Custo (espaço):** g(n) = 3
    - 3 variáveis (soma, i, n)

Classificação de Algoritmos
(complexidade)

- **Qual o custo do seguinte algoritmo de divisão de elementos ?**
  - Considerando somente atribuições como operações relevantes e que todas as atribuições possuem o mesmo custo;

```
for(i = 0; i < n; ++i)
  for(j = 0; j < n; ++j)
    a[i,j] /= k;
```

Custo (tempo): $g(n) = 1 + 2n + 2n^2$

- 1 na inicialização
- n no laço externo (i)
- n no laço interno (j)
- n^2 no laço interno (j)
- n^2 no laço interno ($a[i,j]$)

Custo (espaço): $g(n) = n^2 + 4$

- 4 variáveis simples (i, n, j, k)
- n^2 variável composta (a)

Classificação de Algoritmos
(complexidade)

- Considerando a função **$g(n) = 1 + 2n + 2n^2$**, temos o seguinte crescimento em função de n:
  - $g(1) = 1 + 2*1 + 2*(1)^2 = 5$
  - $g(5) = 1 + 2*5 + 2*(5)^2 = 61$
  - $g(10) = 1 + 2*10 + 2*(10)^2 = 221$
  - $g(100) = 1 + 2*100 + 2*(100)^2 = 20201$
  - $g(1000) = 1 + 2*1000 + 2*(1000)^2 = 2002001$
  - ...

  O último termo é dominante, portanto, conforme o valor de n aumenta, os dois primeiros termos podem ser desprezados

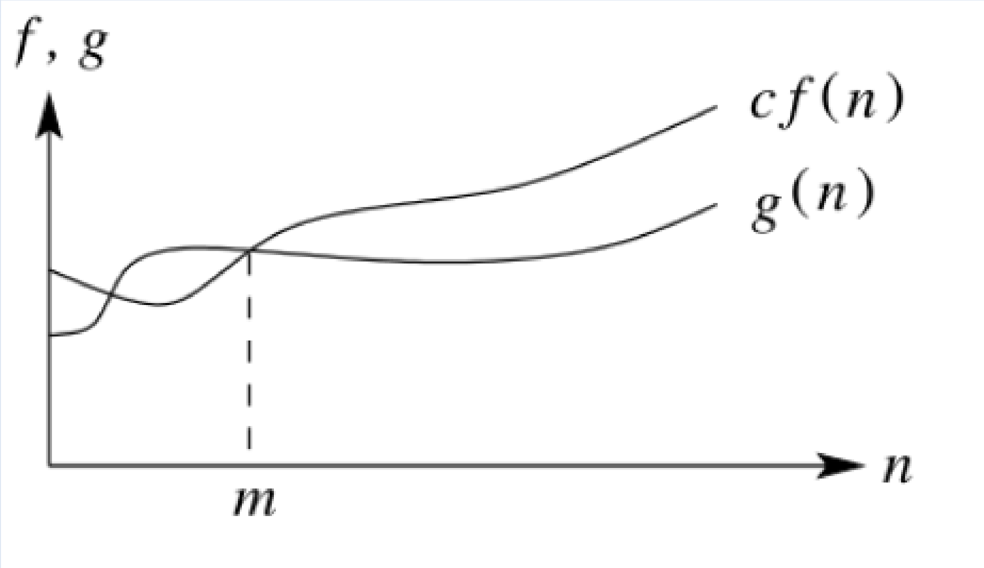
Notação Assintótica

- Notação matemática usada para analisar o comportamento das funções
  - Utilizada para descrever o uso de recursos computacionais
  - **Notação:** Grade-O ou Big-O

- **Permite**
  - Prever o comportamento do algoritmo;
  - Determinar qual algoritmo utilizar;

Notação Assintótica

- Dominância Assintótica
  - Uma função f(n) domina assintoticamente outra função g(n) se existem duas constantes positivas **c** e **m** tais que, para todo $n \geq m$, temos $0 \leq |g(n)| \leq c*|f(n)|$



