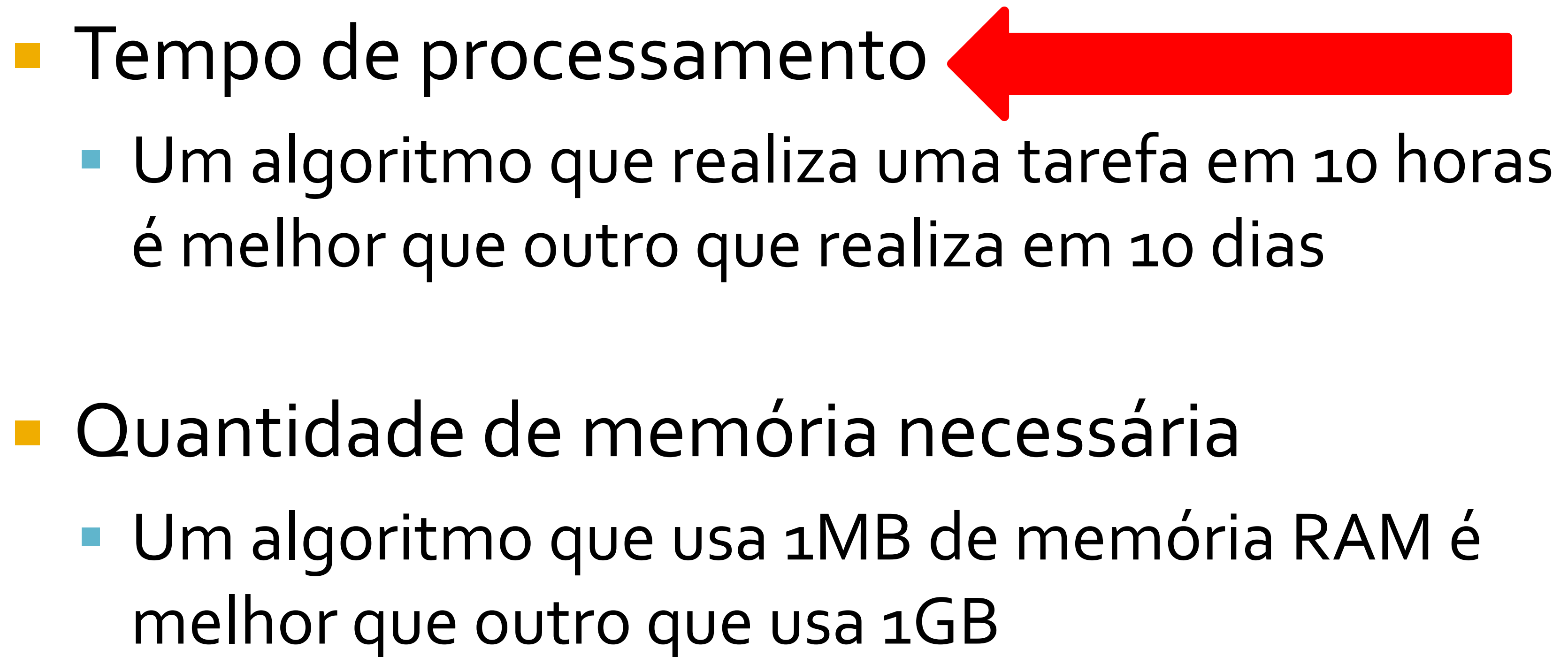
Análise de complexidade

Como escolher um algoritmo?



- Tempo de processamento
  - Um algoritmo que realiza uma tarefa em 10 horas é melhor que outro que realiza em 10 dias
- Quantidade de memória necessária
  - Um algoritmo que usa 1MB de memória RAM é melhor que outro que usa 1GB

Tempo de processamento

- Medir o tempo gasto por um algoritmo
  - Não é uma boa opção
  - Depende do compilador
    - Pode preferir algumas construções ou otimizar melhor
  - Depende do hardware
    - GPU vs. CPU, desktop vs. smartphone

