Análise e Síntese de Algoritmos

Revisão [CLRS, Cap. 1-3]

2011/2012

Resumo

Algoritmos

Algoritmo

- Procedimento computacional bem definido que aceita uma dada entrada e produz uma dada saída
  - Ferramenta para resolver um problema computacional bem definido
    - Ordenação de sequências de valores
    - Caminhos mais curtos em grafos dirigidos
    - etc.

Pseudo-Código

- Utilizado na descrição de algoritmos
- Apresentar os detalhes essenciais de um algoritmo sem a verbosidade das linguagens de programação

Exemplo: Ordenação

- Entrada: sequência de valores $A[1..n]$
- Objectivo: ordenar valores em A de forma crescente
- Saída: sequência de valores ordenados $A[1..n]$

InsertionSort

```
InsertionSort(A)
for j = 2 to length[A]
        do key = A[j]
           i = j - 1
           while i > 0 and A[i] > key
               do A[i+1] = A[i]
                  i = i - 1
           A[i+1] = key
```

Análise de Algoritmos

- Como aferir a complexidade um algoritmo?
- Como comparar dois algoritmos diferentes?
  - Notação assimptótica
- Que modelo computacional utilizar?
  - Modelo RAM (Random Access Machine)
  - Execução sequencial de instruções
  - Apenas 1 processador
  - Outros modelos computacionais relacionados polinomialmente com modelo RAM

Análise de Algoritmos

Medidas de Complexidade

- Tempo necessário (execução)
- Espaço necessário

- Tanto o tempo como o espaço dependem do tamanho da entrada
- Entrada depende do problema que o algoritmo pretende resolver
- Exemplo: No InsertionSort uma medida razoável é o número de elementos a ordenar
- Exemplo: Num grafo as medidas utilizadas são o número de vértices e o de arcos

Tempo de Execução

n	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$	$f^a(n) = n!$
10	0.01 μs	0.10 μs	1.00 μs	3.63 ms
20	0.02 μs	0.40 μs	1.00 ms	77.1 anos
30	0.03 μs	0.90 μs	1.00 s	$8.4 * 10^{15}$ anos
40	0.04 μs	1.60 μs	18.3 min	
50	0.05 μs	2.50 μs	13 dias	
100	0.10 μs	10 μs	$4 * 10^{13}$ anos	
1000	1.00 μs	1 ms		

Tempo de Execução

- Exemplo: Algoritmo InsertionSort
  - c_i: custo de executar a instrução i
  - t_j: número de vezes que ciclo `while é executado para cada` j, j=2, . . . , n

InsertionSort

InsertionSort(A)

		▷ Custo	#Exec.
1	**for** j = 2 **to** length[A]	▷ c_1	n
2	**do** key = A[j]	▷ c_2	$n-1$
3	i = j - 1	▷ c_3	$n-1$
4	**while** i > 0 and A[i] > key	▷ c_4	$\sum t_j : j = 2..n$
5	**do** A[i+1] = A[i]	▷ c_5	$\sum t_j - 1 : j = 2..n$
6	i = i - 1	▷ c_6	$\sum t_j - 1 : j = 2..n$
7	A[i+1] = key	▷ c_7	$n-1$

Tempo de Execução

- Exemplo: Algoritmo InsertionSort
  - $T(n)$: tempo de execução do algoritmo em função do número de elementos a ordenar

$$\begin{aligned} T(n) = \quad & c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + \\ & c_5 \sum_{j=2}^{n}(t_j - 1) + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7(n-1) \end{aligned}$$

  - O tempo de execução depende da sequência a ordenar !

Tempo de Execução

Análise melhor-caso

- Sequência de entrada já ordenada

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

- $T(n)$ é função linear de n

Análise pior-caso

- Sequência de entrada ordenada por ordem inversa
- $t_j = j$ para $j = 2 \ldots n$

$$T(n) = \alpha n^2 + \beta n + \gamma$$

- $T(n)$ é função quadrática de n

Análise pior-caso

Uso do pior-caso como valor para a complexidade

- Representa um limite superior/inferior no tempo de execução
- Ocorre numerosas vezes
- O valor médio é muitas vezes próximo do pior-caso
- Geralmente é mais fácil de calcular

Caso-médio

- Importante em algoritmos probabilísticos
- É necessário saber a distribuição dos problemas

Síntese de Algoritmos

- Dividir para conquistar
- Programação Dinâmica
- Algoritmos ganânciosos (greedy)

Ordenação — Dividir para Conquistar

MergeSort

```
MergeSort(A, p, r)
if p < r
    then q = ⌊(p + r)/2⌋
         MergeSort(A, p, q)
         MergeSort(A, q+1, r)
         Merge(A, p, q, r)
```

- No pior caso, tempo de execução cresce com $n \, lgn$
    - Admitindo que tempo de execução de Merge cresce com n

Exemplo...