Notação Assintótica

- Dizemos que $g(n)$ é $O(f(n))$ (ou ainda **$g(n) = O(f(n))$**) para expressar que $f(n)$ domina assintoticamente $g(n)$
  - Lê-se: "**g(n) é da ordem de no máximo f(n)**"
- Exemplos
  - $g(n) = 2n$ e $f(n) = n$
    - $|2n| \leq 3*|n|$, para todo $n \geq 0$
    - $g(n) = O(f(n)) = O(n)$
  - $g(n) = (n + 1)^2$ e $f(n)=n^2$
    - $|(n + 1)^2| \leq 4* |n^2|$, para todo $n \geq 1$
    - $g(n) = O(f(n)) = O(n^2)$

Notação Assintótica

- Exemplos
  - $g(n) = 3n^3 + 2n^2 + n$ e $f(n)=6n^3$
    - $|3n^3 + 2n^2 + n| \leq 6 * |n^3|$, para todo $n \geq 0$
    - $g(n) = O(n^3)$
    - ”**g(n) é da ordem de no máximo f(n)**”
    - Note que também, $g(n) = O(n^4)$
    - Porém, na análise assintótica, estamos interessados nos limites estreitos (justeza)
- Justeza
  - Considere uma função $g(n) = O(f(n))$. Se para toda função $h(n)$ tal que $g(n) = O(h(n))$ também for verdade que $f(n) = O(h(n))$, então $f(n)$ é um limite assintótico justo ou estreito para $g(n)$.

Notação Assintótica

- No exemplo anterior: $g(n) = 3n^3 + 2n^2 + n$
  - $g(n) = O(n^4)$ não é um limite justo
    - Considere $h(n) = 6n^3$ e $f(n) = n^4$, temos que:
    - 1) $g(n) = O(h(n))$, como mostrado anteriormente
    - 2) $h(n)$ não é dominante assintóticamente sobre $f(n)$, i.e., não é verdade que $f(n) = O(h(n))$
    - Logo, $f(n) = O(n^4)$ não é um limite justo para $g(n)$

Notação Assintótica

- Exemplo da análise de algoritmos
  - Comparar cada elemento de um vetor aos outros elementos
  - Função de custo: comparação

Algoritmo

```
for(i = 0; i < n; ++i)
  for(j = 0; j < n; ++j)
    if (i != j)
      comparar(i,j);
```

$$g(n) = n(n-1) \Rightarrow O(n^2)$$

Algoritmo Otimizado

```
for(i = 0; i < n - 1; ++i)
  for(j = i + 1; j < n; ++j)
    comparar(i,j);
```

$$g(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2} \Rightarrow O(n^2)$$

Notação Assintótica

- Exercício: Qual o custo (tempo) dos seguintes algoritmos considerando atribuições e operações matemáticas ?

Algoritmo1

```
for(i = 0; i < n; ++i)
  for(j = 0; j < n; ++j)
    a[i][j] = b[i][j] * c[i][j];
```

Algoritmo2

```
for(i = 0; i < n-1; ++i)
  for(j = i + 1; j < n; ++j) {
    aux = a[i][j];
    a[i][j] = a[j][i];
    a[j][i] = aux;
  }
```

Notação Assintótica

Algoritmo1

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
    a[i][j] = b[i][j]*c[i][j];
```

- 1 atribuição feita uma única vez: i=0
- 1 soma feita n vezes: ++i
- 1 atribuição feita n vezes: ++i
- 1 atribuição feita n vezes: j=0
- 1 soma feita n^2 vezes: ++j
- 1 atribuição feita n^2 vezes: ++j
- 1 multiplicação feita n^2 vezes: b[i][j]*c[i][j]
- 1 atribuição feita n^2 vezes: a[i][j]= b[i][j]*c[i][j]

$$1 + 3n + 4n^2 \Rightarrow O(n^2)$$

Notação Assintótica

Algoritmo2

