UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
Departamento de Ciências de Computação

SCC-501 - Capítulo 4
Métodos de Ordenação

João Luís Garcia Rosa[1]

[1]Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
http://www.icmc.usp.br/~joaoluis

2011

Sumário

Sumário

Importância da Ordenação

Ordenação (ou **Classificação**): Tornar mais simples, rápida e viável a recuperação de uma determinada informação, num conjunto grande de informações.

Sumário

Terminologia básica

- **arquivo** de tamanho *n* é uma sequência de *n* itens: r[0], r[1], r[2], ... , r[n-1]
- cada item no arquivo é chamado de **registro.**
- uma **chave** k[i], é associada a cada registro r[i]
- a chave geralmente é um campo do registro
- **ordenação pela chave** é quando os registros são classificados por um campo chave.

Terminologia básica

- **ordenação interna**: dados estão na memória principal;
- **ordenação externa**: os dados estão em um meio auxiliar;
- **ordenação estável**: se, para os registros *i* e *j*, k[i] igual a k[j]; se r[i] precede r[j] no arquivo original, r[i] precederá r[j] no arquivo classificado. Um algoritmo de ordenação é dito **estável**, se ele preserva a ordem relativa original dos registros com mesmo valor de chave.
- a ordenação ocorre sobre os próprios registros ou sobre uma tabela auxiliar de ponteiros:

Terminologia básica

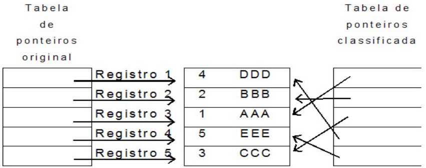
Figuras [2]:

Registro 1	4	DDD
Registro 2	2	BBB
Registro 3	1	AAA
Registro 4	5	EEE
Registro 5	3	CCC

Arquivo Original

1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

Arquivo classificado



CLASSIFICAÇÃO POR ENDEREÇOS
(Normalmente utilizada em arquivos muito grandes)

Terminologia básica

ATENÇÃO

Não existe um método de ordenação considerado universalmente superior a todos os outros. É necessário analisar o problema e, com base nas características dos dados, decidir qual o método que melhor se aplica à ele.

Sumário

Considerações sobre a Eficiência

- Alguns aspectos de medida de eficiência:
    - tempo para codificação do programa de ordenação;
    - tempo de máquina para a execução do programa;
    - espaço de memória necessário.
- Normalmente o tempo gasto é medido pelo número de operações críticas, ao problema, que são efetuadas.
- Nesse caso as operações críticas são:
    - comparações de chaves;
    - movimentação de registros ou de ponteiros;
    - troca entre dois registros.
- A medida é tomada pela análise do melhor caso, do pior caso e do caso médio.
- O resultado é uma fórmula em função de n (número de registros do arquivo).

Considerações sobre a Eficiência

- Na realidade o tempo gasto não depende exclusivamente do tamanho do arquivo, mas também de outros aspectos, por exemplo: se existe uma pré ordenação no arquivo ou não.

Tipos de Ordenação

Ordenação:

- por Troca
- por Seleção
- por Inserção
- de Shell
- por Cálculo de Endereço
- por Intercalação
- por Contagem de Menores
- por Contagem de Tipos
- de Raízes

Sumário

Ordenação por Troca

- 2 tipos:
  - Bolha (*bubble sort*)
  - *Quicksort*
- Bolha
  - Característica do método:
    - algoritmo fácil
    - pouco eficiente
  - Idéia básica é percorrer o arquivo sequencialmente várias vezes. Cada passagem consiste em comparar cada elemento no arquivo e seu sucessor (x[i] com x[i+1]) e trocar os dois elementos se não estiverem na ordem certa.

Ordenação por Troca: Bolha

- Exemplo: seja o seguinte arquivo

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- Primeira passagem:
  - x[0] com x[1] (25 com 57): nenhuma troca
  - x[1] com x[2] (57 com 48): troca
  - x[2] com x[3] (57 com 37): troca
  - x[3] com x[4] (57 com 12): troca
  - x[4] com x[5] (57 com 92): nenhuma troca
  - x[5] com x[6] (92 com 86): troca
  - x[6] com x[7] (92 com 33): troca

Ordenação por Troca: Bolha

- Depois da primeira passagem:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	48	37	12	57	86	33	92

- 92 é o maior elemento do conjunto
- Próxima passada não compara mais com esse elemento pois já está na posição correta. E assim por diante.
- x é um vetor com números inteiros a ordenar
- n é o número de elementos do vetor

Ordenação por Troca: Bolha

- O conjunto completo de iterações:

iteração	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
0	25	57	48	37	12	92	86	33
1	25	48	37	12	57	86	33	92
2	25	37	12	48	57	33	86	92
3	25	12	37	48	33	57	86	92
4	12	25	37	33	48	57	86	92
5	12	25	33	37	48	57	86	92
6	12	25	33	37	48	57	86	92

Ordenação por Bolha - Versão 1

```
bolha(int v[], int T)
{
 int i, aux, troca = 1;
 while (troca) // 6 vezes para o exemplo anterior
 {
  troca = 0;
  for (i = 0; i < T-1; i++) // 42 vezes
    if (v[i] > v[i+1])
    {
      aux = v[i];
      v[i] = v[i+1];
      v[i+1] = aux;
      troca = 1;
    }
 }
}
```