- **Estudar o número de vezes que operações são executadas**

Exemplo – tempo de processamento

- Achar o máximo de um vetor

```
int vmax(int *vec, int n) {            -
        int i;                         -
        int max = vec[0];              1
        for(i = 1; i < n; i++) {       n-1
                if(vec[i] > max) {     n-1
                        max = vec[i];  A < n-1
                }                      n-1
        }                              n-1
        return max;                    1
}
```

- Complexidade: $f(n) = n\text{-}1$
- Esse algoritmo é ótimo

Análise do tempo de processamento

- Análise de complexidade feita em função de n
  - n indica o tamanho da entrada
    - Número de elementos no vetor
    - Número de vértices num grafo
    - Número de linhas de uma matriz
- Diferentes entradas podem ter custo diferente
  - Melhor caso
  - Pior caso
  - Caso médio

Exemplo – Busca sequencial

- Recuperar um registro num arquivo procurando sequencialmente
- Quantos registros precisam ser processados em uma busca?
- Melhor caso:
- Pior caso:
- Caso médio:

Exemplo – MinMax

- Problema: encontrar o valores minimo e máximo em um vetor

```
      -    void minmax(int *vec, int n, int *min, int *max) {
      -       int i;
      1       int *min = vec[0];
      1       int *max = vec[0];
    n-1       for(i = 1; i < n; i++) {
    n-1          if(vec[i] < *min) {
A < n-1             *min = vec[i];
      -          }
    n-1          if(vec[i] > *max) {
B < n-1             *max = vec[i];
      -          }
      -       }
      -    }
```

melhor caso: $f(n) = 2(n-1)$
pior caso: $f(n) = 2(n-1)$
caso médio: $f(n) = 2(n-1)$

Exemplo – MinMax2

- Se vec[i] < *min, então não precisamos checar se vec[i] > *max

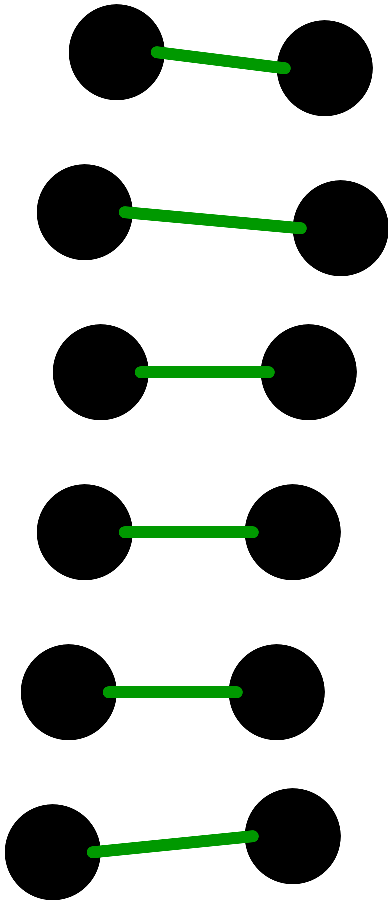
```
-         void minmax2(int *vec, int n, int *min, int *max) {
-             int i;
1             int *min = vec[0];
1             int *max = vec[0];
n-1           for(i = 1; i < n; i++) {
n-1               if(vec[i] < *min) {
A < n-1               *min = vec[i];
-                 } else {
n-1-A                 if(vec[i] > *max) {
B < n-1-A                 *max = vec[i];
-                     }
-                 }
-             }
-         }
```

melhor caso:
(decrescente)
$f(n) = n-1$

pior caso:
(crescente)
$f(n) = 2(n-1)$

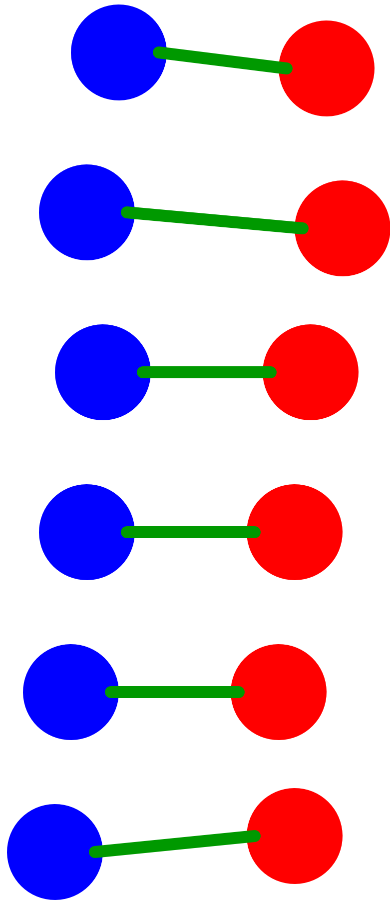
caso médio:
(aleatório)
$f(n) > 3(n-1)/2$

