Introdução: Análise de Complexidade de Algoritmos

Prof. Renê R. Veloso

Notas dos autores: Cormen e Ziviani

Estruturas de dados

- Estruturas de dados e algoritmos estão intimamente ligados:
  - não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas,
  - assim como a escolha dos algoritmos em geral depende da representação e da estrutura dos dados.
- Para resolver um problema é necessário escolher uma abstração da realidade, em geral mediante a definição de um conjunto de dados que representa a situação real.
- A seguir, deve ser escolhida a forma de representar esses dados.

Escolha da Representação dos Dados

- A escolha da representação dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados.
- Considere a operação de adição:
  - Para pequenos números, uma boa representação é por meio de barras verticais (caso em que a operação de adição é
  - Já a representação por dígitos decimais requer regras relativamente complicadas, as quais devem ser memorizadas.
  - Entretanto, quando consideramos a adição de grandes números é mais fácil a representação por dígitos decimais (devido ao princípio baseado no peso relativo da posição de cada dígito).

Programas

- Programar é basicamente estruturar dados e construir algoritmos.
- Programas são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados.
- Programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.
- Um computador só é capaz de seguir programas em linguagem de máquina (sequência de instruções obscuras e desconfortáveis).
- É necessário construir linguagens mais adequadas, que facilitem a tarefa de programar um computador.
- Uma linguagem de programação é uma técnica de notação para programar, com a intenção de servir de veículo tanto para a expressão do raciocínio algorítmico quanto para a execução automática de um algoritmo por um computador.

Tipos de Dados

- Caracteriza o conjunto de valores a que uma constante pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função.
- Tipos simples de dados são grupos de valores indivisíveis (como os tipos básicos *integer, boolean, char* e *real* do Pascal).
  - Exemplo: uma variável do tipo *boolean* pode assumir o valor verdadeiro ou o valor falso, e nenhum outro valor.
- Os tipos estruturados em geral definem uma coleção de valores simples, ou um agregado de valores de tipos diferentes.

Tipos Abstratos de Dados (TAD)

- Modelo matemático, acompanhado das operações definidas sobre o modelo.
  - Exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação.
- TADs são utilizados como base para o projeto de algoritmos.
- A implementação do algoritmo em uma linguagem de programação exige a representação do TAD em termos dos tipos de dados e dos operadores suportados.
- A representação do modelo matemático por trás do tipo abstrato de dados é realizada mediante uma estrutura de dados.
- Podemos considerar TADs como generalizações de tipos primitivos e procedimentos como generalizações de operações primitivas.
- O TAD encapsula tipos de dados. A definição do tipo e todas as operações ficam localizadas numa seção do programa.

Implementação de TADs (1)

- Considere uma uma lista de inteiros. Poderíamos definir TAD Lista, com as seguintes operações:
  1. faça a lista vazia;
  2. obtenha o primeiro elemento da lista; se a lista estiver vazia, então retorne nulo;
  3. insira um elemento na lista.
- Há várias opções de estruturas de dados que permitem uma implementação eficiente para listas (por ex., o tipo estruturado arranjo).

Implementação de TADs (2)

- Cada operação do tipo abstrato de dados é implementada como um procedimento na linguagem de programação escolhida.
- Qualquer alteração na implementação do TAD fica restrita à parte encapsulada, sem causar impactos em outras partes do código.
- Cada conjunto diferente de operações define um TAD diferente, mesmo que atuem sob um mesmo modelo matemático.
- A escolha adequada de uma implementação depende fortemente das operações a serem realizadas sobre o modelo.

Medida do Tempo de Execução de um Programa

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos.
- Depois que um problema é analisado e decisões de projeto são finalizadas, é necessário estudar as várias opções de algoritmos a serem utilizados, considerando os aspectos de tempo de execução e espaço ocupado.
- Muitos desses algoritmos são encontrados em áreas como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras.

Tipos de Problemas na Análise de Algoritmos

- **Análise de um algoritmo particular**.
  - Qual é o custo de usar um dado algoritmo para resolver um problema específico?
  - Características que devem ser investigadas:
    - análise do número de vezes que cada parte do algoritmo deve ser executada,
    - estudo da quantidade de memória necessária.
- **Análise de uma classe de algoritmos**.
  - Qual é o algoritmo de menor custo possível para resolver um problema particular?
  - Toda uma família de algoritmos é investigada.
  - Procura-se identificar um que seja o melhor possível.
  - Coloca-se **limites** para a complexidade computacional dos algoritmos pertencentes à classe.