Ordenação por Bolha - Versão 2

```
bolha(int v[], int T)
{
 int i, j, aux, troca = 1;
 for (i = 0; i < T-1 && troca; i++) // 6 vezes
 {
  troca = 0;
  for (j = 0; j < T-i-1; j++)// 27 vezes
  {
   if (v[j] > v[j+1])
   {
    troca = 1;
    aux = v[j];
    v[j] = v[j+1];
    v[j+1] = aux;
   }
  }
 }
```

Ordenação por Bolha

- **Observação**: Os algoritmos apresentados trazem dois aperfeiçoamentos que tornaram o método melhor:
  - teste de parada quando já ordenada (versões 1 e 2);
  - não efetuar as comparações até o final do vetor em cada passo, considerando-se que os elementos vão ficando na posição correta em cada passo (apenas versão 2).
- Eficiência do método da bolha sem considerar as melhorias descritas acima:
  - $(n-1)$ comparações nas $(n-1)$ passagens
  - total de comparações: $(n-1)*(n-1) = n^2 - 2n + 1$, que é $\mathcal{O}(n^2)$
  - o número de trocas não é maior que o número de comparações, mas gasta mais tempo na execução.

Ordenação por Bolha

- Teste de parada trará vantagem quando se trabalha com arquivo pré-ordenado. Deve-se, porém, considerar o número de vezes que a variável *troca* é testada e recebe valor!
- A redução do número de comparações em cada passagem:
  - 1a. Passagem: $n - 1$ comparações
  - 2a. Passagem: $n - 2$ comparações
  - 3a. Passagem: $n - 3$ comparações
  - ...
  - $(n - 1)a$. Passagem: 1 comparação
- soma: $1 + 2 + 3 + 4 + ... + (n - 1) = \frac{(n^2 - n)}{2}$
- portanto: $\mathcal{O}(n^2)$

Ordenação por Troca de Partição: *quicksort*

- Seja x um vetor a ordenar e n seu número de elementos.
- Idéia básica:
  1. Tome dois ponteiros i e j. Inicie i com 0 e j com $n - 1$;
  2. Compare $x[i]$ com $x[j]$, se $x[i] \leq x[j]$ não é necessário trocar, decremente j e continue comparando;
  3. Se $x[i] > x[j]$, troque $x[i]$ com $x[j]$, incremente i e continue comparando, até a próxima troca;
  4. Após a próxima troca, decremente j.
  5. Repita os passos **2** a **4** até $i = j$.
- Após essa primeira passagem, o elemento $x[i]$ (que é o mesmo que $x[j]$, pois $i = j$) está na posição correta. Os elementos à esquerda dele são menores e os da direita são maiores.
- Aplicar novamente o método para as duas partições.

Ordenação por Troca de Partição: *quicksort*

- Tome o seguinte exemplo:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
$[25]_i$	57	48	37	12	92	86	$[33]_j$

- 25 com 33. Não troca:j–:$[25]_i$-57-48-37-12-92-$[86]_j$-33
- 25 com 86. Não troca: j–:$[25]_i$-57-48-37-12-$[92]_j$-86-33
- 25 com 92. Não troca: j–:$[25]_i$-57-48-37-$[12]_j$-92-86-33
- 25 com 12. **Troca**: i++:12-$[57]_i$-48-37-$[25]_j$-92-86-33
- 57 com 25. **Troca**: j–:12-$[25]_i$-48-$[37]_j$-57-92-86-33
- 25 com 37. Não troca: j–:12-$[25]_i$-$[48]_j$-37-57-92-86-33
- 25 com 48. Não troca: j–:12-$[25]_{i,j}$-48-37-57-92-86-33
- $i = j$: Pára. 25 está na posição correta ($x[1]$). Duas partições: (12) e (48-37-57-92-86-33).

Ordenação por Troca de Partição: *quicksort*

- Classifica-se primeiro a partição da esquerda (12) colocando a partição da direita (48-37-57-92-86-33) numa pilha.
- Como um arquivo de um elemento já está classificado (12 é $x[0]$), o processo se repete para a partição da direita: $[48]_i$-37-57-92-86-$[33]_j$.
- Troca 48 com 33: i++: 12-25-(33-$[37]_i$-57-92-86-$[48]_j$)
- Não troca 37 com 48: i++: 12-25-(33-37-$[57]_i$-92-86-$[48]_j$)
- Troca 57 com 48: j–: 12-25-(33-37-$[48]_i$-92-$[86]_j$-57)
- Não troca 48 com 86: j–: 12-25-(33-37-$[48]_i$-$[92]_j$-86-57)
- Não troca 48 com 92: j–: 12-25-(33-37-$[48]_{i,j}$-92-86-57)
- $i = j$: Pára. 48 está na posição correta ($x[4]$). Duas partições: (33-37) e (92-86-57).

Ordenação por Troca de Partição: *quicksort*

- Classifica-se a partição da esquerda (33-37) colocando a da direita (92-86-57) na pilha.
- Não troca 33 com 37: j–: 12-25-($[33]_{i,j}$-37)-48-92-86-57.
- $i = j$: Pára. 33 está na posição correta ($x[2]$). Como a partição da esquerda é vazia e a da direita contém um único elemento (37 é $x[3]$), pega-se a partição da pilha ($[92]_i$-86-$[57]_j$).
- Troca 92 com 57: i++: 12-25-33-37-48-(57-$[86]_i$-$[92]_j$)
- Não troca 86 com 92: i++: 12-25-33-37-48-(57-86-$[92]_{i,j}$)
- $i = j$: Pára. 92 ($x[7]$) está na posição correta. Classifica-se a partição da esquerda 57-86 e coloca-se a partição da direita (vazia) na pilha.
- Não troca 57 com 86: j–: 12-25-33-37-48-($[57]_{i,j}$-86)-92.
- $i = j$: Pára. 57 ($x[5]$) está na posição correta.