MinMax, dá pra fazer melhor?



- Comparar elementos par-a-par
  - Custo: $n/2$ comparações

MinMax, dá pra fazer melhor?



- Comparar elementos par-a-par
  - Custo: $n/2$ comparações
- Elementos vermelhos são maiores que os azuis
- Encontrar o máximo entre os elementos vermelhos
  - Custo: $n/2$ comparações
- Encontrar o mínimo entre os elementos azuis
  - Custo: $n/2$ comparações

Exemplo – MinMax3

```
-      void minmax3(int *vec, int n, int *min, int *max) {
-          int i;
1          int *min = INT_MAX;
1          int *max = INT_MIN;
n/2        for(i = 0; i < n; i += 2) {
n/2            if(vec[i] < vec[i+1]) {
n/4                a = i; v = i+1;
-              } else {
n/4                a = i+1; v = i;
-              }
n/2            if(vec[a] < *min)
A < n/2            *min = vec[a];
n/2            if(vec[v] > *max)
B < n/2            *max = vec[v];
-          }
-      }
```

melhor caso:

$f(n) = 3n/2$

pior caso:

$f(n) = 3n/2$

caso médio:

$f(n) = 3n/2$

Algoritmo ótimo

MinMax, comparação

Algoritmo	f(n)		
	Melhor caso	Pior caso	Caso médio
MinMax1	2(n-1)	2(n-1)	2(n-1)
MinMax2	n-1	2(n-1)	> 3(n-1)/2
MinMax3	3n/2	3n/2	3n/2

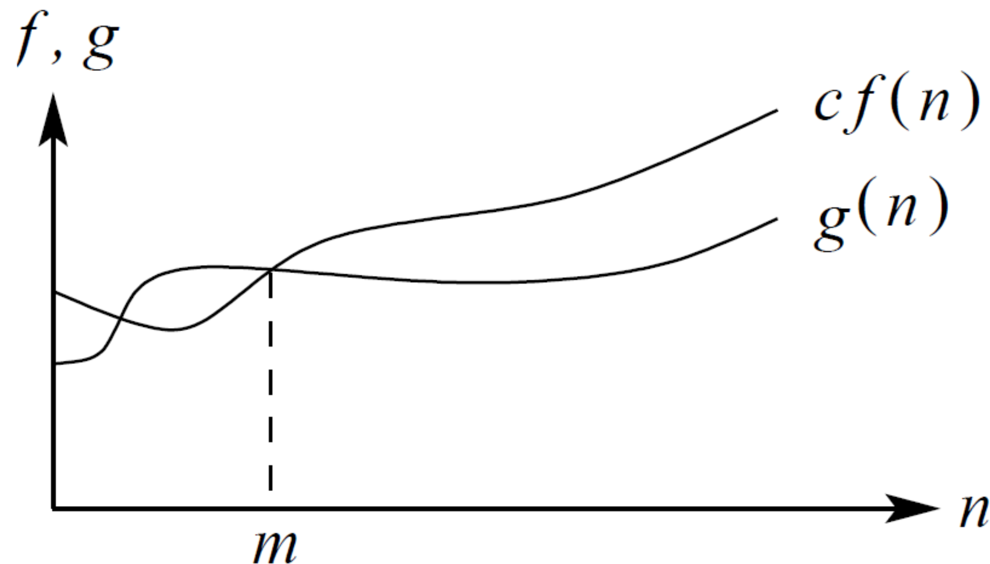
Comportamento Assintótico de Funções

Comportamento assintótico

- Para valores suficientemente pequenos de n, qualquer algoritmo custa pouco para ser executado, mesmo os ineficientes
  - Escolha de um algoritmo não é um problema crítico
- Logo, analisamos algoritmos para grandes valores de n
  - Estudamos o comportamento assintótico das funções de complexidade de um programa (comportamento pra grandes valores de n)

