MC102 – Algoritmos e Programação de Computadores

Instituto de Computação

UNICAMP

Primeiro Semestre de 2014

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 da 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

Vetor inicial: $(57, 32, 25, 11, 90, 63)$

Os elementos sublinhados representam os elementos que serão trocados.

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 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 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\n", vetor[i]);

  selectionSort(vetor, 10);

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

  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 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 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, entre as posições 0 e i, de forma a deixar o vetor ordenado até a posição i.
  - No passo seguinte, consideramos que o vetor está ordenado até i.

Insertion Sort

Exemplo: $(57, 25, 32, 11, 90, 63)$

O elemento sublinhado representa onde está o índice i.

$(57, \underline{25}, 32, 11, 90, 63)$: 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: $(11, 31, 54, 58, 66, \underline{12}, 47)$, com $i = 5$ e $aux = 12$.

$(11, 31, 54, 58, \underline{66}, 12, 47), \quad j = 4$
$(11, 31, 54, \underline{58}, 66, 66, 47), \quad j = 3$
$(11, 31, \underline{54}, 58, 58, 66, 47), \quad j = 2$
$(11, \underline{31}, 54, 54, 58, 66, 47), \quad j = 1$
$(\underline{11}, 31, 31, 54, 58, 66, 47), \quad j = 0$

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

$(11, \underline{12}, 31, 54, 58, 66, 47), \quad j = 0$

Insertion Sort

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

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

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;

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

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;

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

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;

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

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 os seguintes passos:
  - 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 executando os passos a cima, 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: $(57, 35, 25, 11, 90, 63)$

Elementos sublinhados estão sendo comparados:

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

- Isto termina a primeira iteração de trocas. 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 termos do número de comparações entre elementos do vetor (tanto no melhor, quanto no pior caso)?
- Escreva uma função `k-esimo` 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.