Ordenação por Troca de Partição: *quicksort*

- Como a partição da esquerda é vazia e a da direita contém um único elemento ($86 = x[6]$), e a pilha está vazia, vetor classificado:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
12	25	33	37	48	57	86	92

- O algoritmo *quicksort* pode ser definido mais adequadamente como um procedimento recursivo, como será visto no próximo capítulo.
- Por ora, apresenta-se um procedimento iterativo para o *quicksort*.

Programa *quicksort*

```
void quicksort(int x[], int n)
{
 struct elemtipo elemento; struct LIFO pilha; int i,j;
 pilha.top = -1; elemento.inf = 0; elemento.sup = n-1;
 push(&pilha, &elemento);
 while (!vazia(&pilha)) // repete enquanto existir algum subvetor não classificado
 {
   pop(&pilha,&elemento);
   while (elemento.inf < elemento.sup)
   { // processa a subvetor seguinte
     particao(x, elemento.inf, elemento.sup, &j);
      // empilha o subvetor maior
     if (j-elemento.inf > elemento.sup-j)
     { // empilha o subvetor inferior
       i = elemento.sup;
       elemento.sup = j-1;
       push(&pilha, &elemento);
        // processa o subvetor superior
       elemento.inf = j+1; elemento.sup = i;
     }
     else
     { // empilha o subvetor superior
       i = elemento.inf;
       elemento.inf = j+1;
       push(&pilha, &elemento);
        // processa o subvetor inferior
       elemento.inf = i; elemento.sup = j-1;
     }
   }
 }
```

Programa *quicksort*

```
void particao(int x[], int min, int max, int *p)
{
  int a, i, temp, j, troca;
  a = x[min]; // a é o elemento cuja posição final é procurada (pivô)
  j = max;
  i = min;
  troca = 0;
  while (i < j)
  {
    if (!troca)
      while (x[i] <= x[j] && i < j)
        j--;
    else
      while (x[i] <= x[j] && i < max)
        i++;
    if (i < j)
    { // troca x[i] e x[j]
      temp = x[i];
      x[i] = x[j];
      x[j] = temp;
      troca = 1-troca;
    }
  }
  x[j] = a;
  *p = j;
}
```

Eficiência do *quicksort*

Suponha que:

- o tamanho do arquivo de dados n seja uma potência de 2, ou seja, $n = 2^m$, de modo que $m = log_2 n$;
- a posição correta para o pivô termine sempre sendo o meio exato do subvetor.

Assim

- ocorrerão aproximadamente n comparações (na realidade $n - 1$) na primeira passagem, após a qual o arquivo será dividido em dois sub-arquivos com tamanho $n/2$;
- para cada um destes arquivos ocorrem $n/2$ comparações e é formado um total de 4 arquivos, cada qual com o tamanho $n/4$, e assim por diante;
- depois de separar os sub-arquivos m vezes, existirão n arquivos de tamanho 1.

Eficiência do *quicksort*

- Assim, o número total de comparações para a ordenação inteira será aproximadamente

$$n + 2 * (n/2) + 4 * (n/4) + ... + n * (n/n)$$

ou

$$n + n + n + ... + n \ (m \text{ termos})$$

comparações. Assim $\mathcal{O}(n * m)$ ou $\mathcal{O}(n\ log\ n)$ (lembre-se de que $m = log_2 n$).

- Portanto, se as condições descritas anteriormente forem satisfeitas, a ordenação rápida (*quick sort*) será $\mathcal{O}(n\ log\ n)$, o que é relativamente eficiente.

Sumário

Seleção Direta

- Idéia Básica:
    1. Selecionar o maior elemento do conjunto;
    2. Trocá-lo com o último elemento;
    3. Repetir os itens (1) e (2) com os $N - 1$ elementos restantes, depois com os $N - 2$ e assim por diante até sobrar o primeiro elemento que será o menor do conjunto.

Seleção Direta

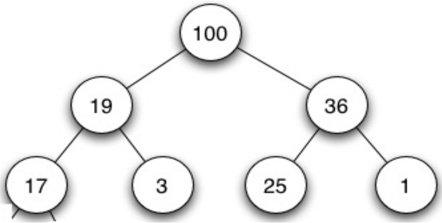
```
void selectsort(int x[], int n)
{
  int i, indx, j, maior;
  for (i = n-1; i > 0; i--)
  {
    maior = x[0];
    indx = 0;
    for (j = 1; j <= i; j++)
      if (x[j] > maior)
      {
        maior = x[j];
        indx = j;
      }
    x[indx]= x[i];
    x[i] = maior;
  }
}
```

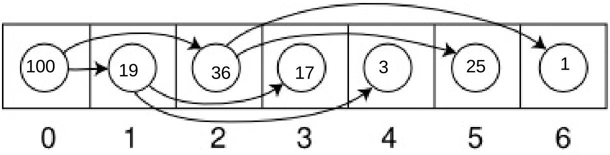
Desempenho do método: $\mathcal{O}(n^2)$

Heap Sort

- Baseado no princípio de ordenação por seleção em árvore binária.
- O método consiste em duas fases distintas: primeiro é feita a montagem da árvore binária (HEAP) contendo todos os elementos do vetor, de tal forma que o valor contido em qualquer nó seja maior do que os valores de seus sucessores e, numa segunda fase, o HEAP é usado para a seleção dos elementos na ordem desejada.