Dominação assintótica

- Uma função $\mathrm{f}(n)$ domina assintoticamente outra função $\mathrm{g}(n)$ se existem duas constantes positivas c e m tais que, para $n \geq m$, temos $|\mathrm{g}(n)| \leq c|\mathrm{f}(n)|$.



Exemplos

- $f(n) = n^2$, $g(n) = n$

  $f(n)$ domina assintoticamente $g(n)$
  $c = 1$, $m = 0$

  $|g(n)| \leq 1\,|f(n)|$ para todo $n \geq m = 0$

Exemplos

- $f(n) = n^2$, $g(n) = (n+1)^2$

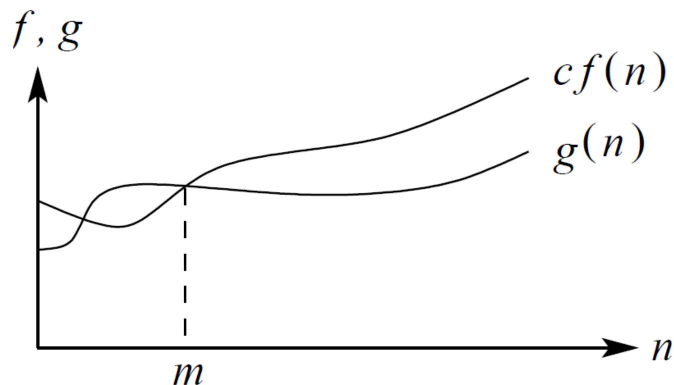
  $f(n)$ e $g(n)$ dominam assintoticamente uma à outra

  $|f(n)| \leq 1|g(n)|$ para todo $n \geq m = 0$

  $|g(n)| \leq 4|f(n)|$ para todo $n \geq m = 1$

Notação O

- Definimos $g(n) = O(f(n))$ se $f(n)$ domina assintoticamente $g(n)$
  - Lê se $g(n)$ é da ordem no máximo $f(n)$
- Quando dizemos que o tempo de execução de um programa $T(n) = O(n^2)$, existem constantes c e m tais que $T(n) \leq cn^2$ para $n \geq m$



Notação O – Exemplos (1)

- $f(n) = (n+1)^2 = O(n^2)$
  - Pois $(n+1)^2 \leq 4n^2$, para $n \geq m = 1$

- $f(n) = n^2$ e $g(n) = n$
  - $n = O(n^2)$, (faça $m = 0$ e $c = 1$)
  - Mas n^2 não é $O(n)$
    - Suponha que existam c e m tais que para todo $n \geq m$, $n^2 \leq cn$
    - Logo $n \leq c$ para todo $n \geq m$, contradição

Notação O – Exemplos (2)

- $f(n) = 3n^3 + 2n^2 + n = O(n^3)$
  - Basta mostrar que $f(n) \le 6n^3$, para $n \ge m = 0$
  - Podemos dizer que $f(n) = 3n^3 + 2n^2 + n = O(n^4)$, mas essa afirmação é mais fraca que $f(n) = O(n^3)$

- $f(n) = \log_5(n) = O(\log(n))$
  - $\log_b(n)$ difere de $\log_c(n)$ por uma constante $\log_b(c)$
  - $f(n) \le \log_5(e)\log(n)$, para todo $n \ge m = 0$

