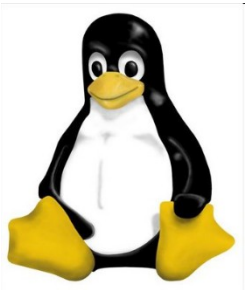
INSTITUTO FEDERAL
BAIANO



Estrutura de Dados

Métodos de Ordenação

Prof. José Honorato Ferreira Nunes
honoratonunes@softwarelivre.org
http://softwarelivre.org/zenorato

Algoritmos de Ordenação

São algoritmos que colocam os elementos de uma dada sequência em uma certa ordem (ascendente/descendente).

As ordens mais usadas são a **numérica** e a **lexicográfica** (quando ordenamos palavras ou textos).

Algoritmos de Ordenação

- Os tipos de ordenação vão dos mais simples:
  - **Bubble sort** (Ordenação por trocas)
  - **Selection sort** (Ordenação por seleção)
  - **Insertion sort** (Ordenação por inserção)

  Aos mais **sofisticados** como:
  - **Count sort**
  - **Quick sort**
  - **Merge sort**
  - **Heap sort**
  - **Shell sort**

  entre outros....

Bubble Sort – Ord. bolha

Bubble sort, ou ordenação por flutuação (literalmente "por bolha"), é um dos mais simples algoritmos de ordenação. A idéia é percorrer o vetor diversas vezes, a cada passagem fazendo flutuar para o topo o maior elemento da sequência.

A complexidade desse algoritmo é de ordem quadrática, por isso, ele não é recomendado para situações que precisem de velocidade e operem com grandes quantidades de dados.

```
void bubble(int v[], int tam) {
     int i, aux,trocou;
     do {
          tam--;
          trocou = 0; //usado para otimizar o algoritmo
          for(i = 0; i < tam; i++)
              if(v[i] > v[i + 1]) {
                 aux=v[i];
                 v[i]=v[i+1];
                 v[i+1]=aux;
                 trocou = 1;
              }
        } while(trocou);
}
```

Ordenação - bolha

Ordenação bolha:

- processo básico:
  - quando dois elementos estão fora de ordem, troque-os de posição até que o i-ésimo elemento de maior valor do vetor seja levado para as posições finais do vetor
- continue o processo até que todo o vetor esteja ordenado

Maior elemento

v0	4	2	5	1
v1	2	4	5	1
v2	2	4	5	1
v3	2	4	1	5
	0	1	2	3

2º maior elemento

v4	2	4	1	5
v5	2	4	1	5
	2	1	4	5
	0	1	2	3

3º maior elemento

v6	2	1	4	5
	1	2	4	5
	0	1	2	3

Ordenação - Bolha

25 48 37 12 57 86 33 92	*25x48*
25 48 37 12 57 86 33 92	*48x37 troca*
25 37 48 12 57 86 33 92	*48x12 troca*
25 37 12 48 57 86 33 92	*48x57*
25 37 12 48 57 86 33 92	*57x86*
25 37 12 48 57 86 33 92	*86x33 troca*
25 37 12 48 57 33 86 92	*86x92*
25 37 12 48 57 33 86 92	final da primeira passada

o maior elemento, 92, já está na sua posição final

Ordenação - Bolha

25 37 12 48 57 33 86 92	*25x37*
25 37 12 48 57 33 86 92	*37x12 troca*
25 12 37 48 57 33 86 92	*37x48*
25 12 37 48 57 33 86 92	*48x57*
25 12 37 48 57 33 86 92	*57x33 troca*
25 12 37 48 33 57 86 92	*57x86*
25 12 37 48 33 57 86 92	final da segunda passada

o segundo maior elemento, 86, já está na sua posição final

```
25 12 37 48 33 57 86 92    25x12 troca
12 25 37 48 33 57 86 92    25x37
12 25 37 48 33 57 86 92    37x48
12 25 37 48 33 57 86 92    48x33 troca
12 25 37 33 48 57 86 92    48x57
12 25 37 33 48 57 86 92    final da terceira passada
```

