Análise de algoritmos

José Eduardo De Lucca
Depto Informática – UFSC
delucca@inf.ufsc.br

Projeto de algoritmos

- **Em geral, há mais de uma solução**
  - **Melhor algoritmo para solucionar problema**
- **Técnicas básicas**
  - **Dividir para conquistar**
  - **Algoritmos aleatorizados**
  - **programação dinâmica**
  - **algoritmos gulosos**
  - **heurísticas**
  - **redução a outro problema**
  - **estruturas de dados**

O que analisar em um algoritmo?

Eficiência, custo

- Tempo
  - atualmente, o mais importante
  - quanto tempo para resolver um problema
  - função da quantidade ou tamanho da entrada
- Espaço
  - já foi crítico, hoje menos
  - quanta memória necessita
  - função da quantidade ou tamanho da entrada

Pensemos: algoritmo para jogar xadrez

Tempo de execução

- **Tempo de execução**
  - **varia com a entrada**
  - **cresce com o tamanho da entrada**
- **Difícil definir o "caso médio"**
- **Foco no "pior caso"**
  - **mais fácil de analisar**
  - **fundamental para algumas aplicações**
    - **jogos, finanças, robótica**

Estudos experimentais

1. Escrever um programa que implemente o algoritmo sendo estudado
2. Executar o programa com entradas de tamanhos e composições diferentes
3. Usar uma rotina para obter medida de tempo precisa
   - System.currentTimeMillis()
4. Traçar os resultados

Experimental - limitações

- Necessário implementar o algoritmo
- Resultados podem não representar todo o universo de entradas possível
- Resultados podem estar associados à (in)competência do implementador
- Para comparação, o mesmo hardware e software devem ser utilizado

Análise teórica

- Usa uma descrição de alto-nível do algoritmo
- Leva em consideração todas as entradas possíveis
- Permite avaliar a velocidade independente do ambiente hard/soft

Modelos computacionais

- **Estudo de algoritmos**
  - **modelo abstrato completo**
  - **prova, análise e comparação de algoritmos**
- **Máquina de Turing (Alan Turing)**
  - **modelo mais básico**
  - **autômato finito**
  - **não adequado para avaliação de custos reais em um programa.**

Modelos computacionais (ii)

- **Máquina RAM de custo fixo**
  - simplificação de computador real
  - Custo (tempo) fixo
    - todos os acessos à memória
    - todas as operações
  - conjunto de instruções básicas
    - atribuições/acessos, vetores
    - if, while, for
  - Muito mais realista que M.Turing
  - Problema
    - irreal quanto ao custo de cada instrução

```
a <- 3
b <- a*5
if b == 5 then
    a <- 2
else
    a <- 1
end if
```

Máquina RAM de custo variável

- Cada instrução tem custo próprio
- Custo (tempo) do exemplo

  3 C1 + C2 + C3

  – C1 – custo de atribuição

  – C2 – custo de multiplicação

  – C3 – custo de teste

```
a <- 3
b <- a*5
if b == 5 then
      a <- 2
else
      a <- 1
end if
```

Exercício cálculo x^y – Potencia (x.y)

- **Requer: x>0**

```
if y==0 then
    return 1
else
    result <- x
    for I = 1 to y-1 do
        result <- result * x
    end for
end if
return result
```

Calcular o custo (tempo) supondo que y > 0 (o que ocorrerá na maioria das vezes)

C1 = custo de teste
C2 = custo de atribuição
C3 = custo de multiplicação

C1 + C2 + (y-1)*(C3+C1)
Eficiente?
$2^8 = 2*2*2*2*2*2*2*2$

Outro algoritmo? Potencia2(x.y)

- **Requer x>0**

```
if y==0 then
    return 1
end if
r <- 1
if y>1 then
    r <- Potencia2(x.y/2)
    r <- r*r
end if
if y mod 2 == 1 then
    r <- r*x
end if
return r
```

Melhora?

Avaliar c/ Potencia2(2.7)

Qual é o custo (tempo)?

Operações primitivas

- Independentes de linguagem
- Computações básicas de um algoritmo
- Pseudo-código
  - Avaliação de expressão
  - Atribuição de valor a variável
  - Atribuição de valor para posição de vetor
  - Chamar uma rotina
  - Retornar de uma rotina

Contando operações primitivas

- MaiorDoArray(A, n)

```
atual <- A[0]
for i <- 1 to n-1 do
    if A[i] > atual then
        atual <- A[i]
    end if
{incremento do i}
return atual
```

- Nº de operações

```
2
2 + n
(n – 1)*2
(n – 1)*2

(n – 1)*2
1
Total 7n–1
```

Ordem de um algoritmo

- Cálculo do custo incômodo
- Difícil para comparação entre algoritmos
- Importa quanto demora EXATAMENTE um algoritmo?
- Importa a relação dados X tempo
- Ordem de um algoritmo
  - taxa de crescimento do tempo em função da quantidade ou tamanho dos dados de entrada