Notação O – Propriedades

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

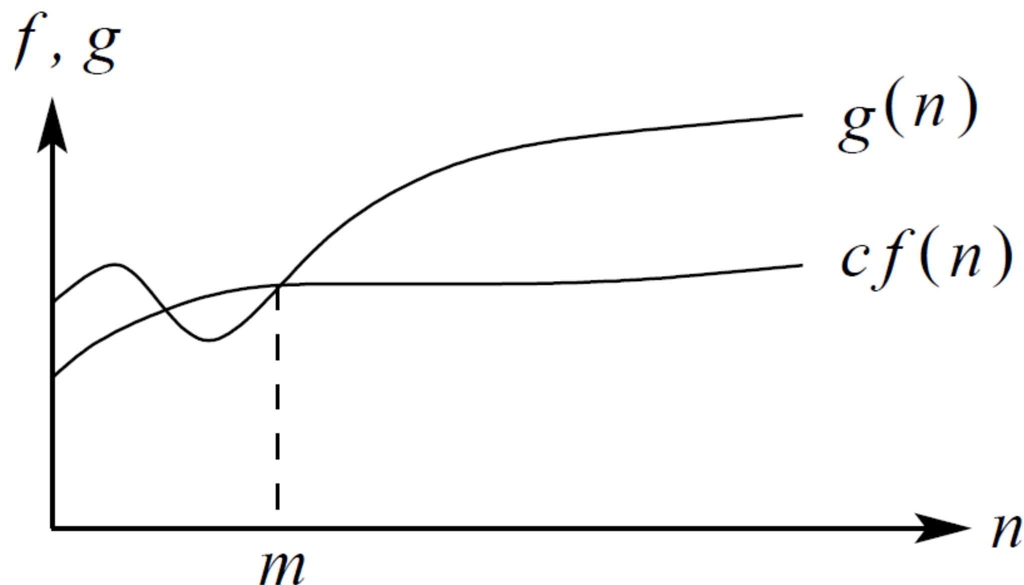
Notação O – Exemplos (3)

- Imagine um programa com três fases
  - A primeira com custo $O(n)$
  - A segunda com custo $O(n^2)$
  - A terceira com custo $O(n \log(n))$

- Aplicando a regra da soma
  - O tempo de execução total do programa é $O(n^2)$

Notação Ω

- Uma função $g(n)$ é $\Omega(f(n))$ se $g(n)$ domina assintoticamente $f(n)$
- Notação O denota um limite superior e a notação Ω denota um limite inferior

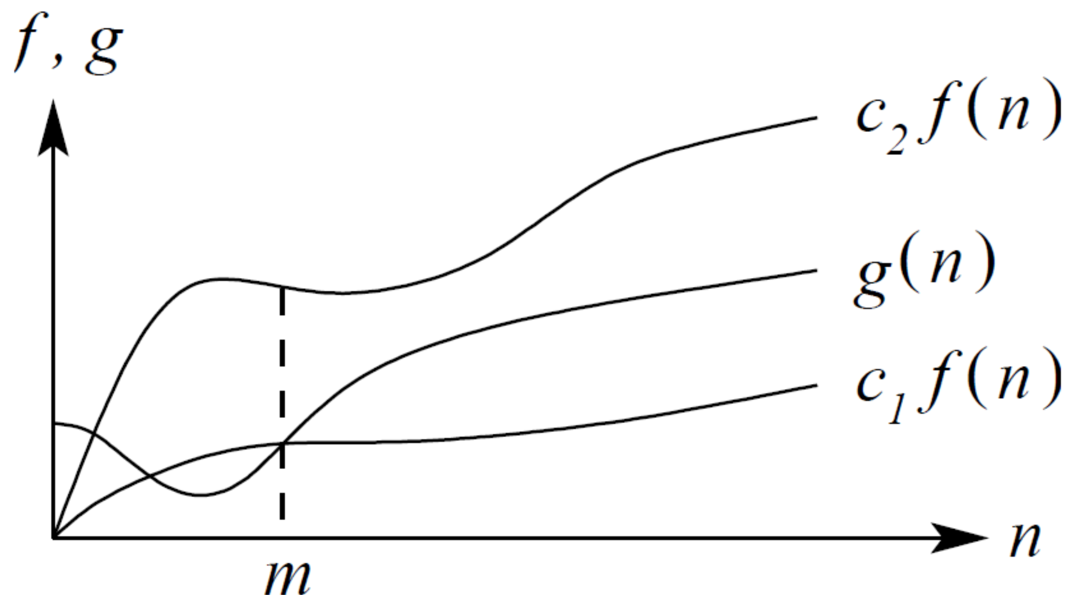


Notação Ω – Exemplo

- $\mathrm{f}(n) = 3n^3 + 2n^2 + n = \Omega(n^3)$
  - Basta mostrar que $n^3 \leq 3n^3 + 2n^2 + n$, para $n \geq m = 0$
  - Podemos dizer que $\mathrm{f}(n) = 3n^3 + 2n^2 + n = \Omega(n^2)$, mas essa afirmação é mais fraca que $\mathrm{f}(n) = \Omega(n^3)$