Custo de um Algoritmo

- Determinando o menor custo possível para resolver problemas de uma classe, temos a medida da dificuldade inerente para resolver o problema.
- Quando o custo de um algoritmo é igual ao menor custo possível, o algoritmo é **ótimo** para a medida de custo considerada.
- Podem existir vários algoritmos para resolver o mesmo problema.
- Se a mesma medida de custo é aplicada a diferentes algoritmos, então é possível compará-los e escolher o mais adequado.

Medida do Custo pela Execução do Programa

- Tais medidas são inadequadas e os resultados jamais devem ser generalizados:
  - os resultados são dependentes do compilador que pode favorecer algumas construções em detrimento de outras;
  - os resultados dependem do *hardware*;
  - quando grandes quantidades de memória são utilizadas, as medidas de tempo podem depender desse aspecto.
- Apesar disso, há argumentos a favor de medidas reais de tempo.
  - Ex.: quando há vários algoritmos para resolver um mesmo tipo de problema, todos com um custo de execução dentro da mesma ordem de grandeza.
  - Assim, são considerados tanto os custos reais das operações como os custos não aparentes, tais como alocação de memória, indexação, carga, dentre outros.

Medida do Custo por meio de um Modelo Matemático

- Usa um modelo matemático baseado em um computador idealizado.
- Deve ser especificado o conjunto de operações e seus custos de execuções.
- É mais usual ignorar o custo de algumas das operações e considerar apenas as operações mais significativas.
- Ex.: algoritmos de ordenação. Consideramos o número de comparações entre os elementos do conjunto e ignoramos operações aritméticas, de atribuição e manipulações de índices, entre outras.

Função de Complexidade

- Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade** f.
- $f(n)$ é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n.
- Função de **complexidade de tempo**: $f(n)$ mede o tempo necessário para executar um algoritmo em um problema de tamanho n.
- Função de **complexidade de espaço**: $f(n)$ mede a memória necessária para executar algoritmo em um problema de tamanho n.
- Utilizaremos f para denotar uma função de complexidade de tempo daqui para a frente.
- A complexidade de tempo na realidade não representa tempo diretamente, mas o número de vezes que determinada operação considerada relevante é executada.

Tamanho da Entrada de Dados

- A medida do custo de execução de um algoritmo depende principalmente do tamanho da entrada dos dados.
- É comum considerar o tempo de execução de um programa como uma função do tamanho da entrada.
- Para alguns algoritmos, o custo de execução é uma função da entrada particular dos dados, não apenas do tamanho da entrada.

Melhor Caso, Pior Caso e Caso Médio (1)

- **Melhor caso**: menor tempo de execução sobre todas as entradas de tamanho n.
- **Pior caso**: maior tempo de execução sobre todas as entradas de tamanho n.
- Se f é uma função de complexidade baseada na análise de pior caso, o custo de aplicar o algoritmo nunca é maior do que $f(n)$.
- **Caso médio** (ou caso esperado): média dos tempos de execução de todas as entradas de tamanho n.