Idem para 57.

```
12 25 37 33 48 57 86 92    12x25
12 25 37 33 48 57 86 92    25x37
12 25 37 33 48 57 86 92    37x33 troca
12 25 33 37 48 57 86 92    37x48
12 25 33 37 48 57 86 92    final da quarta passada
```

Idem para 48.

```
12 25 33 37 48 57 86 92    12x25
12 25 33 37 48 57 86 92    25x33
12 25 33 37 48 57 86 92    33x37
12 25 33 37 48 57 86 92    final da quinta passada
```

Idem para 37.

```
12 25 33 37 48 57 86 92    12x25
12 25 33 37 48 57 86 92    25x33
12 25 33 37 48 57 86 92    final da sexta passada
```

Idem para 33.

```
12 25 33 37 48 57 86 92    12x25
12 25 33 37 48 57 86 92    final da sétima passada
```

Idem para 25 e, conseqüentemente, 12.

```
12 25 33 37 48 57 86 92    final da ordenação
```

Implementação Iterativa(I):

```
/* Ordenação bolha */
void bolha (int n, int* v)
{
   int i,j;
   for (i=n-1; i>=1; i--)
      for (j=0; j<i; j++)
         if (v[j]>v[j+1]) {
            int temp = v[j];      /* troca */
            v[j] = v[j+1];
            v[j+1] = temp;
         }
}
```

maior elemento
(n=4; i=n-1=3)

4	2	5	1
2	4	5	1
2	4	5	1
2	4	1	5

2º maior elemento
(i=n-2=2)

2	4	1	5
2	4	1	5
2	1	4	5

3º maior elemento
(i=n-3=1)

2	1	4	5
1	2	4	5

0 1 2 3

Implementação Iterativa (II):

```
/* Ordenação bolha (2a. versão) */
void bolha (int n, int* v)
{  int i, j;
   for (i=n-1; i>0; i--) {
        int troca = 0;
        for (j=0; j<i; j++)
            if (v[j]>v[j+1]) {
                  int temp = v[j];  /* troca */
                  v[j] = v[j+1];
                  v[j+1] = temp;
                  troca = 1;
            }
      if (troca == 0) return;    /* não houve troca */
   }
}
```

pára quando há

uma passagem inteira

sem trocas

Esforço computacional:

- esforço computacional ≅ número de comparações
  ≅ número máximo de trocas
  - primeira passada: n-1 comparações
  - segunda passada: n-2 comparações
  - terceira passada: n-3 comparações
  - ...

- tempo total gasto pelo algoritmo:
  - T proporcional a (n-1) + (n-2) + ... + 2 + 1 = (n-1+1)*(n-1) / 2 = n*(n-1) / 2
  - algoritmo de ordem quadrática: $O(n^2)$

Selection Sort

Selection sort, *ou* ordenação por seleção, é um algoritmo de ordenação que procura passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os (n-1) elementos restantes, até os últimos dois elementos.

```
void selection_sort(int num[], int tam) {
 int i, j, min;
 for (i = 0; i < (tam-1); i++) {
    min = i;
    for (j = (i+1); j < tam; j++) {
       if(num[j] < num[min]) {
           min = j;
       }
    }
    if (i != min) {
      int swap = num[i];
      num[i] = num[min];
      num[min] = swap;
    }
  }
}
```

Insertion Sort

Insertion sort, ou *ordenação por inserção*, é um algoritimo simples e eficiente quando aplicado a um pequeno número de elementos pouco desordenados.
Em termos gerais, ele percorre um vetor de elementos da esquerda para a direita e à medida que avança vai deixando os elementos mais à esquerda ordenados.

```
void insertionSort(int v[], int n) {
  int i, j, chave;
  for(j=1; j<n; j++) {
       chave = v[j];
       i = j-1;
       while(i >= 0 && v[i] > chave){
              v[i+1] = v[i];
              i--;
       }
        v[i+1] = chave;
    }
 }
```