Notação Θ

- Uma função $g(n)$ é $\Theta(f(n))$ se $g(n)$ e $f(n)$ dominam assintoticamente uma à outra
  - Definição equivalente: $g(n) = \Theta(f(n))$ se $g(n) = O(n)$ e $g(n) = \Omega(f(n))$



Notação Θ

- Notação Θ é um limite assintótico firme
- Diz que duas funções crescem de forma similar e que a diferença é constante

Exercícios

- Prove que $4\log_2(n) + 16 = O(n)$
  - $4\log_2(n) + 16 \leq n$ para $n \geq m = 64 = 2^6$
- Prove que $4\log_2(n) + 16 = O(\log_2 n)$
  - $4\log_2(n) + 16 \leq 5\log_2(n)$ para $n \geq m = 2^{17}$
- $2^{n+1} = O(2^n)$. Verdadeiro ou falso?
  - Verdadeiro, faça $c = 2$ e $m = 0$
- $2^{2n} = O(2^n)$. Verdadeiro ou falso?
  - Falso.
  - Prova: Suponha $2^{2n} \leq c2^n$, divida por 2^n e obtenha $2^n \leq c$

Exercícios

- Por que falar "o tempo de execução do algoritmo A é pelo menos $O(2^n)$" não faz sentido?
  - Um algoritmo com tempo de execução $O(2^n)$ realiza no máximo $c2^n$ operações. Falar que um algoritmo realiza *pelo menos no máximo* $c2^n$ operações não faz sentido.

Exercícios

- Prove que $\max(f(n), g(n)) = \Theta(f(n) + g(n))$
  - $\max(f(n), g(n)) \leq 1(f(n) + g(n))$ para $n \geq m = 0$
  - $\max(f(n), g(n)) \geq (1/2)(f(n) + g(n))$, para $n \geq m = 0$

Classes de Comportamento Assintótico

Complexidade Assintótica (1)

- Se f é uma função de complexidade para um algoritmo, então $O(f)$ é considerada a complexidade assintótica do algoritmo
- Podemos comparar algoritmos usando suas complexidades assintóticas
  - Um algoritmo $O(n)$ é melhor do que um $O(n^2)$
  - Algoritmos com a mesma complexidade assintótica são equivalentes

Complexidade Assintótica (2)

- Às vezes, a constante da função de complexidade de um algoritmo importar
  - Um algoritmo com complexidade $2n^2$ é melhor do que um com complexidade $100n$ para valores de n menores que 50
  - Quando dois algoritmos têm a mesma complexidade assintótica, podemos desempatar usando a constante da função de complexidade

$f(n) = O(1)$

- Complexidade constante
- Tempo de execução do algoritmo independe do tamanho da entrada
- Os passos do algoritmo são executados um número fixo de vezes

- Exemplo: determinar se um número é ímpar

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

- Complexidade logarítmica
- Típico de algoritmos que dividem um problema transformando-o em problemas menores (dividir para conquistar)
- Tempo de execução pode ser considerado menor do que uma constante grande
  - Quando n é um milhão, $\log(n) \approx 20$
  - A base do logarítmo tem impacto pequeno
- Exemplo: busca binária

$f(n) = O(n)$

- Complexidade linear
- O algoritmo realiza um número fixo de operações sobre cada elemento da entrada
- Melhor situação para um algoritmo que processa *n* elementos de entrada e produz *n* elementos de saída

- Exemplo: busca sequencial, calcular fatorial

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

- Típico de algoritmos que dividem um problema em subproblemas, resolve cada subproblema de forma independente, e depois combina os resultados