Heap





Heap Sort

- A árvore binária é construída sobre o próprio vetor de tamanho *n*, onde:
  1. o sucessor à esquerda do elemento de índice *i* é o elemento de índice $2 * (i + 1) - 1$, se $2 * (i + 1) - 1 < n$, caso contrário não existe;
  2. o sucessor à direita do elemento de índice *i* é o elemento de índice $2 * (i + 1)$, se $2 * (i + 1) < n$, caso contrário não existe.
- Dada um vetor k_0, k_1, ..., k_{n-1}, os elementos $k_{n/2}$, ..., k_{n-1} vão formar um *heap*. Esses elementos formam a linha inferior da árvore binária associada (suas folhas).

Heap Sort

- PASSOS:
  1. Expandir o *heap* $k_{n/2}$, ..., k_{n-1} para k_0, k_1, ..., k_{n-1} fazendo-se as devidas trocas de posição dos elementos. Nesse momento teremos em k_0 o maior elemento do vetor;
  2. trocar k_0 com k_{n-1};
  3. expandir o *heap* k_1, k_2, ..., k_{n-2} pela inclusão de k_0 (k_{n-1} anterior). Após essa inclusão, k_0 será o segundo maior elemento;
  4. trocar k_0 com k_{n-2} e repetir o processo até a ordenação total.

Heap Sort

```
void fazheap(int a[], int n)
{
  int i;
          // reajusta os elementos em a[0:n-1] para formar um heap
  for (i = (n-1)/2; i >= 0; i--)
    ajuste(a,i,n-1);
}


void heapsort(int a[], int n)
{
  int t, i;
          // a[0:n-1] contém n elementos a ser ordenado.
          // HeapSort rearranja-os em ordem não-decrescente.
  fazheap(a,n); // transforma o vetor em um heap
          // troca o novo máximo com o elemento no final do vetor
  for (i = n-1; i >= 1; i--)
  {
    t = a[i];
    a[i] = a[0];
    a[0] = t;
    ajuste(a, 0, i-1);
  }
}
```

Heap Sort

```
void ajuste(int a[], int i, int n)
{
         // Árvores binárias completas com raízes 2(i+1)-1 e 2(i+1) são
         // combinadas com o nó i para formar um heap com raiz i.
         // Nenhum nó tem endereço maior que n-1 ou menor que 0.
  int item, j;
  j = 2*(i+1)-1;
  item = a[i];
  while (j <= n)
  {
    if ((j < n) && (a[j] < a[j+1]))
      j = j + 1;
         // compara sucessor da esquerda com o da direita :  j é o maior
    if (item >= a[j])
      break;
         // uma posição para item é encontrada
    a[(j+1)/2-1] = a[j];
    j = 2*(j+1)-1;
  }
  a[(j+1)/2-1] = item;
}
```

Heap Sort

iteração	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
fazheap: i=3	25	57	48	37	12	92	86	33
fazheap: i=2	25	57	92	37	12	48	86	33
fazheap: i=1	25	57	92	37	12	48	86	33
fazheap: i=0	92	57	86	37	12	48	25	33
heapsort: i=7	86	57	48	37	12	33	25	92
heapsort: i=6	57	37	48	25	12	33	86	92
heapsort: i=5	48	37	33	25	12	57	86	92
heapsort: i=4	37	25	33	12	48	57	86	92
heapsort: i=3	33	25	12	37	48	57	86	92
heapsort: i=2	25	12	33	37	48	57	86	92
heapsort: i=1	12	25	33	37	48	57	86	92

Eficiência do *Heap Sort*

- À primeira vista, parece que o *heap sort* não apresenta bons resultados.
- Afinal, deve-se mover os elementos de maior valor para o início antes de serem finalmente colocados em sua posição correta (no final).
- De fato, o algoritmo não é recomendado para pequenos conjuntos de elementos.
- Mas para valores grandes de n o seu desempenho melhora muito!

Eficiência do *Heap Sort*

- No pior caso, existem $n/2$ passos de "escorregamento" para o início necessários para posicionar elementos através de $log(n/2)$, $log(n/2-1)$, ..., $log(n-1)$ posições.
- Logo, a fase de ordenação usa $n-1$ passos de escorregamento, como no máximo, $log(n-1)$, $log(n-2)$, ..., 1 movimentos respectivamente.
- Devem ser acrescentados os $n-1$ movimentos necessários para guardar o elemento recém-escorregado no extremidade direita.
- Logo, a ordenação pelo *heap sort* leva cerca de $n * log\ n$ passos, mesmo no pior caso.
- Desempenho do método: $\mathcal{O}(n\ log\ n)$ mesmo no pior caso (excelente!)