Count Sort

A ***Ordenação por contagem*** é um método relativamente simples, mas que usa um vetor auxiliar de mesma dimensão do vetor original, o que pode ser ruim pois consome o dobro da memória em relação aos métodos de ordenação simples. São passados para a função o vetor original, o vetor ordenado e o tamanho de ambos.
O método consiste em contar quantos elementos são menores que o examinado, este número será o índice do elemento durante a sua ordenação.

```
void ordena_por_contagem(int vet[], int ord[], int n){
    int i,j,p;
   //determinar a posição de cada elemento do vetor quando
ordenado
   for(i=0;i<n;i++){
       p=0;
       for(j=0;j<n;j++)
          if (vet[i]>vet[j]) p++;
       ord[p]=vet[i];
    }
}
```

QuickSort

O **Quicksort** é um método de ordenação muito rápido e eficiente, inventado por C. A. Hoare em 1960, quando visitou a Universidade de Moscou como estudante. Ele criou o '*Quicksort'* ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rapidamente. Foi publicado em 1962 após uma série de refinamentos.

O Quicksort é um algoritmo de ordenação **não-estável** (isto é, dados iguais podem ficar fora da ordem original de entrada).

QuickSort

O Quicksort adota a estratégia de **divisão e conquista**. Os passos são:

1. Escolher um elemento da lista, denominado *pivô*;
2. Rearranjar a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao final do processo o pivô estará em sua posição final e haverá duas sublistas não ordenadas. Essa operação é denominada *partição*;
3. Recursivamente ordenar a sublista dos elementos menores e a sublista dos elementos maiores;
4. A base da recursão são as listas de tamanho zero ou um, que estão sempre ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

QuickSort

```
void quicksort (int v[], int primeiro, int ultimo)
{
  int i, j, m, aux;
  i=primeiro; j=ultimo;
  m=v[(i+j)/2];
  do {
    while (v[i] < m) i++;
    while (v[j] > m) j--;
    if (i<=j) {
         aux=v[i];
         v[i]=v[j];
         v[j]=aux;
         i++; j--;
    }
   } while (i<=j);
//continuação
  if (primeiro<j)
      quicksort(v,primeiro,j);
   if (ultimo>i)
      quicksort(v,i,ultimo);
}
```

QuickSort

```
void swap(int* a, int* b) {
   int tmp;
   tmp = *a;
   *a = *b;
   *b = tmp;
}

int partition(int vec[], int left, int right) {
 int i, j;
 i = left;
 for (j = left + 1; j <= right; ++j) {
   if (vec[j] < vec[left]) {
       ++i;
       swap(&vec[i], &vec[j]);
   }
 }
 swap(&vec[left], &vec[i]);
 return i;
 }

 void quickSort(int vec[], int left, int right) {
   int r;
   if (right > left) {
      r = partition(vec, left, right);
      quickSort(vec, left, r - 1);
      quickSort(vec, r + 1, right);
    }
}
```

MergeSort

O Merge sort, ou ordenação por intercalação, é um exemplo de algoritmo de ordenação do tipo dividir-para-conquistar. Sua idéia básica é criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, ele divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes. Estas 2 partes são então combinadas para se chegar ao resultado final.

Portanto, os 3 passos seguidos pelo MergeSort são:

1. Dividir: Dividir os dados em subsequências pequenas;
2. Conquistar: Classificar as duas metades recursivamente aplicando o merge sort;
3. Combinar: Juntar as duas metades em um único conjunto já ordenado.

Considerações

- Existem muitos métodos diferentes de ordenação de dados.
- Métodos "inplace" significam que a ordenação vai ocorrer no próprio vetor (original).
- A estabilidade também é um fator a se considerar. Ela se refere à ordem em que dados repetidos vão aparecer. Quando esta ordem é mantida diz-se que o algoritmo é estável.
- O mais indicado vai depender da quantidade e do grau de ordenação(ou, desordenação) dos mesmos.
- Um fator a se considerar é a **complexidade** destes algoritmos. Esta medida se refere ao tempo estimado para que a ordenação ocorra.
- Existem várias páginas na Internet tratando deste assunto. Muitas delas contem animações que mostram como o algoritmo se comporta.
- Alguns links:
  - http://pt.wikipedia.org/wiki/Algoritmos_de_ordena%C3%A7%C3%A3o
  - http://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html
  - animações
    - http://people.cs.ubc.ca/~harrison/Java/sorting-demo.html
    - http://cg.scs.carleton.ca/~morin/misc/sortalg/