Análise e Síntese de Algoritmos

Revisão

CLRS, Cap. 1-4

Resumo

- Algoritmos
- Análise de algoritmos
- Síntese de algoritmos
- Notação assimptótica
- Outra notação utilizada
- Somatórios
- Recorrências
  - Exemplos
- Dois problemas

Algoritmos

- 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.

- Descrição de algoritmos utilizando pseudo-código
  - Apresentar os detalhes essenciais de um algoritmo sem a verbosidade das linguagens de programação

Um Exemplo: Ordenação

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

```
InsertionSort(A)
    for j = 2 to length[A]
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key
            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
    - Tempo de execução
  - Espaço necessário

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

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

$$
\begin{aligned}
T(n) \quad = \quad & c_1 n + c_2(n-1) + c_4(n-1) + \\
& + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)
\end{aligned}
$$

  - O tempo de execução depende da sequência a ordenar **!**

Tempo de Execução

- Análise melhor-caso:
  - Sequência de entrada já ordenada

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \\ &\quad + c_5(n-1) + c_8(n-1) \end{aligned}$$

  - 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 do 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
  - É, geralmente, 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 gananciosos

Ordenação — Dividir para Conquistar

```
MergeSort(A, p, r)
1.      if p < r
2.          q = ⌊(p+r) / 2⌋
4.          MergeSort(A, p, q)
5.          MergeSort(A, q+1, r)
6.          Merge(A, p, q, r)
```

- No pior caso, tempo de execução cresce com n log n
  - Admitindo que tempo de execução de Merge cresce com n

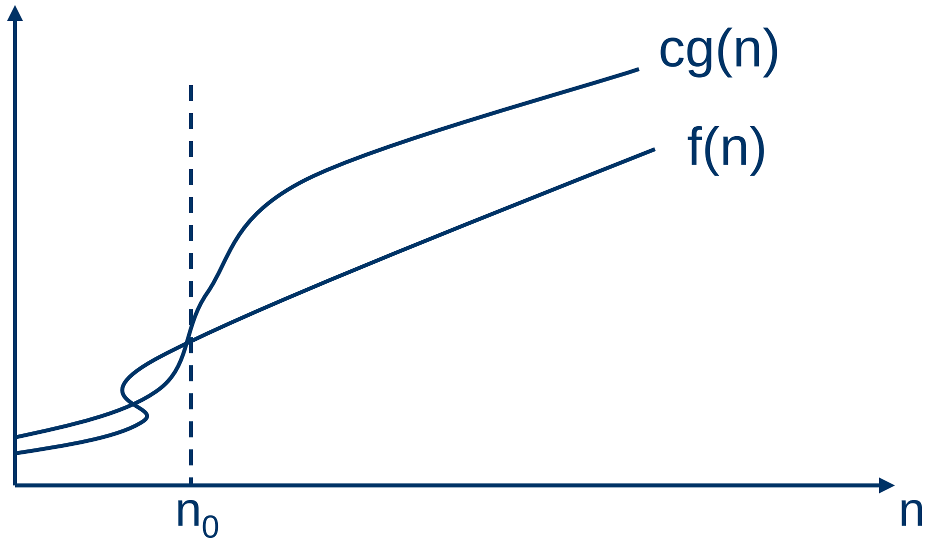
- Exemplo
- Merge ?

Notação Assimptótica

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

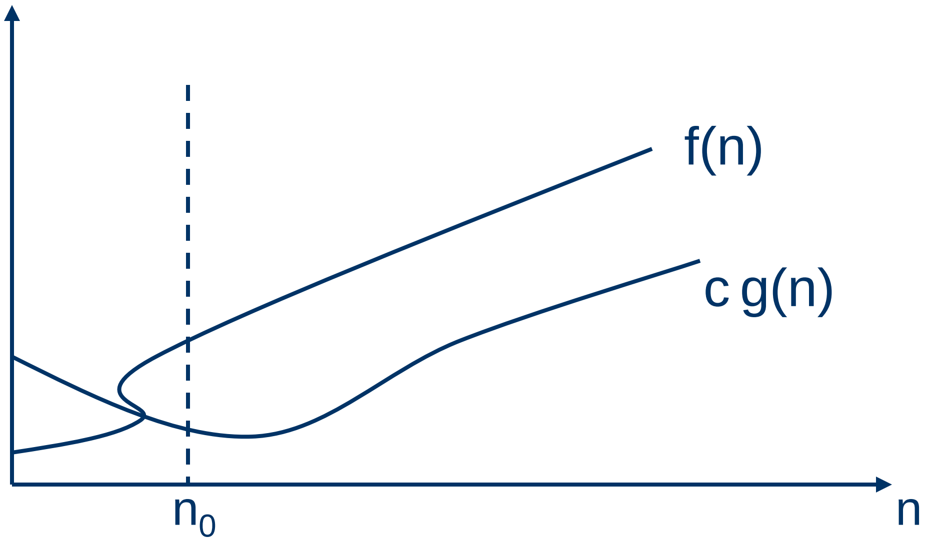
Notação Assimptótica

- 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

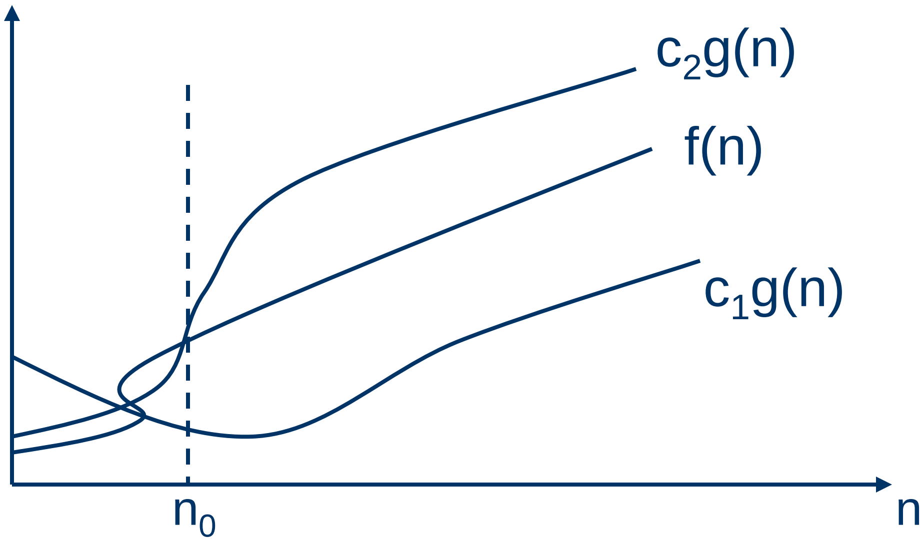
- Notação Ω: Limite Assimptótico Inferior
  - $\Omega(g(n))$ =
    { f(n) : existem constantes positivas c e n_0, tal que $0 \leq c\, g(n) \leq f(n)$, para $n \geq n_0$ }
  - $f(n) = \Omega(g(n))$, significa $f(n) \in \Omega(g(n))$



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 $f(n) \in \Theta(g(n))$



Mais Notação

- Floor & Ceiling: $\lfloor\ \rfloor$, $\lceil\ \rceil$
- Polinómios
  - 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

$$\sum_{k=1}^{n} \Theta(f(k)) = \Theta\left(\sum_{k=1}^{n} f(k)\right)$$

  - Telescópica
- Limite de somatórios
  - Indução matemática
  - Limitar termos
  - Dividir somatório
