MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2013

Roteiro

Ordenação

- Vamos estudar alguns algoritmos para o seguinte problema:

> Dada uma coleção de elementos, com uma relação de ordem entre si, gerar uma saída com os elementos ordenados.

- Nos nossos exemplos, usaremos um vetor de inteiros como a coleção de elementos.
  - É claro que quaisquer inteiros possuem uma relação de ordem entre si.
- Apesar de usarmos inteiros, os algoritmos servem para ordenar qualquer coleção de elementos que possam ser comparados.

Ordenação

- O problema de ordenação é um dos mais básicos em computação.
  - Muito provavelmente 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 números inteiros.
- Devemos ordenar os elementos de `vet` crescentemente.
- A ideia do algoritmo é a seguinte:
  - Ache o menor elemento a partir da posição 0. Troque então este elemento com o elemento da posição 0.
  - Ache o menor elemento a partir da posição 1. Troque então este elemento com o elemento da posição 1.
  - Ache o menor elemento a partir da posição 2. Troque então este elemento com o elemento da posição 2.
  - E assim sucessivamente...

Selection Sort

Exemplo: $(5, 3, 2, 1, 90, 6)$

Após a iteração 0: $(\underline{1}, 3, 2, \underline{5}, 90, 6)$.
Após a iteração 1: $(1, \underline{2}, \underline{3}, 5, 90, 6)$.
Após a iteração 2: $(1, 2, \underline{3}, 5, 90, 6)$.
Após a iteração 3: $(1, 2, 3, \underline{5}, 90, 6)$.
Após a iteração 4: $(1, 2, 3, 5, \underline{6}, \underline{90})$.

Selection Sort

- Como encontrar o índice do menor elemento a partir de uma posição inicial do vetor de tamanho n?

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

Selection Sort

Criamos então uma função que retorna o índice do menor elemento de um vetor:

```
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 para achar o índice do menor elemento, como implementar o algoritmo de ordenação?
- Ache o menor elemento a partir da posição 0 e troque com o elemento da posição 0.
- Ache o menor elemento a partir da posição 1 e troque com o elemento da posição 1.
- Ache o menor elemento a partir da posição 2 e troque com o elemento da posição 2.
- E assim sucessivamente...

Selection Sort

Criamos então uma função que troca dois valores inteiros.

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

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

Selection Sort

```
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

```
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 não precisa ir até o último elemento do vetor.

Selection Sort

```
#include <stdio.h>

int main() {
  int i, vetor[10] = {14, 7, 8, 34, 56, 4, 0, 9, -8, 100};
  printf("Vetor Antes:\n");
  for (i = 0; i < 10; i++)
    printf("%d ", vetor[i]);
  printf("\n");

  selectionSort(vetor, 10);

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

  return 0;
}
```

Selection Sort

```
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]);
  }
}
```

Análise de custo (pior caso): comparações entre elementos do vetor.

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

Selection Sort

```
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]);
  }
}
```

Análise de custo (pior caso): trocas entre elementos do vetor.

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

Selection Sort

```
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]);
  }
}
```

Análise de custo (melhor caso): comparações entre elementos do vetor.

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

Selection Sort

```
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]);
  }
}
```

Análise de custo (melhor caso): trocas entre elementos do vetor.

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

Selection Sort

- É possível melhor 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 números inteiros, que devemos deixar ordenado.
- A ideia do algoritmo é a seguinte:
  - A cada passo, uma porção de 0 até $i-1$ do vetor já está ordenada.
  - Devemos inserir o item da posição i na posição correta para deixar o vetor ordenado até a posição i.
  - No passo seguinte, consideramos que o vetor está ordenado até i.

Insertion Sort

Exemplo: $(5, 3, 2, 1, 90, 6)$

O elemento sublinhado representa onde está o índice i.
$(5, \underline{3}, 2, 1, 90, 6)$: vetor ordenado entre as posições 0 e 0.
$(3, 5, \underline{2}, 1, 90, 6)$: vetor ordenado entre as posições 0 e 1.
$(2, 3, 5, \underline{1}, 90, 6)$: vetor ordenado entre as posições 0 e 2.
$(1, 2, 3, 5, \underline{90}, 6)$: vetor ordenado entre as posições 0 e 3.
$(1, 2, 3, 5, 90, \underline{6})$: vetor ordenado entre as posições 0 e 4.
$(1, 2, 3, 5, 6, 90)$: vetor ordenado entre as posições 0 e 5.

Insertion Sort

- Vamos supor que o vetor está ordenado de 0 até $i - 1$.
- Vamos inserir o elemento da posição i no lugar correto.

