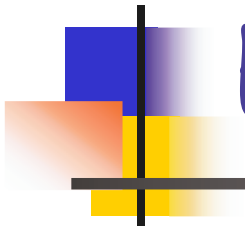
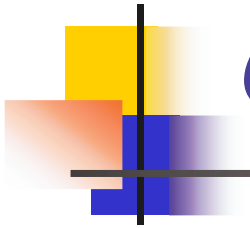


Métodos de Ordenação Elementares

Algoritmos e Estuturas de Dados

Inverno 2006

Cátia Vaz



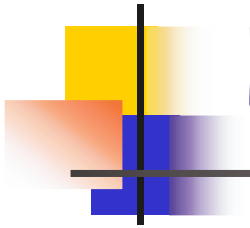
Contexto

- **Objectivo:** estudar métodos de ordenação de qualquer tipo de dados.
- Para a ordenação de um determinado tipo de dados, é associado a cada elemento uma **chave** que o caracteriza.
  - exemplo: a ordem natural do tipo de dados
- **Interface *Comparable***
  - A classe que descreve o tipo de dados tem que implementar a interface **`Comparable`** para especificar uma ordem natural:
    - o método **`compareTo`** especifica uma **ordem total**;
    - exemplo de tipos comparáveis: **`String`**, **`Integer`** e **`Double`**
    - para que os objectos de uma classe definida pelo utilizador sejam comparáveis, é necessário que a classe implemente a interface **`Comparable`**.

Operações Abstractas

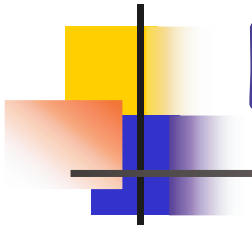
- As operações abstractas nos dados a utilizar são:
  - a comparação (**less**);
  - a troca (**exch**).

```
static boolean less(int v, int w) {
   return v < w;}

static void exch(int[] a, int i, int j) {
   int t = a[i];
   a[i] = a[j];
   a[j] = t;}

static void compExch(int[] a, int i, int j) {
   if (less(a[i],a[j])) exch(a,i,j);}
```



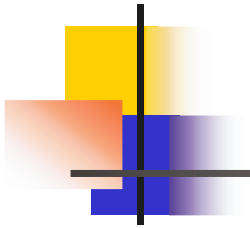
Definições

- **Tipos de algoritmos de ordenação:**
  - **não adaptativos** – sequência de operações independente da ordenação original dos dados;
  - **adaptativos** – sequência de operações dependente do resultado de comparações.
- Um algoritmo de ordenação é dito **estável** se preserva a ordem relativa dos elementos com chaves repetidas
- Um algoritmo de ordenação é dito **interno**, se o conjunto de todos os dados a ordenar couber na memória; caso contrário é dito **externo**.



Definições

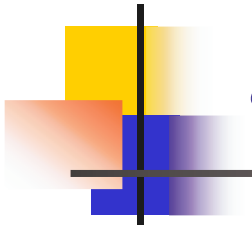
- Um algoritmo de ordenação é dito **directo** se os dados são acedidos directamente nas operações de comparação e troca; caso contrário é dito **indirecto**.



Selection Sort

- Ideia do Algoritmo:
  - procurar o menor elemento e trocar com o elemento na primeira posição
  - procurar o segundo menor elemento e trocar com o elemento na segunda posição
  - proceder assim até a ordenação estar completa

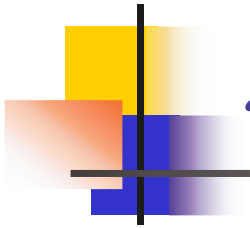
```
static void selectionSort(int[] a, int left, int right){
   int min;
   for(int i= left; i<right; i++){
        min=i;
        for(int j=i+1; j<=right;j++)
               if( less(a[j], a[min] ) min=j;
        exch(a,i,min);
   }
}
```



Selection Sort

- **Propriedade**: *Selection Sort* usa aproximadamente $N^2/2$ comparações e **N** trocas.
  - para cada item **i** de **0** a **N-2** há uma troca e **N-1-i** comparações.

    Logo há **N-1** trocas e **(N-1)+(N-2)+...+2+1=N(N-1)/2** comparações
- **Nota:** o desempenho pouco depende da ordenação inicial dos dados; o que depende desta é o número de vezes que min é actualizado.