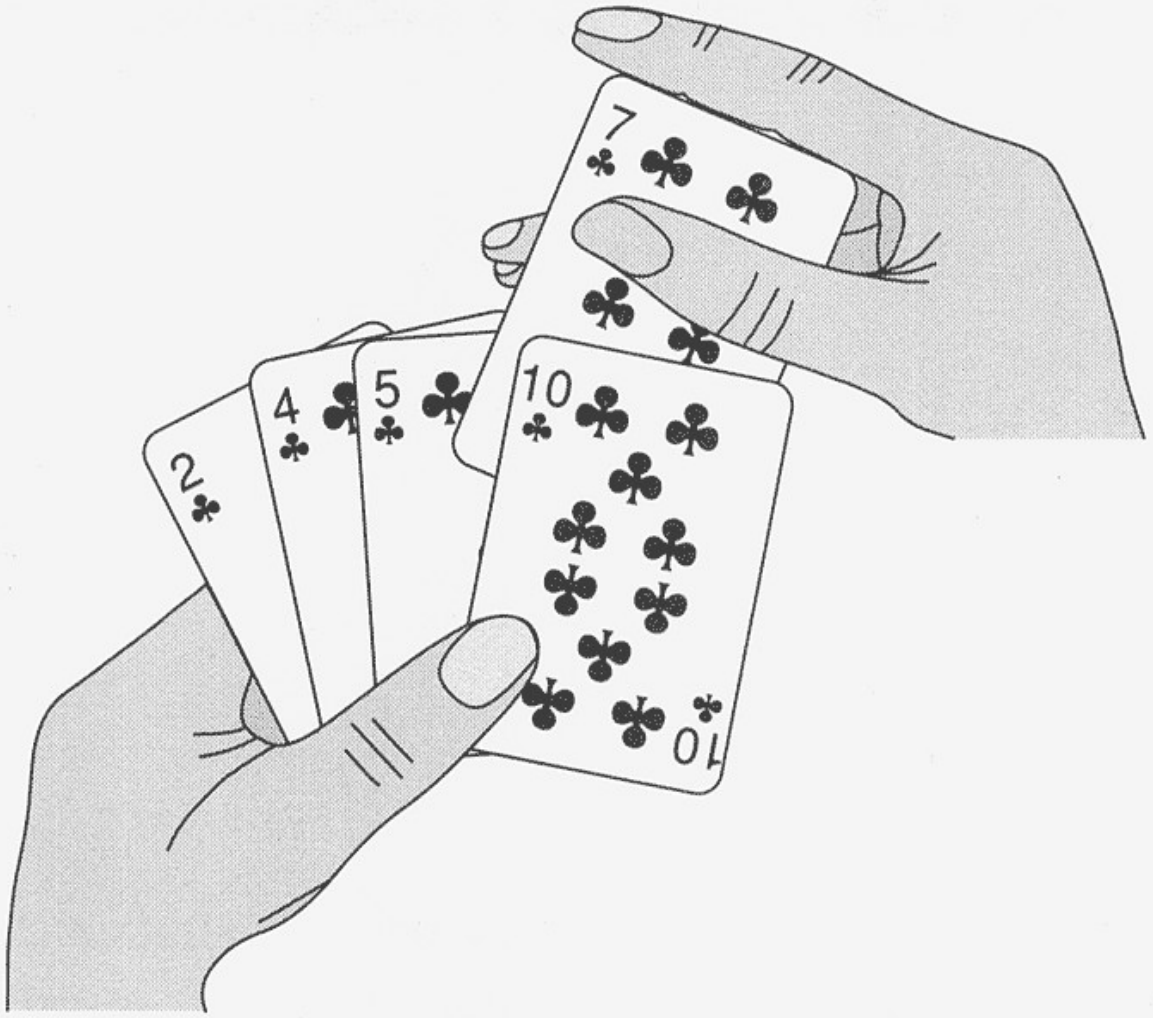

Figure 2.1 Sorting a hand of cards using insertion sort.