Sumário

Inserção Simples

- É o método que consiste em inserir informações num conjunto já ordenado.
- ```
  void insersort(int x[], int n)
  {
   int i, k, y;
   for (k = 0; k < n; k++)
   {
     y = x[k];
     for (i = k-1; i >= 0 && y < x[i]; i--)
       x[i+1] = x[i];
     x[i+1] = y;
   }
  }
  ```
- Desempenho: $\mathcal{O}(n^2)$
  ```

Inserção Simples

iteração	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
k=0	25	57	48	37	12	92	86	33
k=1	25	57	48	37	12	92	86	33
k=2 (for i=1)	25	**48**	**57**	37	12	92	86	33
k=3 (for i=2,1)	25	**37**	**48**	**57**	12	92	86	33
k=4 (for i=3,2,1,0)	**12**	**25**	**37**	**48**	**57**	92	86	33
k=5	12	25	37	48	57	92	86	33
k=6 (for i=5)	12	25	37	48	57	**86**	**92**	33
k=7 (for i=6,5,4,3,2)	12	25	**33**	**37**	**48**	**57**	**86**	**92**

Inserção Simples

- Variações do Método:
  - inserção com pesquisa binária: consiste em utilizar o método da busca binária para localizar a posição a ser inserido o elemento:
    - Diminui o número de comparações mas ainda é necessário efetuar o deslocamento dos elementos para a inserção.
    - Isso sendo executado n vezes resulta em $\mathcal{O}(n^2)$ substituições.
    - De modo geral não ajuda!
  - Inserção em lista ligada: consiste em não mover as informações e sim efetuar as inserções nas ligações:
    - O tempo gasto com comparações continua sendo $\mathcal{O}(n^2)$, além do uso de um vetor adicional para o link.
  - A melhor variação é a inserção com incrementos decrescentes, também chamado de ordenação de Shell.

Sumário

Ordenação de Shell

- Tem como objetivo aumentar o passo de movimento dos elementos ao invés das posições adjacentes (passo = 1).
  - consiste em classificar sub-arquivos do original;
  - esses sub-arquivos contêm todo *k*-ésimo elemento do arquivo original;
  - o valor de *k* é chamado de *incremento*;
  - por exemplo, se *k* é 5, o sub-arquivo consistindo dos elementos $x[0]$, $x[5]$, $x[10]$, ... é classificado primeiro. Cinco sub-arquivos, cada um contendo um quinto dos elementos do arquivo original, são classificados dessa maneira.
  - São eles:
    sub-arquivo 1 -> $x[0]$ $x[5]$ $x[10]$ ...
    sub-arquivo 2 -> $x[1]$ $x[6]$ $x[11]$ ...
    sub-arquivo 3 -> $x[2]$ $x[7]$ $x[12]$ ...
    sub-arquivo 4 -> $x[3]$ $x[8]$ $x[13]$ ...
    sub-arquivo 5 -> $x[4]$ $x[9]$ $x[14]$ ...

Ordenação de Shell

- Após a ordenação dos sub-arquivos:
  - define-se um novo incremento menor que o anterior;
  - gera-se novos sub-arquivos;
  - e aplica-se novamente o método da inserção nesses novos sub-arquivos.
- E assim sucessivamente para novos valores de k, até que k seja igual a 1.
- Sequência de incrementos => definida previamente: h_1, h_2, ..., h_t, com as condições: $h_t = 1$ e $h_{i+1} < h_i$.

Ordenação de Shell