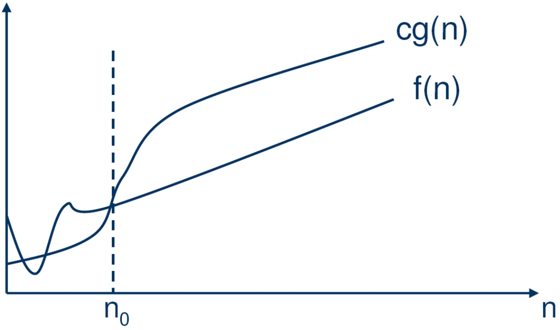
Notação Assimptótica

- Objectivo é caracterizar tempos de execução dos algoritmos para tamanhos arbitrários das entradas
- A notação assimptótica permite estabelecer taxas de crescimento dos tempo de execução dos algoritmos em função dos tamanhos das entradas
- Constantes multiplicativas e aditivas tornam-se irrelevantes
  - Nota: Tempo de execução de cada instrução não é essencial para o comportamento assimptótico de um algoritmo
- Simbolos da notação assimptótica
  - Θ, O, Ω
  - o, ω

Notação Assimptótica

Notação O: Limite Assimptótico Superior

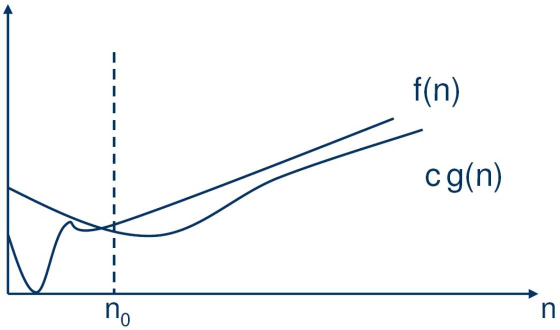
- $O(g(n)) = \{f(n)$: existem constantes positivas c e n_0, tal que $0 \leq f(n) \leq cg(n)$, para $n \geq n_0\}$
- $f(n) = O(g(n))$, significa $f(n) \in O(g(n))$



Notação Assimptótica

Notação Ω: Limite Assimptótico Inferior

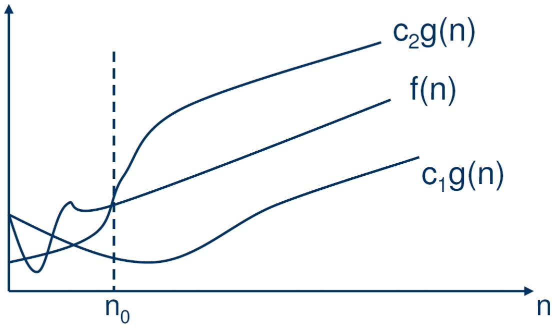
- $\Omega(g(n)) = \{f(n)$: existem constantes positivas c e n_0, tal que $0 \leq cg(n) \leq f(n)$, para $n \geq n_0\}$
- $f(n) = \Omega(g(n))$, significa $f(n) \in \Omega(g(n))$



Notação Assimptótica

Notação Θ: Limite Assimptótico Apertado

- $\Theta(g(n)) = \{f(n)$: existem constantes positivas c_1, c_2 e n_0, tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, para $n \geq n_0\}$
- $f(n) = \Theta(g(n))$, significa $f(n) \in \Theta(g(n))$



Notação Adicional

- Floor e Ceiling: $\lfloor \quad \rfloor \quad \lceil \quad \rceil$
- Polinónios
  - Um polinómio cresce com o maior grau que o compõe
- Exponenciais
  - Uma exponencial (base > 1) cresce mais depressa do que um polinómio
- Logaritmos

Somatórios

- Utilizados no cálculo do tempo de execução de ciclos: $\sum_{k=1}^{n} a_k$
- Propriedades
  - Linearidade : $\sum_{k=1}^{n} \Theta(f(k)) = \Theta(\sum_{k=1}^{n} f(k))$
  - Telescópica : $\sum_{k=1}^{n} (a_k - a_{k-1}) = a_n - a_0$
- Limite de somatórios
  - Indução matemática
  - Limitar termos
  - Dividir somatório
  - Aproximação por integrais

Problemas

Problema 1

Dado um vector com n valores, propor um algoritmo eficiente que determina a existência de dois números no vector cuja soma é x.

Problemas

Problema 2

Dado um vector com n valores ordenados, propor um algoritmo eficiente que determina a existência de dois números no vector cuja soma é x.

Problemas

Problema 2

Dado um vector com n valores ordenados, propor um algoritmo eficiente que determina a existência de dois números no vector cuja soma é x.

Solução - Problema 2

- Utilizar dois apontadores colocados inicialmente na primeira (i) e na última (j) posições do vector
- Se soma das posições apontadas é superior a x, decrementar j
- Se soma das posições apontadas é inferior a x, incrementar i
- Se soma das posições apontadas é igual a x, terminar

Complexidade: $O(n)$

Problemas

Problema 1

Dado um vector com n valores, propor um algoritmo eficiente que determina a existência de dois números no vector cuja soma é x.

Solução - Problema 1

- Ordenar vector e resolver problema 2

Complexidade: $O(n\ lgn)$