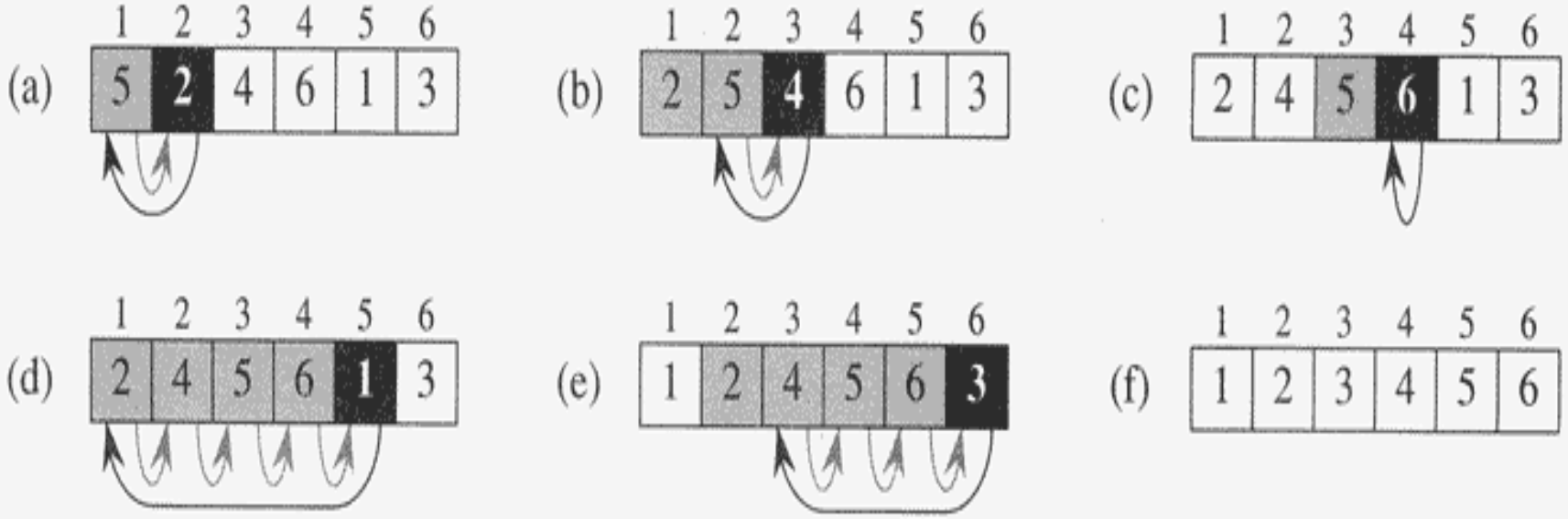

Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** $key \leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow key$	c_8	$n-1$

$$f(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n} t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

Pior caso: Sequência em ordem inversa à desejada. Neste caso, t_j=j, ou seja, todas as comparações sempre.

$$f(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5[\frac{n(n+1)}{2}-1]+c_6 c_7[\frac{n(n+1)}{2}-n]+c_8(n-1)$$

$$f(n)=(\frac{c_5+c_6+c_7}{2})n^2+(c_1+c_2+c_4+\frac{c_5-c_6-c_7}{2}+c_8)n-(c_2+c_4+c_5+c_8)$$

$$f(n)=an^2+bn+c$$

Tempo de execução quadrático!

$$f(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5\sum_{j=2}^{n} t_j+c_6\sum_{j=2}^{n}(t_j-1)+c_7\sum_{j=2}^{n}(t_j-1)+c_8(n-1)$$

Melhor caso: Sequência ordenada na ordem desejada.
Assim, $t_j=1$.

$$\sum_{j=2}^{n} t_j=\sum_{j=2}^{n} 1=n-1 \qquad \sum_{j=2}^{n}(t_j-1)=0$$

$$f(n)=c_1 n+c_2(n-1)+c_4(n-1)+c_5(n-1)+c_8(n-1)$$

$$f(n)=(c_1+c_2+c_4+c_5+c_8)n-(c_2+c_4+c_5+c_8)$$

$$f(n)=an+b$$

Tempo de execução linear!

Exemplo - Maior Elemento

- Considere o algoritmo para encontrar o maior elemento de um vetor de inteiros $A[1..n], n \geq 1$.

```
int Max(TipoVetor A)
{ int i, Temp;
  Temp = A[0];
  for (i = 1; i < N; i++) if (Temp < A[i]) Temp = A[i];
  return Temp;
}
```

- Seja f uma função de complexidade tal que $f(n)$ é o número de comparações entre os elementos de A, se A contiver n elementos.

Exemplo - Maior e Menor Elemento (1)

- Encontrar o maior e o menor elemento de $A[1..n], n \geq 1$.
- Um algoritmo simples pode ser derivado do algoritmo para achar o maior elemento.

```
void MaxMin1(TipoVetor A, int *Max, int *Min)
{ int i; *Max = A[0]; *Min = A[0];
  for (i = 1; i < N; i++)
    { if (A[i] > *Max) *Max = A[i];
      if (A[i] < *Min) *Min = A[i];
    }
}
```

- Seja $f(n)$ o número de comparações entre os n elementos de As.

Exemplo - Maior e Menor Elemento (2)

```
void MaxMin2(TipoVetor A, int *Max, int *Min)
{ int i; *Max = A[0]; *Min = A[0];
  for (i = 1; i < N; i++)
    { if (A[i] > *Max) *Max = A[i];
      else if (A[i] < *Min) *Min = A[i];
    }
}
```

- MaxMin1 pode ser facilmente melhorado: a comparação $A[i] <$ Min só é necessária quando a comparação $A[i] >$ Max dá falso.

É possível melhorar?