  - Aproximação por integrais

Recorrências

- Utilizadas no cálculo do tempo de execução de procedimentos recursivos

- Resolução de recorrências:
  - Por substituição
    - Sugestão de solução
    - Provar solução utilizando indução
  - Por iteração
    - Expandir recorrência com o intuito de a expressar em termos apenas de n e das condições iniciais
  - Teorema Mestre

Recorrências

- Teorema Mestre:
  - Permite resolver recorrências da forma $T(n) = a\,T(n/b) + f(n)$, $a \geq 1$, $b \geq 1$
    - a sub-problemas, cada com tamanho n/b
  - Teorema:
    - Sejam $a \geq 1$, $b \geq 1$ constantes, seja $f(n)$ uma função e seja $T(n)$ definido por $T(n) = a\,T(n/b) + f(n)$. Então $T(n)$ é limitado assimptoticamente da seguinte forma:
      - Se $f(n) = O(n^{\log_b a-\varepsilon})$ para $\varepsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$
      - Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \lg n)$
      - Se $f(n) = \Omega(n^{\log_b a+\varepsilon})$, e $a \cdot f(n/b) < c\,f(n)$ para alguma constante $c < 1$ e n suficientemente grande, então $T(n) = \Theta(f(n))$

Exemplo de aplicação

- $T(n) = 9\, T(n/3) + n$
  - $a=9$, $b=3$, $f(n)=n$
  - $f(n) = O(n^{2-\varepsilon})$
  - $T(n) = \Theta(n^2)$
- $T(n) = T(2n/3) + 1$
  - $a=1, b=3/2$, $f(n)=1$
  - $f(n) = \Theta(n^0)$
  - $T(n) = \Theta(\lg n)$

Exemplos de Recorrências

- Torres de Hanoi:
  - 3 suportes
  - n discos colocados por ordem decrescente de tamanho num dado suporte
  - Qual o menor número de movimentos de disco individuais que é necessário e suficiente para deslocar os n discos para outro suporte, mantendo sempre a relação de ordem entre os discos?

  - T_n: menor número de movimentos para transferir os n discos

Torres de Hanoi

- $T_0 = 0$
- $T_1 = 1$
- T_n ?
  - Transferir n-1 discos para outro suporte (T_{n-1} movimentos)
  - Deslocar maior disco para suporte final
  - Transferir n-1 discos para o suporte final (T_{n-1} movimentos)

  - Pelo que $T_n \leq 2T_{n-1} + 1$, para $n > 0$
  - Mas,
    - $T_n \geq 2T_{n-1} + 1$, para $n > 0$
      - Temos que realizar T_{n-1} movimentos para poder deslocar o disco maior, mais o movimento do disco maior, mais T_{n-1} adicionais para colocar discos sobre o maior

Torres de Hanoi

- Recorrência:
  - $T_0 = 0$
  - $T_n = 2\,T_{n-1} + 1,\ n > 0$

- Solução:
  - Por substituição: com $T_n = 2^n - 1$

  - Número de movimentos é dado por: $T_n = 2^n - 1$

Exemplos de Recorrências

- Multiplicação de Matrizes: [CLR, Sec. 28.2]
  - Algoritmo usual (3 ciclos): $\Theta(n^3)$
  - Algoritmo de Strassen (recursivo): $\Theta(n^{\lg 7}) = O(n^{2.81})$

Multiplicação de Matrizes

- Algoritmo usual:

MatrixMultiplication(A, B)
  n = número de linhas de A
  for i = 1 **to** n