```
void shellsort(int x[], int n, int incrmnts[], int numinc)
{
 int incr, j, k, span, y;
 for (incr = 0; incr < numinc; incr++)
 {
   span = incrmnts[incr]; // span é o tamanho do incremento
   for(j = span; j < n; j++)
   { // insere elemento x[j] em sua posição correta dentro de seu sub-arquivo
     y = x[j];
     for (k = j-span; k >= 0 && y < x[k]; k -= span)
       x[k+span] = x[k];
     x[k+span] = y;
   }
 }
}
```

Ordenação de Shell

incremento	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
5 (0)	25	57	48	37	12	92	86	33
5 (1)	25	57	48	37	12	92	86	33
5 (2)	25	57	**33**	37	12	92	86	**48**
3 (0)	25	57	33	37	12	92	86	48
3 (1)	25	**12**	33	37	**57**	92	86	48
3 (2)	25	12	33	37	57	92	86	48
3 (3)	25	12	33	37	57	92	86	48
3 (4)	25	12	33	37	**48**	92	86	**57**
1 (0)	**12**	**25**	33	37	48	92	86	57
1 (1)	12	25	33	37	48	92	86	57
1 (2)	12	25	33	37	48	92	86	57
1 (3)	12	25	33	37	48	92	86	57
1 (4)	12	25	33	37	48	92	86	57
1 (5)	12	25	33	37	48	**86**	**92**	57
1 (6)	12	25	33	37	48	**57**	**86**	**92**

Eficiência da Ordenação de Shell

- Um problema com o *shell sort* ainda não resolvido é a escolha dos incrementos que fornece os melhores resultados.
- É desejável que ocorra o maior número possível de interações entre as diversas cadeias
- "Se a uma sequência, previamente ordenada, de distância k for em seguida aplicada uma ordenação de distância i, então esta sequência permanece ordenada de distância k."

Eficiência da Ordenação de Shell

- Knuth [1] mostra que uma escolha razoável de incrementos é a sequência (escrita em ordem inversa):

  $$1, 4, 13, 40, 121, ...$$

  onde $h_{k-1} = 3h_k + 1$, $h_t = 1$ e $t = (log_3 n) - 1$.
- Ele também recomenda a sequência

  $$1, 3, 7, 15, 31, ...$$

  onde $h_{k-1} = 2h_k + 1$, $h_t = 1$ e $t = (log_2 n) - 1$.
- Para esta última escolha, a análise matemática indica que o esforço computacional necessário à ordenação de n elementos através do *shell sort* é proporcional a $n^{1.2}$ [2].

Ordenação por Cálculo de Endereço

- Também conhecida por **espalhamento**.
- Nesse método uma função f é aplicada a cada elemento. O resultado dessa função determina em qual dos diversos sub-arquivos o registro será colocado.
- A função deverá ter a propriedade de que se $x \leq y$, $f(x) \leq f(y)$. Essa função é chamada de **preservadora da ordem**.
- Cada item é posicionado num sub-arquivo na sequência correta. Depois de todos os itens do arquivo original serem posicionados nos sub-arquivos, os sub-arquivos poderão ser concatenados para gerar o resultado classificado.

Ordenação por Cálculo de Endereço

- Considere novamente o arquivo exemplo abaixo:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- Cria-se 10 sub-arquivos, um para cada um dos dez primeiros dígitos (mais significativos) possíveis (0 a 9). Inicialmente, estes sub-arquivos estão vazios. Um vetor de ponteiros $f[10]$ é declarado, onde $f[i]$ aponta para o primeiro elemento no arquivo cujo primeiro dígito (mais significativo) seja i.
- Depois de rastrear o primeiro elemento (25), ele será posicionado no arquivo encabeçado por $f[2]$.
- Cada sub-arquivo será mantido como uma lista ligada classificada dos elementos do vetor original.

Ordenação por Cálculo de Endereço

- Depois de processar cada elemento do arquivo original, os sub-arquivos aparecerão como:

$$
\begin{array}{lcl}
f(0) & = & NULL \\
f(1) & \longrightarrow & \boxed{12 \mid NULL} \\
f(2) & \longrightarrow & \boxed{25 \mid NULL} \\
f(3) & \longrightarrow & \boxed{33 \mid \quad\quad} \longrightarrow \boxed{37 \mid NULL} \\
f(4) & \longrightarrow & \boxed{48 \mid NULL} \\
f(5) & \longrightarrow & \boxed{57 \mid NULL} \\
f(6) & = & NULL \\
f(7) & = & NULL \\
f(8) & \longrightarrow & \boxed{86 \mid NULL} \\
f(9) & \longrightarrow & \boxed{92 \mid NULL}
\end{array}
$$

Programa Cálculo de Endereço

```
void end(int x[], int n)
{
 PNO f[10], p;
 int prim, i, j, y;
 struct tipo_no no[NUMELTS];
 for (i = 0; i < n; i++) // Inicia lista ligada vazia
   no[i].info = 0;
 for (i = 0; i < n-1; i++)
   no[i].prox = no[i+1].prox;
 no[n-1].prox = NULL;
 for (i = 0; i < 10; i++)
   f[i] = NULL;
 for (i = 0; i < n; i++)
 {
   y = x[i];
   prim = y/10; // encontra o primeiro dígito do número de dois dígitos decimais
   coloca (&f[prim], y);
          // coloca insere y na posição correta na lista ligada apontada por f[prim]
 }
 i = 0;
 for (j = 0; j < 10; j++)
 {
   p = f[j];
   while (p != NULL)
   {
     x[i++] = p -> info;
     p = p -> prox;
   }
 }
}
```

Eficiência do Cálculo de Endereço

- As exigências de espaço da ordenação por cálculo de endereço são aproximadamente $2 * n$ (usado pelo vetor *no*) além de alguns nós de cabeçalho e variáveis temporárias.
- Para avaliar as exigências de tempo: se os n elementos originais forem uniformemente distribuídos pelos m sub-arquivos e o valor de n/m for aproximadamente 1, o tempo para a ordenação será praticamente $\mathcal{O}(n)$.
- Por outro lado, se n/m for muito maior que 1, ou se o arquivo original não for uniformemente distribuído pelos m sub-arquivos, será necessário um trabalho considerável.
- Desempenho: $\mathcal{O}(n^2)$

Sumário

Ordenação por Intercalação

- A intercalação *mergesort* é o método que combina dois ou mais arquivos classificados num terceiro arquivo classificado.
- Pode-se usar essa técnica para classificar um arquivo da seguinte maneira:
  - divida o arquivo em n sub-arquivos de tamanho 1;
  - intercale pares de sub-arquivos adjacentes.
- Tem-se então, aproximadamente $n/2$ sub-arquivos de tamanho 2.
- Repita o processo até restar apenas um arquivo de tamanho n.

Ordenação por Intercalação

- Considere mais uma vez o arquivo exemplo abaixo:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
25	57	48	37	12	92	86	33

- As passagens sucessivas da ordenação por intercalação ficam:

[25] [57] [48] [37] [12] [92] [86] [33]

[25 57] [37 48] [12 92] [33 86]

[25 37 48 57] [12 33 86 92]

[12 25 33 37 48 57 86 92]

Ordenação por Contagem de Menores [5]

- Idéia básica: se soubermos quantos são os elementos menores que um determinado valor, saberemos a posição que o mesmo deve ocupar no arranjo ordenado:
  - Por exemplo, se há 5 valores menores do que o elemento 7, o elemento 7 será inserido na sexta posição do arranjo.
- Usa-se um arranjo auxiliar para manter a contagem de menores e um outro para montar o arranjo ordenado.

Ordenação por Contagem de Menores

- Exemplo:

0	1	2	3	4	5	6	7	8	9
4	2	1	3	7	9	8	3	0	5

Arranjo original A desordenado

Ordenação por Contagem de Menores

- 1^o. Passo: criar arranjo auxiliar:

0	1	2	3	4	5	6	7	8	9
4	2	1	3	7	9	8	3	0	5

Arranjo original A desordenado

0	1	2	3	4	5	6	7	8	9

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

Ordenação por Contagem de Menores

- 1^o. Passo: criar arranjo auxiliar:

0	1	2	3	4	5	6	7	8	9
4	2	1	3	7	9	8	3	0	5

Arranjo original A desordenado

0	1	2	3	4	5	6	7	8	9
5	2	1	3	7	9	8	3	0	6

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

Ordenação por Contagem de Menores

- 2^o. Passo: montar arranjo final ordenado:

0	1	2	3	4	5	6	7	8	9
4	2	1	3	7	9	8	3	0	5

Arranjo original A desordenado

0	1	2	3	4	5	6	7	8	9
5	2	1	3	7	9	8	3	0	6

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

0	1	2	3	4	5	6	7	8	9

Arranjo final B ordenado: A[i] vai para a posição X[i] de B
→ Atenção com elemento 3 duplicado

Ordenação por Contagem de Menores

- 2^o. Passo: montar arranjo final ordenado:

0	1	2	3	4	5	6	7	8	9
4	2	1	3	7	9	8	3	0	5

Arranjo original A desordenado

0	1	2	3	4	5	6	7	8	9
5	2	1	3	7	9	8	3	0	6

Arranjo auxiliar X, em que X[i]=número de elementos no arranjo A que são menores que A[i] → indicam a posição correta de A[i] no arranjo ordenado

0	1	2	3	4	5	6	7	8	9
0	1	2	3	3	4	5	7	8	9

Arranjo final B ordenado: A[i] vai para a posição X[i] de B
→ Atenção com elemento 3 duplicado

Ordenação por Contagem de Menores