```
for(i = 0; i < n-1; ++i)
  for(j = i + 1; j < n; ++j)
    aux = a[i][j];
    a[i][j] = a[j][i];
    a[j][i] = aux;
```

- 1 atribuição feita uma única vez: i=0
- 1 subtração feita uma única vez: n-1
- 1 soma e atribuição feitas $n-1$ vezes: ++i
- 1 soma e atribuição feitas $n-1$ vezes: j=i+1
- 4 atribuições e uma soma dentro do laço j: ++j, aux = a[i][j], a[i][j] = a[j][i], a[j][i] = aux feitas (n-1)+(n-2)+ ... + 1 vezes

$$2+4(n-1)+5\frac{n(n-1)}{2}=\frac{5n^2}{2}+\frac{3n}{2}-2 \Rightarrow O(n^2)$$

Notação Assintótica

- Propriedades do O

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \cdot f(n) &= O(f(n)) \text{ (c constante)} \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n))) &= O(f(n)) \\
O(f(n) + g(n)) &= O(max(f(n), g(n))) \\
O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)) \\
f(n) \cdot O(g(n)) &= O(f(n) \cdot g(n))
\end{aligned}
$$

- Ainda temos a propriedade da transitividade
  - Se g(n) = O(f(n)) e f(n) = O(h(n)), então g(n) = O(h(n))

Notação Assintótica

- Nomes de expressões comuns de O
  - O(1) = constante
  - O(log n
  - O(
  - O(
  - O(
  - O(
  - O(

Número de instruções executadas

g(n)	1	n	$n \cdot log(n)$	n^2	n^3	2^n	$n!$
2	1	2	0.6	4	8	4	2
5	1	5	3.5	25	125	32	120
10	1	10	10	100	1000	1024	3628800
20	1	20	26	400	8000	1048576	$2.4 \cdot 10^{18}$
100	1	100	200	10000	1000000	$1.2 \cdot 10^{30}$	$9.3 \cdot 10^{157}$

  - O(2^n): exponencial
  - O(n!): fatorial

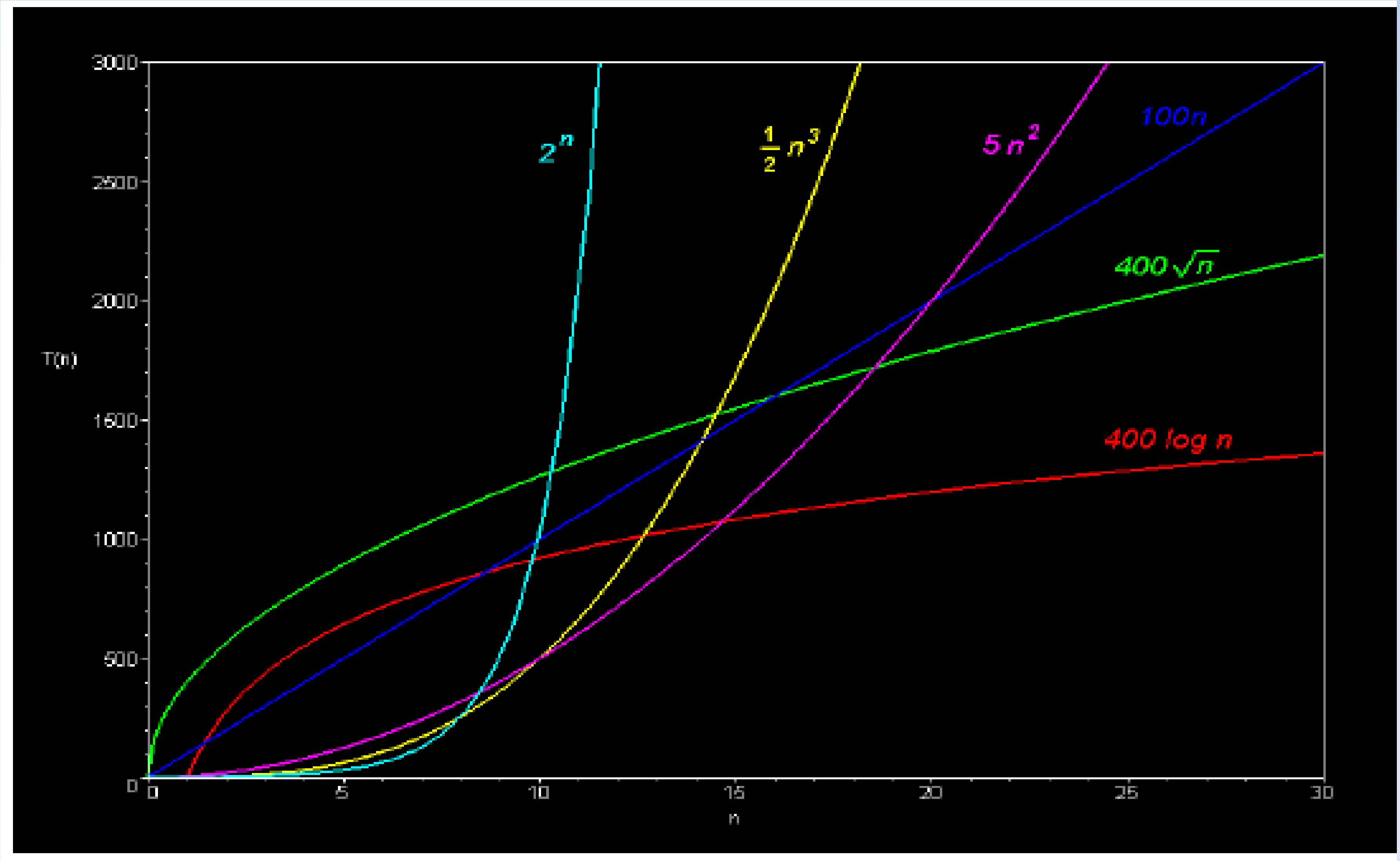
Notação Assintótica

- Noções de grandezas
  - Se 1 instrução = 10^{-6} seg

n	$n \cdot log(n)$	n^2	n^3
10	33.2	100	1.000
100	664	10.000	1 seg
1000	9966	1 seg	16,7 min
10.000	132.877	100 seg	278 hrs
100.000	1,7 seg	2,8 hrs	31,7 anos

Notação Assintótica

- Noções de grandezas



Notação Assintótica

- Limite assintótico inferor – Ômega (Ω)
  - $g(n) = \Omega(f(n))$