    for j = 1 **to** n
      $c_{ij} = 0$
      for k = 1 **to** n
        $c_{ij} = c_{ij} + a_{ik} b_{kj}$
  return C

- Tempo de execução: $\Theta(n^3)$

Um Problema — I

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

Um Problema — II

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

Solução — II

- Vector ordenado
- Utilizar dois apontadores colocados inicialmente na primeira (i) e na última (j) posições
  - Se soma das posições apontadas > x
    - decrementar j
  - Se soma das posições apontadas < x
    - incrementar i
  - Se soma das posições apontadas = x
    - terminar

- Complexidade: O(n)

Solução — I

- Basta ordenar vector e resolver problema II
- Complexidade: $\Theta(n \lg n)$

Revisão

- Algoritmos
  - Análise e síntese
- Notação assimptótica & outras notações
- Somatórios & recorrências
- Exemplos

Análise e Síntese de Algoritmos

Revisão

CLRS, Cap. 6-10, 13, 14

Contexto

- Revisão
  - Algoritmos e complexidade
  - Notação
  - Fundamentos: somatórios, recorrências, etc.

  - Exemplos de algoritmos
    - Ordenação
    - Procura
    - Selecção

Resumo

- Estruturas de dados
  - Amontoados (heaps)
- Ordenação
  - HeapSort; MergeSort; QuickSort; CountingSort
- Procura & Selecção
  - Pesquisa linear e pesquisa binária
  - Tópicos sobre selecção

- Estruturas de dados elementares

- Exemplos

Amontoados — Propriedades

- Array de valores interpretado como uma árvore binária (essencialmente completa)
  - length[A]: tamanho do array
  - heap_size[A] : número de elementos na heap
  - Raíz da árvore: A[1]

  - Relações entre nós da árvore:
    - Parent(i) = $\lfloor i / 2 \rfloor$
    - Left(i) = 2 * i
    - Right(i) = 2 * i + 1

- Propriedade do amontoado: A[Parent(i)] $\geq$ A[i]
- Exemplo

Árvore Binária Completa

- Cada nó tem 2 (ou 0) filhos
- Qualquer nó folha (i.e. nó sem descendentes) tem uma mesma profundidade d
  - Profundidade: número de nós entre a raíz e um dado nó
- Todos os nós internos tem grau 2
  - Cada nó interno tem exactamente 2 filhos

Árvore Binária (Essencial/) Completa

- Todos os nós internos têm grau 2, com 1 possível excepção
  - Existe nó à profundidade d-1 com filho esquerdo, mas sem filho direito
- Nós folha posicionados à profundidade d ou d-1
  - Qualquer nó interno à profundidade d-1, posicionado à esquerda de qualquer nó folha à mesma profundidade

Operações sobre Amontoados

- SiftDown
  - A partir da posição i
    - Propriedade do amontoado verificada nas duas sub-árvores com pai i
  - Recursivamente trocar valores entre elementos que não verifiquem propriedade do amontoado

  - Complexidade:
    - Altura da heap: $h = \lfloor \lg n \rfloor$
    - Complexidade de SiftDown: $O(h) = O(\lg n)$

Operações sobre Amontoados (Cont.)

