MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2015

Roteiro

O problema da ordenação

- Vamos estudar alguns algoritmos para o seguinte problema:

> Dada uma coleção de elementos, com uma relação de ordem entre eles, ordenar os elementos da coleção de forma crescente.

- Nos nossos exemplos, a coleção de elementos será representada por um vetor de inteiros.
    - Números inteiros possuem uma relação de ordem entre eles.
- Apesar de usarmos inteiros, os algoritmos que estudaremos servem para ordenar qualquer coleção de elementos que possam ser comparados entre si.

O problema da ordenação

- O problema da ordenação é um dos mais básicos em computação.
- Muito provavelmente este é um dos problemas com maior número de aplicações diretas ou indiretas (como parte da solução para um problema maior).
- Exemplos de aplicações diretas:
  - criação de *rankings*.
  - definição de preferências em atendimentos por prioridade.
  - criação de listas.
- Exemplos de aplicações indiretas:
  - otimização de sistemas de busca.
  - manutenção de estruturas de bancos de dados.

Selection Sort

- Seja `vet` um vetor, contendo `n` números inteiros, que desejamos ordenar de forma crescente.
- A ideia do algoritmo é a seguinte:
  - Encontre o menor elemento a partir da posição 0. Troque este elemento com o elemento da posição 0.
  - Encontre o menor elemento a partir da posição 1. Troque este elemento com o elemento da posição 1.
  - Encontre o menor elemento a partir da posição 2. Troque este elemento com o elemento da posição 2.
  - E assim sucessivamente...

Selection Sort

- No exemplo abaixo, os elementos sublinhados representam os elementos que serão trocados na `i`-ésima iteração do Selection Sort:

  Iteração 0: $(\underline{57}, 32, 25, \underline{11}, 90, 63)$
  Iteração 1: $(11, \underline{32}, \underline{25}, 57, 90, 63)$
  Iteração 2: $(11, 25, \underline{32}, 57, 90, 63)$
  Iteração 3: $(11, 25, 32, \underline{57}, 90, 63)$
  Iteração 4: $(11, 25, 32, 57, \underline{90}, \underline{63})$
  Iteração 5: $(11, 25, 32, 57, 63, \underline{90})$

Selection Sort

- Podemos criar uma função que retorna o índice do menor elemento de um vetor `vet` (formado por `n` números inteiros) a partir de uma posição inicial dada:

```
int indiceMenor(int vet[], int n, int inicio) {
  int j, min = inicio;

  for (j = inicio + 1; j < n; j++)
    if (vet[min] > vet[j])
      min = j;

  return min;
}
```

Selection Sort

- Dada a função anterior, que encontra o índice do menor elemento de um vetor a partir de uma dada posição, como implementar o algoritmo de ordenação?
  - Encontre o menor elemento a partir da posição 0 e troque-o com o elemento da posição 0.
  - Encontre o menor elemento a partir da posição 1 e troque-o com o elemento da posição 1.
  - Encontre o menor elemento a partir da posição 2 e troque-o com o elemento da posição 2.
  - E assim sucessivamente...

Selection Sort

- Como vimos anteriormente, podemos criar uma função que troca os valores armazenados em duas variáveis inteiras:

```
void troca(int *a, int *b) {
  int aux;

  aux = *a;
  *a = *b;
  *b = aux;
}
```

Selection Sort

- Usando as funções auxiliares `indiceMenor` e `troca` podemos implementar o Selection Sort da seguinte forma:

```
void selectionSort(int vet[], int n) {
  int i, min;

  for (i = 0; i < n; i++) {
    min = indiceMenor(vet, n, i);
    troca(&vet[i], &vet[min]);
  }
}
```

Selection Sort

- Usando as funções auxiliares `indiceMenor` e `troca` podemos implementar o Selection Sort da seguinte forma:

```
void selectionSort(int vet[], int n) {
  int i, min;

  for (i = 0; i < n - 1; i++) {
    min = indiceMenor(vet, n, i);
    troca(&vet[i], &vet[min]);
  }
}
```

- Note que o laço principal da função não precisa ir até o último elemento do vetor.