Insertion Sort

- Ideia do Algoritmo:
  - considerar os elementos um a um e inseri-los no seu lugar entre os elementos já tratados (mantendo essa ordenação)

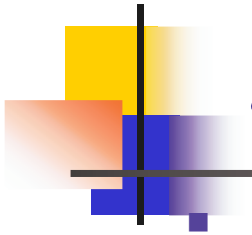
```
static void insertionSort(int[] a, int left, int right){
   for(int i = left+1; i <= right; i++)
        for (j = i; j > left; j--)
               compexch (a,j,j-1);
}
```

- Pode ser melhorado o desempenho deste algoritmo caso se saia do ciclo interno se `less(a[j],a[j-1])` for verdadeira:
  - aumenta o desempenho aproximadamente por um factor de 2.
  - esta modificação torna o algoritmo adaptativo.

Insertion Sort – Versão Adaptativa

```
static void insertionSort(int[] a, int left, int right){
   for(int i = left+1; i <= right; i++) {
        int v = a[i];
        int j = i;
        while (j > left && less(v, a[j-1])) {
               a[j] = a[j-1]; j--;
        }
        a[j] = v;
   }
}
```

- **Propriedade**: *Insertion Sort* usa aproximadamente $N^2/4$ comparações e $N^2/4$ trocas no caso médio e o dobro destes valores no pior caso.

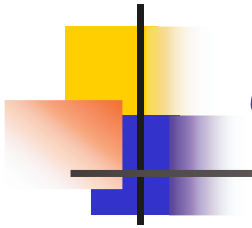


Bubble Sort

- Ideia do Algoritmo :
  - quando o menor elemento é encontrado é sucessivamente trocado com todos à sua esquerda e acaba por ficar na 1ª posição
  - quando o segundo elemento mais pequeno é encontrado é sucessivamente trocado com todos à sua esquerda (excepto o elemento na 1ª posição e acaba por ficar na 2ª posição
  - proceder assim até a ordenação estar completa

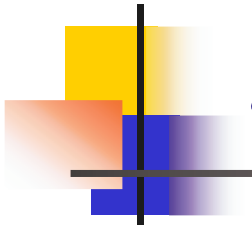
```
static void bubbleSort(int[] a, int left, int right){
   for (int i = left; i < right; i++)
        for (int j = right; j > i; j--)
              compexch(a,j,j-1);
}
```

- **O algoritmo pode ser melhorado, tal como o algoritmo de inserção**



Exercício

- Exercício1: Desenvolver uma implementação mais eficiente do algoritmo *Bubble Sort*.

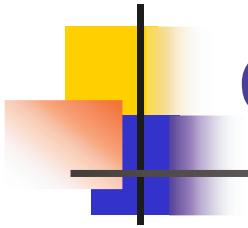


Bubble Sort

- **Propriedade**: *Bubble sort* usa aproximadamente $N^2/2$ comparações e $N^2/2$ trocas no caso médio e no pior caso
  - para cada item **i** de **0** a **N-2** há **N-1-i** trocas e **N-1-i** comparações.

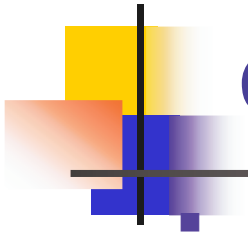
    Logo **(N-1)+(N-2)+...+2+1= N(N-1)/2** comparações e **N(N-1)/2** trocas
- **Nota**: o algoritmo pode depender criticamente dos dados se for modificado para terminar quando não houver mais trocas

Outras propriedades dos algoritmos de ordenação



- Para N elementos grandes com chaves pequenas, o *Selection Sort* é linear:
  - Seja M o racio do tamanho do elemento para o tamanho da chave.
    - Pode assumir-se que o custo da comparação é 1 unidade de tempo e que o custo de troca é M unidades de tempo.
    - $N^2/2$ unidades de tempo para comparações
    - **NM** unidades de tempo para trocas
    - SE M for maior que um múltiplo de N, então o custo é proporcional ao tempo para mover os elementos, isto é a **NM**.

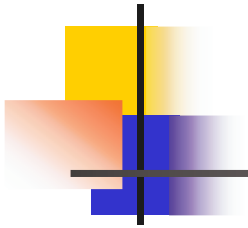
Outras propriedades dos algoritmos de ordenação



- Uma **inversão** é um par de chaves que está fora de ordem numa sequência de N elementos.
- Para contar o número de inversões numa sequência de N elementos, pode adicionar-se, para cada elemento, **o número de elementos à sua esquerda que são maiores**. Designamos esta quantidade por número de inversões correspondentes a cada elemento.
- **Propriedade:** *Insertion Sort* e *Bubble Sort* usam um número linear de comparações e trocas para ordenar N elementos com, **no máximo, um número constante de inversões correspondentes a cada elemento.**
  - O tempo de execução do *insertion sort* depende do número total de inversões da sequência dos **N** elementos
  - Cada passo do *bubble sort* reduz o número de elementos mais pequenos à direita de qualquer elemento de **1**.