```
SiftDown(A, i)
        l = Left(i)
        r = Right(i)
        if l ≤ heap_size and A[l] > a[i]
                largest = l
        else
                largest = i
        if r ≤ heap_size and A[r] > a[largest]
                largest = r
        if largest ≠ i
                swap(A[i], A[largest])
                SiftDown(A, largest)
```

- Exemplo

Operações sobre Amontoados (Cont.)

- BuildHeap
  - Construir amontoado a partir de um array arbitrário

  - Chamada selectiva de SiftDown
  - Complexidade:
    - O(n lg n)
    - É possível provar O(n)

```
BuildHeap(A)
        heap_size[A] = length[A]
        for i = ⌊length[A] / 2⌋ downto 1
                SiftDown(A, i)
```

Ordenação: O Algoritmo HeapSort

- Ideia:
  - Extrair consecutivamente o elemento máximo de uma heap
  - Colocar esse elemento na posição (certa) do array

- Complexidade:
  - O(n lg n)

```
HeapSort(A)
        BuildHeap(A)
        for i = length[A] downto 2
                swap(A[1], A[i])
                heap_size[A]--
                SiftDown(A, 1)
```

Operações sobre Amontoados

- Max & ExtractMax
  - Complexidade:
    - Max: O(1)
    - ExtractMax: O(lg n)

```
Max(A)
        return A[1]
```

```
ExtractMax(A)
        max = A[1]
        A[1] = A[heap_size[A]]
        heap_size[A]--
        SiftDown(A, 1)
        return max
```

- Min & ExtractMin ?

Operações sobre Amontoados

- Insert & SiftUp
  - Complexidade:
    - SiftUp: O(lg n)
    - Insert: O(lg n)

```
Insert(A, key)
        heap_size[A]++
        i = heap_size[A]
        A[i] = key
        SiftUp(A, i)
```

```
SiftUp(A, i)
        j = Parent(i)
        if j > 0 and A[j] < A[i]
                swap(A[i], A[j])
                SiftUp(A, j)
```

Outras Operações

- IncreaseKey & UpdateKey ??
  - IncreaseKey(i, new_key)
    - Novo valor superior a actual
  - UpdateKey(i, new_key)
    - Novo valor diferente de actual
  - DecreaseKey(i, new_key)
    - Novo valor inferior a actual

Ordenação: O Algoritmo QuickSort