Selection Sort

```
#include <stdio.h>

void selectionSort(int vet[], int n);
int indiceMenor(int vet[], int n, int inicio);
void troca(int *a, int *b);

int main() {
  int i, vetor[10] = {14, 7, 8, 34, 56, 4, 0, 9, -8, 100};

  selectionSort(vetor, 10);

  printf("Vetor Ordenado:\n");
  for (i = 0; i < 10; i++)
    printf("%d\n", vetor[i]);

  return 0;
}
```

Selection Sort - Análise de complexidade (pior caso)

```
void selectionSort(int vet[], int n) {
  int i, min;

  for (i = 0; i < n - 1; i++) {
    min = indiceMenor(vet, n, i);
    troca(&vet[i], &vet[min]);
  }
}
```

- Número máximo de comparações entre elementos do vetor:

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

Selection Sort - Análise de complexidade (pior caso)

```
void selectionSort(int vet[], int n) {
  int i, min;

  for (i = 0; i < n - 1; i++) {
    min = indiceMenor(vet, n, i);
    troca(&vet[i], &vet[min]);
  }
}
```

- Número máximo de trocas entre elementos do vetor:

$$f(n) = \sum_{i=0}^{n-2} 1 = n - 1$$

Selection Sort - Análise de complexidade (melhor caso)

```
void selectionSort(int vet[], int n) {
  int i, min;

  for (i = 0; i < n - 1; i++) {
    min = indiceMenor(vet, n, i);
    troca(&vet[i], &vet[min]);
  }
}
```

- Número mínimo de comparações entre elementos do vetor:

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

Selection Sort - Análise de complexidade (melhor caso)

```
void selectionSort(int vet[], int n) {
  int i, min;

  for (i = 0; i < n - 1; i++) {
    min = indiceMenor(vet, n, i);
    troca(&vet[i], &vet[min]);
  }
}
```

- Número mínimo de trocas entre elementos do vetor:

$$f(n) = \sum_{i=0}^{n-2} 1 = n - 1$$

Selection Sort

- É possível diminuir o número de trocas no melhor caso?
- Vale a pena testar se `vet[i]` $\neq$ `vet[min]` antes de realizar a troca?

Insertion Sort

- Seja `vet` um vetor, contendo `n` números inteiros, que desejamos ordenar de forma crescente.
- A ideia do algoritmo é a seguinte:
  - A cada iteração `i`, os elementos das posições 0 até `i-1` do vetor estão ordenados.
  - Então, precisamos inserir o elemento da posição `i`, entre as posições 0 e `i`, de forma a deixar o vetor ordenado até a posição `i`.
  - Na iteração seguinte, consideramos que o vetor está ordenado até a posição `i` e repetimos o processo até que o vetor esteja completamente ordenado.

Insertion Sort

- No exemplo abaixo, o elemento sublinhado representa o elemento que será inserido na `i`-ésima iteração do Insertion Sort:

  $(57, \underline{25}, 32, 11, 90, 63)$: vetor ordenado entre as posições 0 e 0.
  $(25, 57, \underline{32}, 11, 90, 63)$: vetor ordenado entre as posições 0 e 1.
  $(25, 32, 57, \underline{11}, 90, 63)$: vetor ordenado entre as posições 0 e 2.
  $(11, 25, 32, 57, \underline{90}, 63)$: vetor ordenado entre as posições 0 e 3.
  $(11, 25, 32, 57, 90, \underline{63})$: vetor ordenado entre as posições 0 e 4.
  $(11, 25, 32, 57, 63, 90)$: vetor ordenado entre as posições 0 e 5.