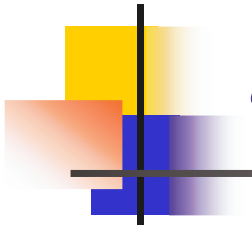
Outras propriedades dos algoritmos de ordenação

- **Propriedade:** *Insertion Sort* usa um número linear de comparações e trocas para ordenar N elementos com, **no máximo, um número constante de elementos que podem ter mais do que um número constante de inversões correspondentes**.
  - O tempo de execução do *insertion sort* depende do número total de inversões da sequência dos **N** elementos e não da forma como essas inversões estão distribuídas.



Shell Sort

- Ideia do Algoritmo:
  - o array é dividido em partições, cada uma contendo os objectos de (índice%h)
    - para uma tabela de **15** posições e **h=4**
      - índices resto **0**, **i%4=0**, são: **0, 4, 8, 12**
      - índices resto **1**, **i%4=1**, são: **1, 5, 9, 13**
      - índices resto **2**, **i%4=2**, são: **2, 6, 10, 14**
      - índices resto **3**, **i%4=3**, são: **3, 7, 11,**
    - dentro de cada partição é feita uma ordenação por inserção



Shell Sort

- quando **h-ordenamos** os dados inserimos qualquer elemento entre os da sua **h-partição** movendo elementos maiores para a direita
  - basta usar ***insertion sort*** com incrementos/decrementos de **h** em vez de 1.
- Os dados não ficam completamente ordenados depois da primeira ordenação das várias partições
  - não basta escolher um só **h.**
- Escolhe-se uma sequência de partições. Qualquer sequência de incremento de partições irá funcionar desde que a última partição seja 1.
- implementação de *Shell sort* usa um passo de *Insertion sort* pelo dados para cada incremento.

Shell Sort

- Exemplo para h=4. Considere-se um array de números como [13,14,94,33,82, 25, 59, 94, 65, 23, 45, 27, 73, 25, 39].
  - podemos **visualizar** isto como quebrar a lista de numeros numa matriz com h colunas.

$$\begin{pmatrix} 13 & 14 & 94 & 33 \\ 82 & 25 & 59 & 94 \\ 65 & 23 & 45 & 27 \\ 73 & 25 & 39 & \end{pmatrix}$$

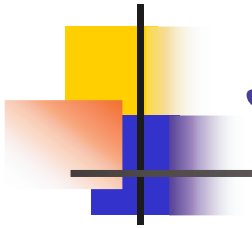
  - quando h-ordenarmos cada coluna utilizando insertion sort ficamos com

$$\begin{pmatrix} 13 & 14 & 39 & 27 \\ 65 & 23 & 45 & 33 \\ 73 & 25 & 59 & 94 \\ 82 & 25 & 94 & \end{pmatrix}$$

  - repetir a h-ordenação para h-1,h-2, ... 1.

Shell Sort

```
public static void shell(int[] a, int left, int right){
   int h;
   for (h = 1; h <= (right-left)/9; h = 3*h+1);
   for ( ; h > 0; h /= 3)
       for (int i = left+h; i <= right; i++){
           int j = i;
           int v = a[i];
           while (j >= left+h && less(v, a[j-h])){
              a[j] = a[j-h];
              j -= h;}
           a[j] = v; }
}
```



Sequência de ordenação

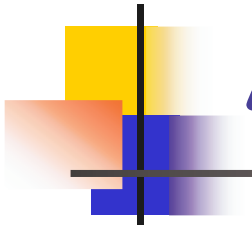
- Difícil de escolher
  - propriedades de muitas sequências foram já estudadas
  - possível provar que umas melhores que outras
    - ex: 1, 4, 13, 40, 121, 364, 1093, 3280, ... (Knuth, 3*hant+1)

    é melhor do que

    1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... (Shell, 2i)

    (porque os elementos nas posições ímpares apenas são comparados com os elementos nas posições pares na última passagem).