```
void contagem_de_menores(int A[], int n)
{
 int X[n], B[n], i, j;

 for (i = 0; i < n; i++) //inicializando arranjo auxiliar
   X[i] = 0;

 for (i = 1; i < n; i++) //contando menores
   for (j = i-1; j >= 0; j-)
     if (A[i] < A[j])
       X[j] += 1;
     else X[i] += 1;

 for (i = 0; i < n; i++) //montando arranjo final
   B[X[i]] = A[i];

 for (i = 0; i < n; i++) //copiando arranjo final para original
   A[i] = B[i];
}
```

Ordenação por Contagem de Menores

- Complexidade de tempo?
- Complexidade de espaço?

Ordenação por Contagem de Menores

- Complexidade de tempo: $\mathcal{O}(n^2)$
- Complexidade de espaço: $\mathcal{O}(3n)$

Sumário

Ordenação por Contagem de Tipos

- Também chamado *counting-sort*,
- Idéia básica: conta-se o número de vezes que cada elemento ocorre no arranjo; se há k elementos antes dele, ele será inserido na posição $k + 1$ do arranjo ordenado:
  - **Restrição**: os elementos devem estar contidos em um intervalo [min, max] do conjunto de números inteiros positivos.
- Usa-se um arranjo auxiliar para manter a contagem de tipos e um outro para montar o arranjo ordenado.

Ordenação por Contagem de Tipos

- Exemplo:

0	1	2	3	4	5	6	7	8	9
1	3	2	7	6	3	1	2	1	7

Arranjo original A desordenado
→ min=1, max=7

Ordenação por Contagem de Tipos

- Exemplo:

0	1	2	3	4	5	6	7	8	9
1	3	2	7	6	3	1	2	1	7

Arranjo original A desordenado
→ min=1, max=7

1	2	3	4	5	6	7

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A

Ordenação por Contagem de Tipos

- Exemplo:

0	1	2	3	4	5	6	7	8	9
1	3	2	7	6	3	1	2	1	7

Arranjo original A desordenado
→ min=1, max=7

1	2	3	4	5	6	7
3	2	2	0	0	1	2

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A
→ há 3 elementos 1, que ocuparão as posições 0, 1 e 2 do vetor ordenado
→ há 2 elementos 2, que ocuparão as posições livres seguintes (3 e 4) ...

Ordenação por Contagem de Tipos

- Exemplo:

0	1	2	3	4	5	6	7	8	9
1	3	2	7	6	3	1	2	1	7

Arranjo original A desordenado
→ min=1, max=7

1	2	3	4	5	6	7
3	2	2	0	0	1	2

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A
→ há 3 elementos 1, que ocuparão as posições 0, 1 e 2 do vetor ordenado
→ há 2 elementos 2, que ocuparão as posições livres seguintes (3 e 4) ...

0	1	2	3	4	5	6	7	8	9

Arranjo final B ordenado

Ordenação por Contagem de Tipos

- Exemplo:

0	1	2	3	4	5	6	7	8	9
1	3	2	7	6	3	1	2	1	7

Arranjo original A desordenado
→ min=1, max=7

1	2	3	4	5	6	7
3	2	2	0	0	1	2

Arranjo auxiliar X, em que X[i] indica o número de elementos i no vetor original A
→ há 3 elementos 1, que ocuparão as posições 0, 1 e 2 do vetor ordenado
→ há 2 elementos 2, que ocuparão as posições livres seguintes (3 e 4) ...

0	1	2	3	4	5	6	7	8	9
1	1	1	2	2	3	3	6	7	7

Arranjo final B ordenado

Ordenação por Contagem de Tipos