Insertion Sort

- Podemos criar uma função que, dados um vetor e um índice `i`, insere o elemento de índice `i` entre os elementos das posições `0` e `i-1` (ordenados), de forma que todos os elementos entre as posições `0` e `i` fiquem ordenados:

```
void insertion(int vet[], int i) {
  int j, aux = vet[i];

  for (j = i - 1; (j >= 0) && (vet[j] > aux); j--)
    vet[j + 1] = vet[j];

  vet[j + 1] = aux;
}
```

Insertion Sort

- Exemplo de execução da função `insertion`:
  - Configuração inicial:

    $(11, 31, 54, 58, 66, \underline{12}, 47)$, `i = 5`, `aux = 12`
  - Iterações:

    $(11, 31, 54, 58, \underline{66}, 12, 47)$, `j = 4`
    $(11, 31, 54, \underline{58}, 66, 66, 47)$, `j = 3`
    $(11, 31, \underline{54}, 58, 58, 66, 47)$, `j = 2`
    $(11, \underline{31}, 54, 54, 58, 66, 47)$, `j = 1`
    $(\underline{11}, 31, 31, 54, 58, 66, 47)$, `j = 0`
  - Aqui temos que `vet[j] < aux`, logo, fazemos `vet[j + 1] = aux`:

    $(11, \underline{12}, 31, 54, 58, 66, 47)$

Insertion Sort - Análise de complexidade (pior caso)

```
void insertionSort(int vet[], int n) {
  int i;

  for (i = 1; i < n; i++)
    insertion(vet, i);
}
```

- Número máximo de comparações entre elementos do vetor:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

Insertion Sort - Análise de complexidade (pior caso)

```
void insertionSort(int vet[], int n) {
  int i;

  for (i = 1; i < n; i++)
    insertion(vet, i);
}
```

- Número máximo de modificações realizadas no vetor:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i} 1 = \sum_{i=1}^{n-1} (i+1) = (n-1)\frac{n+2}{2} = \frac{n^2+n}{2} - 1$$

Insertion Sort - Análise de complexidade (melhor caso)

```
void insertionSort(int vet[], int n) {
  int i;

  for (i = 1; i < n; i++)
    insertion(vet, i);
}
```

- Número mínimo de comparações entre elementos do vetor:

$$f(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Insertion Sort - Análise de complexidade (melhor caso)

```
void insertionSort(int vet[], int n) {
  int i;

  for (i = 1; i < n; i++)
    insertion(vet, i);
}
```

- Número mínimo de modificações realizadas no vetor:

$$f(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Bubble Sort

- Seja `vet` um vetor, contendo `n` números inteiros, que desejamos ordenar de forma crescente.
- O algoritmo faz iterações repetindo os seguintes passos:
  - Se `vet[0] > vet[1]`, troque `vet[0]` com `vet[1]`.
  - Se `vet[1] > vet[2]`, troque `vet[1]` com `vet[2]`.
  - Se `vet[2] > vet[3]`, troque `vet[2]` com `vet[3]`.
  - ...
  - Se `vet[n-2] > vet[n-1]`, troque `vet[n-2]` com `vet[n-1]`.
- Após uma iteração executando os passos acima, o que podemos garantir?
  - O maior elemento estará na posição correta.

Bubble Sort

- Após a primeira iteração de trocas, o maior elemento estará na última posição.
- Após a segunda iteração de trocas, o segundo maior elemento estará na posição correta.
- E assim sucessivamente...
- Quantas iterações são necessárias para deixar o vetor completamente ordenado?

Bubble Sort

- No exemplo abaixo, os elementos sublinhados estão sendo comparados (e, eventualmente, serão trocados):

  $(\underline{57}, \underline{32}, 25, 11, 90, 63)$
  $(32, \underline{57}, \underline{25}, 11, 90, 63)$
  $(32, 25, \underline{57}, \underline{11}, 90, 63)$
  $(32, 25, 11, \underline{57}, \underline{90}, 63)$
  $(32, 25, 11, 57, \underline{90}, \underline{63})$
  $(32, 25, 11, 57, 63, 90)$

- Isto termina a primeira iteração de trocas.
- Como o vetor possui 6 elementos, temos que realizar 5 iterações.
- Note que, após a primeira iteração, não precisamos mais avaliar a última posição do vetor.