- Exemplo: ordenação (eficiente)

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

- Complexidade quadrática
- Típico de algoritmos que operam sobre pares dos elementos de entrada
  - Comumente em um anel dentro de outro
- Útil para resolver problemas de tamanhos relativamente pequenos

- Exemplos: ordenação (ineficiente), imprimir uma matriz

$f(n) = O(n^3)$

- Complexidade cúbica
- Útil para resolver problemas pequenos

- Exemplo: multiplicação de matrizes

$f(n) = O(c^n)$

- Complexidade exponencial
- Típicos de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema
- Não são úteis do ponto de vista prático
  - Quando n é 20, $O(2^n)$ é um milhão

$f(n) = O(n!)$

- Complexidade exponencial
  - Pior do que $O(c^n)$
- Não são úteis do ponto de vista prático
  - Quando n é 20, $O(n!)$ é maior que 2 quintilhões

Comparação de Classes de Complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
3^n	0,059 s	58 min	6,5 anos	3855 séc.	10^8 séc.	10^{13} séc.

Velocidade de Processamento vs. Tamanho da Entrada

Função de custo de tempo	Computador atual (tamanho)	Computador 100 vezes mais rápido	Computador 1.000 vezes mais rápido
n	t_1	$100\ t_1$	$1000\ t_1$
n^2	t_2	$10\ t_2$	$31,6\ t_2$
n^3	t_3	$4,6\ t_3$	$10\ t_3$
2^n	t_4	$t_4 + 6,6$	$t_4 + 10$

Algoritmo Polinomial

- Algoritmo polinomial no tempo de execução tem função de complexidade $O(f(n))$, onde $f(n)$ é um polinômio
- Algoritmos polinomiais geralmente são obtidos através de um entendimento mais profundo da estrutura do problema
  - Enquanto algoritmos exponenciais são típicos de soluções força bruta
- Um problema é considerado
  - Intratável: se não existe algoritmo polinomial para resolvê-lo
  - Bem resolvido: se existe algoritmo polinomial para resolvê-lo

Algoritmos Exponenciais – Exceções

- Existem algoritmos de complexidade exponencial que são úteis
  - Por exemplo, o algoritmo *simplex* tem pior caso de tempo de execução exponencial, mas na média executa muito mais rápido do que isso
  - Infelizmente, estas exceções são incomuns e a maioria dos algoritmos exponenciais conhecidos não são muito úteis

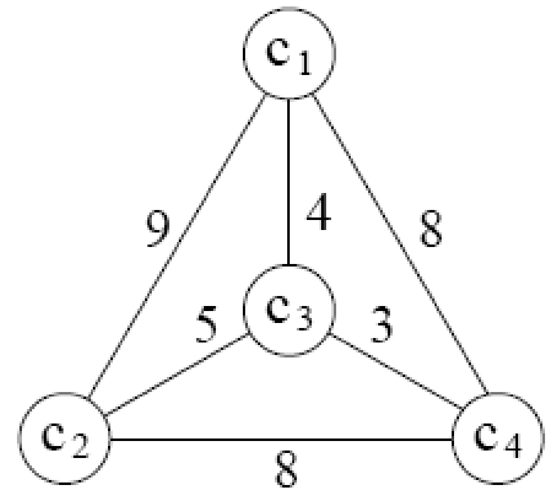
Problema do Caixeiro Viajante

- Um caixeiro viajante deseja visitar n cidades de tal forma que sua viagem inicie e termine em uma mesma cidade
- Cada cidade deve ser visitada uma única vez
- Duas cidades i, j podem ser ligadas por uma estrada de comprimento $c_{i,j}$
- O problema é encontrar a menor rota para a viagem

Problema do Caixeiro Viajante

- A figura abaixo mostra 4 cidades (c_1, c_2, c_3 e c_4) e os pesos nas arestas mostram os comprimentos de cada estrada

- O percurso c_1, c_3, c_4, c_2, c_1 é a solução para o problema, cujo percurso total tem distância 24