Tempo de execução estimado

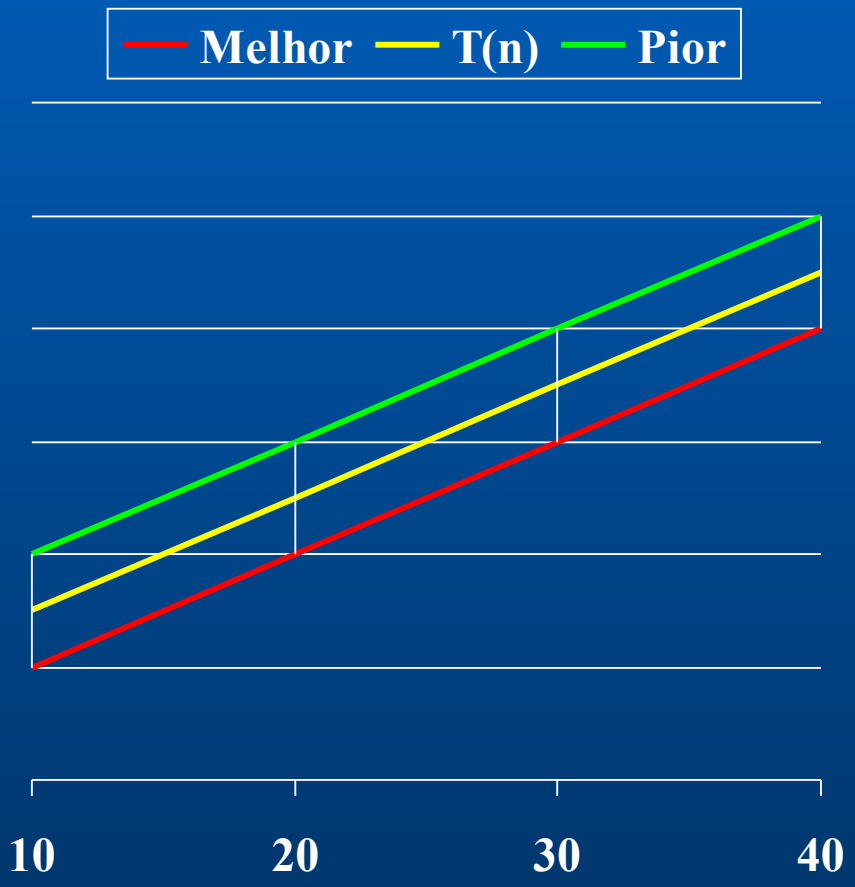
- O algoritmo anterior executa 7n-1 operações primitivas no pior caso
- Considere:
  - *"a"* tempo da op. primitiva mais rápida
  - *"b"* tempo da op. primitiva mais lenta
- Seja *T(n)* o tempo real de execução do pior caso do algoritmo

$$a(7n-1) <= T(n) <= b(7n-1)$$

- Ou seja, o tempo T(n) está limitado pelas duas funções **lineares**

Tempo de execução estimado

- T(m) varia dentro da área definida por Melhor – Pior
- Tempo cresce com o tamanho dos dados
- Não importa máquina nem implementação



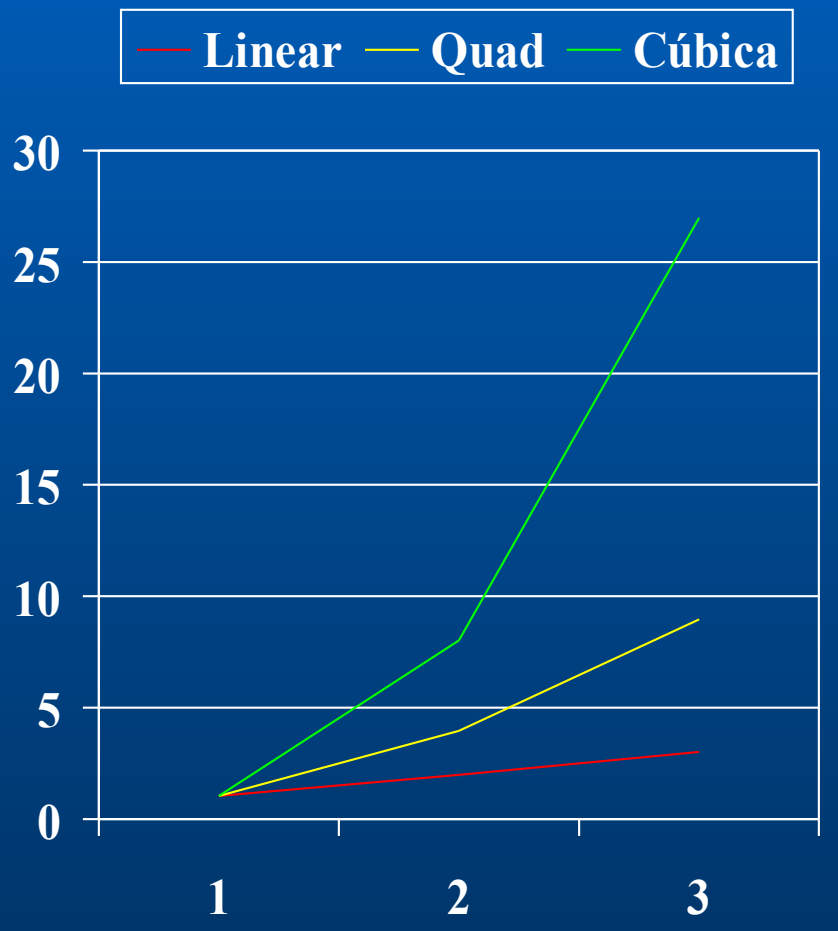
Taxa de crescimento do tempo de execução