Bubble Sort

- O código abaixo realiza as trocas de uma iteração do algoritmo.
- Os pares de elementos das posições 0 e 1, 1 e 2, ..., `i-1` e `i` são comparados e, eventualmente, trocados.
- Assumimos que, das posições `i+1` até `n-1`, o vetor já possui os maiores elementos ordenados.

```
for (j = 0; j < i; j++)
  if (vet[j] > vet[j + 1])
    troca(&vet[j], &vet[j + 1]);
```

Bubble Sort - Análise de complexidade (pior caso)

```
void bubbleSort(int vet[], int n) {
   int i, j;

   for (i = n - 1; i > 0; i--)
     for (j = 0; j < i; j++)
       if (vet[j] > vet[j + 1])
         troca(&vet[j], &vet[j + 1]);
}
```

- Note que as comparações na primeira iteração ocorrem até a última posição do vetor.
- Na segunda iteração, elas ocorrem até a penúltima posição.
- E assim sucessivamente...

Bubble Sort - Análise de complexidade (pior caso)

```
void bubbleSort(int vet[], int n) {
   int i, j;

   for (i = n - 1; i > 0; i--)
     for (j = 0; j < i; j++)
       if (vet[j] > vet[j + 1])
         troca(&vet[j], &vet[j + 1]);
}
```

- Número máximo de comparações entre elementos do vetor:

$$f(n) = \sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

Bubble Sort - Análise de complexidade (melhor caso)

```
void bubbleSort(int vet[], int n) {
   int i, j;

   for (i = n - 1; i > 0; i--)
     for (j = 0; j < i; j++)
       if (vet[j] > vet[j + 1])
         troca(&vet[j], &vet[j + 1]);
}
```

- Número máximo de trocas entre elementos do vetor.

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

Bubble Sort - Análise de complexidade (melhor caso)

```
void bubbleSort(int vet[], int n) {
   int i, j;

   for (i = n - 1; i > 0; i--)
     for (j = 0; j < i; j++)
       if (vet[j] > vet[j + 1])
         troca(&vet[j], &vet[j + 1]);
}
```

- Número mínimo de comparações entre elementos do vetor:

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = (n-1)\frac{n}{2} = \frac{n^2 - n}{2}$$

Bubble Sort

```
void bubbleSort(int vet[], int n) {
   int i, j;

   for (i = n - 1; i > 0; i--)
     for (j = 0; j < i; j++)
       if (vet[j] > vet[j + 1])
         troca(&vet[j], &vet[j + 1]);
}
```

- Número mínimo de trocas entre elementos do vetor.

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 0 = 0$$

Resumo

- Não existe um algoritmo de ordenação que seja o melhor em todas as possíveis situações.
- Para escolher o algoritmo mais adequado para uma dada situação precisamos verificar as características específicas dos elementos que devem ser ordenados.
- Por exemplo:
  - Se os elementos a serem ordenados forem grandes, por exemplo, registros acadêmicos de alunos, o Selection Sort pode ser uma boa escolha, já que ele efetuará, no pior caso, muito menos trocas que o Insertion Sort ou o Bubble Sort.
  - Se os elementos a serem ordenados estiverem quase ordenados (situação relativamente comum), o Insertion Sort realizará muito menos operações (comparações e trocas) do que o Selection Sort ou o Bubble Sort.

Exercícios

- Altere o Bubble Sort para que o algoritmo pare assim que for possível perceber que o vetor estiver ordenado. Qual o custo deste novo algoritmo em termos do número de comparações entre elementos do vetor (tanto no melhor, quanto no pior caso)?
- Escreva uma função `k-ésimo` que, dado um vetor de tamanho n e um inteiro k (tal que $1 \leq k \leq n$), determine o `k-ésimo` maior elemento do vetor. Analise o custo da sua função em termos do número de comparações realizadas entre elementos do vetor.