Problema do Caixeiro Viajante

- Um algoritmo simples seria verificar todas as rotas e escolher a menor delas
- Há $(n - 1)!$ rotas possíveis e a distância total percorrida em cada rota envolve n adições, logo o número total de adições é $n!$
- No exemplo anterior teríamos 24 adições; se tivéssemos 50 cidades, o número de adições seria aproximadamente 10^{64}
- Em um computador que executa 10^9 adições por segundo, o tempo total para resolver o problema com 50 cidades seria maior do que 10^{45} séculos só para executar as adições

Técnicas de Análise de Algoritmos

Análise de Algoritmos

- Determinar o tempo de execução de um algoritmo pode ser complexo
- Determinar a complexidade assintótica, sem preocupação com as constantes envolvidas, pode ser uma tarefa mais simples
- Análise de algoritmos utiliza técnicas de matemática discreta
  - Manipulação de somas, produtos, permutações, coeficientes binomiais, equações de recorrência

Análise do Tempo de Execução

- Comando simples (atribuição, comparação, operação aritmética, acesso a memória): $O(1)$
- Sequência de comandos: máximo dos tempos de execução dos comandos
- Comando condicional: tempo dos comandos dentro do condicional mais o tempo pra testar a condição, que é $O(1)$
- Anel: Tempo de execução dos comandos do anel mais teste de parada (geralmente $O(1)$), multiplicado pelo número de iterações

Análise do Tempo de Execução

- Para funções não recursivas:
  - Comece pelas funções que não chamam nenhuma outra função
  - Depois analise funções que chamam apenas funções analisadas no passo anterior
  - E assim sucessivamente até chegar ao programa principal (`main`)

Exemplo

```
-  int soma_acumulada(int n) {
-      int i;
1      int acumulador = 0;
n      for(i = 0; i < n; i++) {
n          acumulador += i;
-      }
1      return acumulador;
-  }
```

- Qual a função de complexidade do número de atribuições para o acumulador?
- Qual a ordem de complexidade da função **`soma_acumulada`**?

Exemplo

```
         -  void exemplo(int n)
         -  {
         -      int i, j;
         1      int a = 0;
         n      for(i = 0; i < n; i++)
  n(n+1)/2          for(j = n; j > i; j--)
  n(n+1)/2              a += i + j;
         1      exemplo1(n);
         -  }
```

- Qual a complexidade assintótica da função **`exemplo`**?

Exemplo – Ordenação

- Encontre o menor valor no vetor
- Troque-o com o primeiro elemento `V[0]`
- Repita os passos acima com os n-1 itens restantes, depois com os n-2 restantes, até que reste apenas 1

Exemplo – Ordenação

```
            -  void ordena(int *V, int n) {
            -      int i, j, min, x;
          n-1      for(i = 0; i < n - 1; i++) {
          n-1          min = i;
     n(n-1)/2          for(j = i + 1; j < n; j++)
     n(n-1)/2              if(V[j] < V[min])
 A < n(n-1)/2                  min = j;
            -          /* troca A[min] e A[i]: */
          n-1          x = V[min];
          n-1          V[min] = V[i];
          n-1          V[i] = x;
            -      }
            -  }
```

Qual a complexidade assintótica do número de comparações?

Exercício 1

```
-      // A, B e C sao vetores globais
-      void e1(int n) {
-         int i, j, k;
n         for(i = 0; i < n; i++)
n*n          for(j = 0; j < n; j++) {
n*n             C[i][j] = 0;
n*n*n           for(k = n-1; k >= 0; k--) {
n*n*n              C[i][j] = C[i][j] +
                      A[i][k] * B[k][j];
-               }
-            }
-         }
-      }
```

O que faz essa função? Qual sua complexidade assintótica?

Exercício 2

```
      -   void e2(int n)
      -   {
      -       int i, j, x, y;
      1       x = y = 0;
      n       for(i = 1; i <= n; i++) {
n(n+1)/2          for(j = i; j <= n; j++)
n(n+1)/2              x = x + 1;
n(n-1)/2          for(j = 1; j < i; j++)
n(n-1)/2              y = y + 1;
      -       }
      -   }
```