```
/* guardar o elemento que deseja-se inserir */
aux = vet[i];
/* analisar elementos das primeiras j posições */
j = i - 1;

/* enquanto vet[j] > aux */
while ((j >= 0) && (vet[j] > aux)) {
  /* copia o elemento da posição j para a posição j+1 */
  vet[j+1] = vet[j];
  j--;
}

/* insere o elemento na posição correta */
vet[j+1] = aux;
```

Insertion Sort

Exemplo: $(1, 3, 5, 10, 20, 2, 4)$ com $i = 5$

$(1, 3, 5, 10, \underline{20}, 2, 4)$: $aux = 2$, $j = 4$.
$(1, 3, 5, \underline{10}, 20, 20, 4)$: $aux = 2$, $j = 3$.
$(1, 3, \underline{5}, 10, 10, 20, 4)$: $aux = 2$, $j = 2$.
$(1, \underline{3}, 5, 5, 10, 20, 4)$: $aux = 2$, $j = 1$.
$(\underline{1}, 3, 3, 5, 10, 20, 4)$: $aux = 2$, $j = 0$.

Aqui temos que $vet[j] < aux$, logo, fazemos `vet[j+1] = aux`.

$(1, \underline{2}, 3, 5, 10, 20, 4)$: $aux = 2$, $j = 0$.

Insertion Sort

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

  for (i = 1; i < n; i++) {
    aux = vet[i];
    j = i - 1;
    while ((j >= 0) && (vet[j] > aux)) {
      vet[j+1] = vet[j];
      j--;
    }
    vet[j+1] = aux;
  }
}
```

Análise de custo (pior caso): comparações entre elementos do vetor.

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

Insertion Sort

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

  for (i = 1; i < n; i++) {
    aux = vet[i];
    j = i - 1;
    while ((j >= 0) && (vet[j] > aux)) {
      vet[j+1] = vet[j];
      j--;
    }
    vet[j+1] = aux;
  }
}
```

Análise de custo (pior caso): modificações realizadas no vetor.

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

Insertion Sort

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

  for (i = 1; i < n; i++) {
    aux = vet[i];
    j = i - 1;
    while ((j >= 0) && (vet[j] > aux)) {
      vet[j+1] = vet[j];
      j--;
    }
    vet[j+1] = aux;
  }
}
```

Análise de custo (melhor caso): comparações entre elementos do vetor.

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

Insertion Sort

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

  for (i = 1; i < n; i++) {
    aux = vet[i];
    j = i - 1;
    while ((j >= 0) && (vet[j] > aux)) {
      vet[j+1] = vet[j];
      j--;
    }
    vet[j+1] = aux;
  }
}
```

Análise de custo (melhor caso): modificações realizadas no vetor.

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

Bubble Sort

- Seja `vet` um vetor contendo números inteiros.
- Ordenar os elementos de `vet` crescentemente.
- O algoritmo faz algumas iterações repetindo o seguinte:
  - Compare $vet[0]$ com $vet[1]$ e troque-os se $vet[0] > vet[1]$.
  - Compare $vet[1]$ com $vet[2]$ e troque-os se $vet[1] > vet[2]$.
  - .....
  - Compare $vet[n-2]$ com $vet[n-1]$ e troque-os se $vet[n-2] > vet[n-1]$.

Após uma iteração repetindo esses passos, o que podemos garantir?

  - O maior elemento estará na posição correta.

Bubble Sort

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

Bubble Sort

Exemplo: $(5, 3, 2, 1, 90, 6)$

Elementos sublinhados estão sendo comparados:
$(\underline{5}, \underline{3}, 2, 1, 90, 6)$
$(3, \underline{5}, \underline{2}, 1, 90, 6)$
$(3, 2, \underline{5}, \underline{1}, 90, 6)$
$(3, 2, 1, \underline{5}, \underline{90}, 6)$
$(3, 2, 1, 5, \underline{90}, \underline{6})$
$(3, 2, 1, 5, 6, 90)$

- Isto termina a primeira iteração de trocas. Temos que repetir todo o processo mais 4 vezes.
- Note que não precisamos mais avaliar a última posição.

Bubble Sort

- O código abaixo realiza as trocas de uma iteração.
- São comparados e trocados os elementos das posições: 0 e 1, 1 e 2, ..., $i-1$ e i.
- Assumimos que, de $(i+1)$ até $(n-1)$, o vetor já tem os maiores elementos ordenados.

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

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]);
}
```

Bubble Sort

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

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]);
}
```

Análise de custo (pior caso): comparações entre elementos do vetor.

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

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]);
}
```

Análise de custo (pior caso): trocas entre elementos do vetor.

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

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]);
}
```

Análise de custo (melhor caso): comparações entre elementos do vetor.

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

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]);
}
```

Análise de custo (melhor caso): trocas entre elementos do vetor.

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

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 termo do número de comparações entre elementos do vetor (tanto no melhor, quanto no pior caso)?
- Escreva o pseudocódigo do algortimo `k-esimo` que, dado um vetor de tamanho de tamanho n e um inteiro k (tal que $1 \leq k \leq n$), determine o k-ésimo maior elemento do vetor. Escreva uma função em C que implemente seu algoritmo e analise o custo do seu algoritmo em termo do número de comparações realizadas entre elementos do vetor.