    mas pior(20%) que 1, 8, 23, 77, 281, 1073, 4193, ... (4i+1+ 3 2i+ 1)
- na prática utilizam-se sequências que decrescem geometricamente para que o número de incrementos seja logarítmico

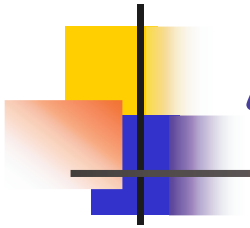


Análise de *Shell Sort*

- **a sequência óptima não foi ainda descoberta (se é que existe)**
- **análise do algoritmo é desconhecida**
  - ninguém encontrou a fórmula que define a complexidade
  - complexidade depende da sequência

- **Propriedade**: o resultado de **h-ordenar** uma sequência de dados que está **k-ordenada** é uma sequência de dados que está simultaneamente **h-** e **k-ordenada**.

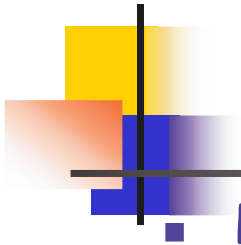
- **Propriedade:** *Shellsort* faz menos do que $O(N^{3/2})$ comparações para os incrementos 1, 4, 13, 40, 121, 364, 1093, ...
  (Knuth, 3*hant+1)



Análise de Shell Sort

- **Propriedade:** *Shellsort* faz menos do que $O(N^{4/3})$ comparações para os incrementos 1, 8, 23, 77, 281, 1073, 4193, 16577, ... (4i+1+ 3 2i+ 1)

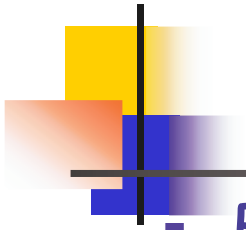
- **Propriedade:** *Shellsort* faz menos do que $O(N \log^2 N)$ comparações para os incrementos 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81,...
- **Vantagens:**
  - rápido/eficiente
  - pouco código
  - **melhor** método para ficheiros pequenos e médios
  - aceitável para elevados volumes de dados



Key-Indexed Counting

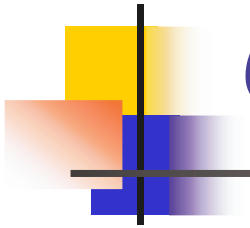
- Problema: N elementos cujas chaves são inteiros distintos entre 0 e M-1.
  - **Solução: ordenar utilizando as chaves como índices (se M não for demasiado grande).**
    - cnt um array de contadores.
      - inicialmente cnt[i]=0 para 0<=i<M;
      - cnt[i] irá contar o número de chaves menor do que i;

```
static void distCount(int[] a, int left, int right){
   int i,j,cnt[]=new int[M];
   int b[]=new int[a.lenght];
   for(i= left;i<= right;j++) cnt[a[i]+1]++;
   for(j=1;j<M;j++) cnt[j]=cnt[j-1];
   for(i= left;i<= right;i++) b[cnt[a[i]]++]=a[i];
   for(int i=left;i<= right;i++) a[i]=b[i-left];
}
```



Key-Indexed Counting

- **Propriedade:** O algoritmo de contabilização de chaves indexadas é um algoritmo de ordenação em tempo linear se o número de chaves distintas é um factor constante da número de elementos a ordenar.



Ordenação dos dados

- Ordenar objectos sem ordem natural ou com uma ordem diferente
  - Ex. Ordenar **`Strings`** por:
    - ordem natural -> "Now is the time"
    - *case insensitive* -> "is Now the time"
- Interface Comparator.
  - A classe que **implementar esta interface** tem que **implementar o método compare** tal que **`compare(v, w)`** especifique uma ordem total.
  - Vantagem: separa a definição do tipo de dados da forma de comparação de dois objectos daquele tipo.
    - adiciona uma nova ordem a um tipo de dados.
    - adiciona uma ordem a um tipo de dados de uma biblioteca que não tenha ordem natural.

Exemplo de alterações para utilizar um *Comparator*

```
static boolean less(Comparator c, Item v, Item w){
   if(c==null){ return v.compareTo(w);}
   return c.compare(v, w) < 0;}

static void exch(Item[] a, int i, int j) {
   Item t = a[i]; a[i] = a[j]; a[j] = t;}

static void insertionSort(Item[] a,int left,int right,Comparator c){
   for(int i = left+1; i <=right; i++) {
        Item v = a[i];
        j = i;
        while (j > left && less(c, v, a[j-1])) {
                a[j] = a[j-1]; j--;
        }
        a[j] = v;
}
```

Cátia Vaz