    - Uma função f(n) é ômega de outra função g(n) se existem duas constantes positivas c e m tais que, para todo $n \geq m$, temos $|g(n)| \geq c^*|f(n)|$
  - Exemplo: $g(n) = 2n$ e $f(n) = n$
    - Já vimos que $g(n) = O(f(n)) = O(n)$
    - Agora, como: $|2n| \geq 1^*|n|$, para todo $n \geq 0$
    - $g(n) = \Omega(f(n)) = \Omega(n)$
- Notação Teta (Θ): Quando uma função é $O(f(n))$ e Ω $(f(n))$ ao mesmo tempo, dizemos que ela é teta
  - No exemplo anterior: $g(n) = \Theta(f(n))$
- Notação Pequeno – o (little – o) (o): quando uma função é

Notação Assintótica

- Técnicas de análise de algoritmos
  - Considerar memória infinita
  - Não considerar o SO e nem o computador
  - Analisar o algoritmo e não o programa
  - Considerar o tamanho das entradas
  - Ter cuidado ao escolher a função de custo:
    - Atribuição
    - Adição
    - Multiplicação
    - Comparação
    - Etc.

Notação Assintótica

- Técnicas de análise de algoritmos
  - A complexidade de um laço é igual ao número de comandos internos multiplicado pelo número de vezes que o laço é executado
  - A complexidade de laços aninhados é o produto dos tamanhos dos laços
  - Para uma sequência de laços do algoritmo, a complexidade é a do laço de maior custo
  - Em testes condicionais, a complexidade é a maior das duas partes (if/else) do teste

Notação Assintótica

- Técnicas de análise de algoritmos
  - Análise de funções não recursivas
    - Calcular o tempo de execução de cada procedimento semparadamente
    - Avaliar os procedimentos que chamam outros procedimentos que não chamam outros procedimentos, usando os tempos já avaliados
  - Análise de funções recursivas
    - Definir uma equação de recorrência para a função recursiva
      - Descreve uma função em termos de seu valor para entradas menores
    - Resolvendo a função de recorrência, é possível determinar o custo do algoritmo
    - Exemplo: função fatorial