```
void countingsort(int A[], int n)
{
  int B[n], i, j, max;

  max=A[0]; //determinando max
  for (i=1; i<n; i++)
    if (A[i]>max)
      max=A[i];
  int X[max+1];

//inicializando arranjo auxiliar
  for (i=0; i<max+1; i++)
    X[i]=0;

//contando tipos
  for (i=0; i<n; i++)
    X[A[i]]++;
//montando arranjo final
  j=0;
  for (i=0; i<max+1; i++)
    while (X[i]!=0) {
      B[j]=i;
      j++;
      X[i]-;
    }

//copiando arranjo final para original
  for (i=0; i<n; i++)
    A[i]=B[i];
}
```

Ordenação por Contagem de Tipos

- Complexidade de tempo?
- Complexidade de espaço?

Ordenação por Contagem de Tipos

- Complexidade de tempo: $\mathcal{O}(n)$, se $max \leq n$:
    - Por que é "tão melhor" do que outros métodos?
- Complexidade de espaço: $\mathcal{O}(3n)$, se $max \leq n$.

Ordenação de Raízes (*Radix sort*) [4]

- Esta ordenação baseia-se nos valores dos dígitos nas representações posicionais dos números sendo ordenados.
- Executa as seguintes ações começando pelo dígito menos significativo e terminando com o mais significativo
  - Pegue cada número na sequência e posicione-o em uma das dez filas, dependendo do valor do dígito sendo processado.
  - Em seguida, restaure cada fila para a sequência original, começando pela fila de números com um dígito 0 e terminando com a fila de números com o dígito 9.
  - Quando essas ações tiverem sido executadas para cada dígito, a sequência estará ordenada.

Ordenação de Raízes (*Radix sort*)

- Exemplo: Lista original: 25 57 48 37 12 92 86 33
- Filas baseadas no dígito menos significativo

	início	final
fila[0]		
fila[1]		
fila[2]	12	92
fila[3]	33	
fila[4]		
fila[5]	25	
fila[6]	86	
fila[7]	57	37
fila[8]	48	
fila[9]		

Ordenação de Raízes (*Radix sort*)

- Depois da primeira passagem: 12 92 33 25 86 57 37 48
- Filas baseadas no dígito mais significativo

	início	final
fila[0]		
fila[1]	12	
fila[2]	25	
fila[3]	33	37
fila[4]	48	
fila[5]	57	
fila[6]		
fila[7]		
fila[8]	86	
fila[9]	92	

- Lista classificada: 12 25 33 37 48 57 86 92

Radix sort - Algoritmo

```
for (k = dígito menos significativo; k <= dígito mais
significativo; k++)
{
 for (i = 0; i < n; i++)
 {
   y = x[i];
   j = k-ésimo dígito de y;
   posiciona y no final da fila[j];
 }
 for (qu = 0; qu < 10; qu++)
   coloca elementos da fila[qu] na próxima posição sequencial;
}
```

Obs.: os dados originais são armazenados no array *x*.

Radix sort - Algoritmo

- Evidentemente, as exigências de tempo para o método de ordenação de raízes dependem da quantidade de dígitos (m) e do número de elementos na lista (n).
- Como a repetição mais externa é percorrida m vezes e a repetição mais interna é percorrida n vezes. Então, $T(n) = \mathcal{O}(m * n)$.
- Se o número de dígitos for menor, $T(n) = \mathcal{O}(n)$
- Se as chaves forem densas (isto é, se quase todo número que possa ser uma chave for de fato uma chave), m se aproximará de $log\ n$. Neste caso, $T(n) = \mathcal{O}(n\ log\ n)$.

Conclusão Geral

- Conclusão Geral: em termos de comparações e atribuições
- Pequenos arquivos: inserção simples
- Grandes arquivos: *quicksort* com elemento central como pivô

Bibliografia I

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Algoritmos - Teoria e Prática.
Ed. Campus, Rio de Janeiro, Segunda Edição, 2002.

[2] Engelbrecht, Angela
Estrutura e Recuperação de Informação II.
Apostila. Engenharia de Computação. PUC-Campinas, 2000.

[3] Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.

Bibliografia II

[4] Liang, Zhao
Aula 4 - Ordenação 2. Introdução à Ciência da Computação II - SCE0535.
Slides. Ciência de Computação. ICMC/USP, 2008.

[5] Pardo, Thiago A. S.
Análise de Algoritmos. SCE-181 Introdução à Ciência da Computação II.
Slides. Ciência de Computação. ICMC/USP, 2008.

[6] Tenenbaum, A. M., Langsam, Y., Augestein, M. J.
Estruturas de Dados Usando C.
Makron Books, 1995.

Bibliografia III

[7] Wirth, N.
Algoritmos e Estruturas de Dados.
LTC, 1989.

Referências I

[1] Knuth, D. E.
The Art of Computer Programming.
Volume 3. Reading, Mass.: Addison-Wesley, 1973.

[2] Shell, D. L.
A High-Speed Sorting Procedure.
Communications of the ACM, Volume 2, Issue 7 (July 1959), pp. 30–32.