- Alterar o ambiente de hardware/software
  - afeta *T(n)* com um fator constante
  - Mas não altera sua a taxa de crescimento
- Taxa de crescimento linear
  - propriedade intrínseca daquele algoritmo

Funções de taxas de crescimento

- Linear ~ n
- Quadrática ~ n^2
- Cúbica ~ n^3

- Taxa de crescimento não é afetada por
  - fatores constantes
  - termos de menores ordens



Notação Big-O

- Dadas duas funções f(n) e g(n)
- Diz-se que

  f(n) é O(g(n))
- se

  f(n) <= k*g(n), com n >= n0
- Ou seja
  - se f é “menor” que g multiplicado por uma constante K, para um valor de n maior que um limite n0

Notação Big-O (ii)

- **Quais funções estamos comparando?**
  - **f(n) e g(n) são algoritmos**
- **Para nosso uso**
  - **f(n) é o algoritmo que estamos estudando**
  - **g(n) será uma função básica**
    - **linear**
    - **quadrática**
    - **cúbica, etc**

Notação Big-O (iii)

- f(n) é O(g(n)) significa que
  - a taxa de crescimento de f(n) não é maior que a taxa de crescimento de g(n)
- então
  - se f(n) é O(n), quero dizer que
    - a taxa de crescimento de f(n) é menor ou igual a uma taxa de crescimento linear
  - se f(n) é $O(n^2)$, quero dizer que
    - a taxa de crescimento de f(n) é no máximo igual a uma taxa de crescimento quadrática

Regras do Big-O

- Se f(n) é um polinômio de grau d
  - f(n) é $O(n^d)$ ou seja
    - joga-se fora termos de mais baixa ordem
    - despreza-se os fatores constantes
- Use a menor classe de funções
  - 2n é O(n) em vez de 2n ser O(2n)
- Use a expressão de classe mais simples
  - 3n+5 é O(n) em vez de 3n+5 ser O(3n)

Análise assintótica

- Define o tempo de execução de um algoritmo
  - usando a notação Big-O
- Análise assintótica
  - encontra-se o pior caso com operações primitivas em função do tam. da entrada
  - expressa-se esta função com notação Big-O
- Aquele nosso algoritmo MaiorDoArray
  - pior caso: 7n-1 operações primitivas
  - MaiorDoArray executa em tempo O(n)

Comparação de algoritmos

- Faz-se a análise assintótica de cada algoritmo
- Compara-se diretamente os resultados
- Algumas funções típicas por ordem de crescimento:

  $k < \sqrt{n} < \log n < n < n \log n < n^c < x^n < n! < n^n$

- Quanto mais rápido o crescimento, pior o desempenho do algoritmo

Exemplo de comparação

- Considerando 2 algoritmos de ordenação
  - Com o desempenho *fa(n) = 100n* e *fb(n) = n^2*
  - Sendo n = número de elementos
- Qual é "melhor"?
  - Se tomarmos 30 elementos:
    - Fa(30) = 3000 e fb(30) = 900
  - Se tomarmos 30.000 elementos:
    - Fa(30000) = 3.000.000
    - Fb(30000) = 900.000.000

Análise dos algoritmos de algumas estruturas

Pilha

- Criar – O(1)
- Esvaziar – O(1)
- Top – O(1)
- Push – O(1)
- Pop – O(1)

Filas

- Criar – O(1)
- Entrar – O(1)
- Sair – O(1)

Deque – Fila dupla

- Criar – O(1)
- Entrar-cabeça – O(1)
- Sair-cabeça – O(1)
- Entrar-cauda – O(1)
- Sair-cauda – O(1)

Listas

- Criar – O(1)
- Inserir – O(1) (sempre?)
- Excluir – O(n) (por quê?)
- Buscar – O(n) (por quê?)

Árvores binárias

- Criar – O(1)
- Inserir – O(h)
- Percorrer – O(n)
- Buscar – O(h)
- Excluir – O(h)

onde

n é o número de elementos

h é a altura da árvore

- válido para árvores balanceadas

- Criar – O(1)
- Inserir – O(log n)
- Percorrer – O(n)
- Buscar – O(log n)
- Excluir – O(log n)

onde

n é o número de elementos

h é altura (log n)

- válido para árvores AVL
- custo de balancear é um fator constante em inserir e excluir

Algoritmos de ordenação

- **Bolha**
  - O (n^2) – pior caso
- **Seleção Direta**
  - O (n^2) – pior caso
- **Inserção Direta**
  - O (n^2) – pior caso
  - Mas O (n) no melhor caso
- **Quicksort**
  - O (n log n)
    - mas pode ser O(n^2)