Análise de algoritmos

Introdução à Ciência de Computação II

Baseados nos Slides do Prof. Dr. Thiago A. S. Pardo

Análise de algoritmos

- Existem basicamente 2 formas de estimar o tempo de execução de programas e decidir quais são os melhores
  - Empírica ou teoricamente

- É desejável e possível estimar qual o melhor algoritmo sem ter que executá-los
  - Função da análise de algoritmos

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

<u>Início</u>

declare soma_parcial numérico;

soma_parcial ← 0;

para i←1 até n faça

    soma_parcial←soma_parcial+i*i*i;

escreva(soma_parcial);

<u>Fim</u>

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

<u>Início</u>

declare soma_parcial numérico;

soma_parcial ← 0;

para i←1 até n faça

    soma_parcial←soma_parcial+i*i*i;

escreva(soma_parcial);

<u>Fim</u>

1 unidade de tempo

1 unidade para inicialização de i, n+1 unidades para testar se i≤n e n unidades para incrementar i = 2n+2

4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando “para”) = 4n unidades

1 unidade para escrita

Calculando o tempo de execução

- Supondo que as operações simples demoram uma unidade de tempo para executar, considere o programa abaixo para calcular o resultado de $\sum_{i=1}^{n} i^3$

<u>Início</u>

declare soma_parcial numérico;

soma_parcial ← 0;

para i←1 até n faça

parcial+i*i*i;

1 unidade de tempo

1 unidade para inicialização de i, n+1 unidades para testar se i≤n e n unidades para incrementar i = 2n+2

4 unidades (1 da soma, 2 das multiplicações e 1 da atribuição) executada n vezes (pelo comando "para") = 4n unidades

1 unidade para escrita

Custo total: somando tudo, tem-se 6n+4 unidades de tempo, ou seja, a função é **O(n)**

Calculando o tempo de execução

- Ter que realizar todos esses passos para cada algoritmo (principalmente algoritmos grandes) pode se tornar uma tarefa cansativa

- Em geral, como se dá a resposta em termos do *big-oh*, costuma-se desconsiderar as constantes e elementos menores dos cálculos
  - No exemplo anterior
    - A linha soma_parcial←0 é insignificante em termos de tempo
    - É desnecessário ficar contando 2, 3 ou 4 unidades de tempo na linha soma_parcial←soma_parcial+i*i*i
    - O que realmente dá a grandeza de tempo desejada é a repetição na linha **para i←1 até n faça**

Regras para o cálculo

- Repetições
  - O tempo de execução de uma repetição é pelo menos o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada

Regras para o cálculo

- Repetições aninhadas
  - A análise é feita de dentro para fora
  - O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições
  - O exemplo abaixo é $O(n^2)$

```
para i←0 até n faça
    para j←0 até n faça
            faça k←k+1;
```

Regras para o cálculo

- Comandos consecutivos
  - É a soma dos tempos de cada um, o que pode significar o máximo entre eles
  - O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser $O(n)$

```
para i←0 até n faça
    k←0;
para i←0 até n faça
    para j←0 até n faça
        faça k←k+1;
```

Regras para o cálculo

- Se... então... senão

  - Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao <u>então</u> e os comandos relativos ao <u>senão</u>

  - O exemplo abaixo é O(n)

```
se i<j
    então i←i+1
    senão para k←1 até n faça
                            i←i*k;
```

Regras para o cálculo

- Chamadas a sub-rotinas
  - Uma sub-rotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/sub-rotina que a chamou

Exercício

- Analise a sub-rotina recursiva abaixo

```
sub-rotina fatorial(n: numérico)
início
    declare aux numérico;
    se n≤1 então
       aux←1
    senão
       aux←n*fatorial(n-1);
    retorne aux;
fim
```

Regras para o cálculo

- Sub-rotinas recursivas
  - Se a recursão é um “disfarce” da repetição (e, portanto, a recursão está mal empregada, em geral), basta analisá-la como tal
  - O exemplo anterior é obviamente O(n)

```
sub-rotina fatorial(n: numérico)
início
declare aux numérico;
se n≤1
        então aux←1
        senão aux←n*fatorial(n-1);
retorne aux;
fim
```

Eliminando a recursão →

```
sub-rotina fatorial(n: numérico)
início
declare aux numérico;
aux←1;
enquanto n>1 faça
        aux←aux*n;
        n←n-1;
retorne aux;
fim
```

Regras para o cálculo

- Sub-rotinas recursivas
  - Em muitos casos (incluindo casos em que a recursividade é bem empregada), é difícil transformá-la em repetição
    - Nesses casos, para fazer a análise do algoritmo, pode ser necessário se recorrer à análise de recorrência
    - *Recorrência: equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores*
      - Caso típico: algoritmos de **dividir-e-conquistar**, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original
        - Exemplos?

Regras para o cálculo

- Exemplo de uso de recorrência
  - Números de Fibonacci
    - 0,1,1,2,3,5,8,13...
    - $f(0)=0$, $f(1)=1$, $f(i)=f(i-1)+f(i-2)$

```
sub-rotina fib(n: numérico)
início
declare aux numérico;
se n≤1
    então aux←1
    senão aux←fib(n-1)+fib(n-2);
retorne aux;
fim
```

Regras para o cálculo

- Exemplo de uso de recorrência
  - Números de Fibonacci
    - 0,1,1,2,3,5,8,13...
    - f(0)=0, f(1)=1, f(i)=f(i-1)+f(i-2)

sub-rotina fib(n: numérico)
<u>início</u>
declare aux numérico;
se n≤1
    então aux←1
    senão aux←fib(n-1)+fib(n-2);
retorne aux;
<u>fim</u>

Seja T(n) o tempo de execução da função.

<u>Caso 1</u>:
Se n=0 ou 1, o tempo de execução é constante, que é o tempo de testar o valor de n no comando se, mais atribuir o valor 1 à variável aux, mais o retorno da função; ou seja, T(0)=T(1)=3.

Regras para o cálculo

- Exemplo de uso de recorrência
  - Números de Fibonacci
    - 0,1,1,2,3,5,8,13...
    - f(0)=0, f(1)=1, f(i)=f(i-1)+f(i-2)

sub-rotina fib(n: numérico)
<u>início</u>
declare aux numérico;
se n≤1
    então aux←1
    senão aux←fib(n-1)+fib(n-2);
retorne aux;
<u>fim</u>

<u>Caso 2</u>:
Se n>2, o tempo consiste em testar o valor de n no comando se, mais o trabalho a ser executado no senão (que é uma soma, uma atribuição e 2 chamadas recursivas), mais o retorno da função; ou seja, a recorrência $T(n)=T(n-1)+T(n-2)+4$, para n>2.