```
QuickSort(A, p, r)
        if p < r
                q = Partition(A, p, r)
                QuickSort(A, p, q)
                QuickSort(A, q+1, r)
```

```
Partition(A, p, r)
        x = A[p]
        i = p - 1
        j = r + 1
        while (TRUE)
                repeat  j = j - 1
                until A[j] ≤ x
                repeat i = i + 1
                until A[i] ≥ x
                if i < j
                        swap(A[i], A[j])
                else
                        return j
```

- Exemplo

Comparação com MergeSort

- QuickSort
  - Array não necessariamente dividido em 2 partes iguais
  - Constantes menores
  - Pior caso (array ordenado): $O(n^2)$
- MergeSort
  - Array dividido em 2 partes *iguais*
  - Necessário fazer Merge
    - Constantes maiores
  - Pior caso: $O(n \lg n)$
- Na prática: QuickSort (aleatorizado) mais rápido

Ordenação: O Algoritmo CountingSort

- Chaves: inteiros de 1 a k
- Contar ocorrências de cada chave, e inserir na região respectiva do array
- Complexidade: O(n + k) = O(n), se k = O(n)

```
CountingSort(A)
        for i = 1 to k
                C[i] = 0
        for j = 1 to length[A]
                C[A[j]]++
        for l = 2 to k
                C[l] = C[l] + C[l-1]
        for j = length[A] downto 1
                B[C[A[j]]] = A[j]
                C[A[j]]--
```

Ordenação: O Algoritmo RadixSort

```
RadixSort(A, d)
        for i = 1 to d
                Ordenar A no dígito i com algoritmo de ordenação estável
```

- Se utilizar CountingSort
  - Complexidade: $O(d(n + k))$
  - Não ordena no lugar, i.e. requer memória adicional...

Procura: Pesquisa Linear

- Procurar elemento em array (ou lista)
  - Array/lista pode não estar ordenada
  - Complexidade: O(n)

```
LinearSearch(A, key)
        for i = 1 to length[A]
                if A[i] = key
                        return i
        return 0
```

Procura: Pesquisa Binária

- Procurar elemento em array
  - Array tem que estar ordenado
  - Complexidade: O(lg n)

```
BinarySearch(A, l, r, key)
        if l <= r
                m = ⌊ (l + r) / 2 ⌋
                if A[m] = key
                        return m
                if A[m] < key
                        return BinarySearch(A, m+1, r, key)
                else if A[m] > key
                        return BinarySearch(A, l, m-1, key)
        return 0
```

Encontrar Máximo e Mínimo

- Comparações para encontrar valor mínimo em array?
  - n-1, valor óptimo
- Comparações para encontrar valor máximo em array?
  - n-1, valor óptimo

- Comparações para encontrar conjuntamente valor máximo e mínimo em array?
  - 2n-2, se selecção é realizada independentemente
  - $3\lceil n / 2 \rceil$, se elementos analisados aos pares, e max e min seleccionados conjuntamente

Estruturas de Dados Elementares

- Amontoados
- Pilhas, Filas
- Listas Ligadas
- Árvores binárias
  - Árvores equilibradas

Um Problema

- Implementar uma fila (FIFO) com uma fila de prioridade??
  - Primeiro elemento inserido é sempre o primeiro retirado
  - Operações: queue(value) & dequeue()

Solução

- Topo do amontoado é o valor mínimo
- Manter contador do número total de elementos inseridos
- Cada elemento inserido no amontoado é caracterizado pelo número de elementos inseridos antes desse elemento
- Push corresponde a insert()
  - O(lg n)
- Pop corresponde a extractMin()
  - O(lg n)

Outro Problema...

- Matriz (nxn)
  - Cada entrada é 0 ou 1
  - Cada linha monoticamente crescente
  - Cada coluna monotonicamente crescente

  - Encontrar algoritmo eficiente para contar número de 0's e de 1's na matriz

Revisão

- Exemplos de Algoritmos & Estruturas de Dados
  - Ordenação
  - Procura & Selecção
  - Estruturas de dados elementares