Algoritmos e Estrutura de Dados I
Métodos de Ordenação

Michel Pires da Silva
michel@cefetmg.br

Departamento de Computação
DECOM-DV

Centro Federal de Educação Tecnológica de Minas Gerais
CEFET-MG

Sumário

Métodos de Ordenação

Ordenar: Processo aplicado a um conjunto de dados objetivando organiza-lo de forma ascendente ou descentente.

- Dicionários, Catálogo Telefônico, etc ...

A ordenação pode facilitar várias operações com conjuntos:

- Buscar um elemento em particular (busca binária, linear, etc.)
- Computar a quantidade de um certo objeto no conjunto ☺
- Ampliar a visão externa aos dados contídos no conjunto
- etc ...

Formalizando

Dado um conjunto de n elementos $E = \langle e_1, e_2, \ldots, e_n \rangle$, a ordenação busca produzir um novo conjunto E' na forma $\langle e'_1, e'_2, \ldots, e'_n \rangle$, tal que, $e'_1 \leq e'_2 \leq \cdots \leq e'_n$ ou $e'_1 \geq e'_2 \geq \cdots \geq e'_n$.

Métodos de Ordenação

O que ordenar?

- Na prática, os valores a serem ordenados raramente são valores isolados. Em geral, sempre se apresentam como **registros** cujo conteúdo possui, dentre seus itens, um valor único, comumente chamado de **chave**.

Exemplos ...

CPF, RG, RA ...

Métodos estáveis e não estáveis

- Um método de ordenação é classificado como estável se a ordem dos itens com **chaves iguais**, já ordenados, não é modificada durante o processo
- Um método de ordenação é dito não estável quando este **afeta a ordem relativa** do conjunto pré ordenado. Nesse contexto, se for o caso, é preciso forçar sua estabilidade.

Métodos de Ordenação

Tipos de Ordenação

- **Ordenação interna:** Aplicado quando é possível colocar o conjunto de dados por completo na memória principal.
- **Ordenação parcial / externa:** Aplicado quando o conjunto de dados a ser ordenado não pode ser armazenado por completo em memória devido a seu tamanho.

Diferença ...

Na ordenação interna, o objetivo é ter acesso direto a qualquer dado do conjunto, ao contrário, na externa ou parcial tem-se acesso somente a partes (blocos) ou linha a linha em forma sequencial.

Métodos de Ordenação

Observação: Um método de ordenação pode apresentar o princípio de ordenação por comparação ou distribuição

- **Comparação:** Método mais conhecido, em que um conjunto de dados é organizado utilizando-se *comparações de chaves.*
- **Distribuição:** Método que leva em consideração a existência de um *conjunto pré definido* de variáveis e que é de *conhecimento prévio os valores existentes.* Ex.: as 52 cartas de um baralho.

Como trabalha os métodos por distribuição ...

1. Distribuir as cartas em treze montes: Ases, dois, três, ..., reis.
2. Colete os montes na ordem específica
3. Distribua novamente as cartas em 4 montes: paus, ouros, copas e espadas.
4. Coloque os montes na ordem específica

Métodos de Ordenação

Métodos por distribuição, exemplos ...

Ordenação Digital, Radix Sort e Bucket Sort ...

Observação 1: O maior problema dos métodos de ordenação por distribuição é compor a regra que será utilizada para organizar os diferentes montes.

Observação 2: O custo computacional associado aos métodos de ordenação por distribuição para um conjunto de dados de n elementos é, geralmente, $O(n)$

Pergunta: Como um método de ordenação por distribuição poderia ser aplicado em uma sala de aula para classificar os alunos?

Métodos de Ordenação

Ordenação interna

- Quando torna-se necessário a utilização de um método de ordenação interna, atente-se ao custo computacional antes de iniciar sua implementação e aplicação no problema estudado.

Custos Associados

- Número de Comparações - C(n)
- Número de Movimentações - M(n)

Observação: A economia de memória interna é primordial nesse caso.

Atenção: Métodos que fazem uso de listas encadeadas não são muito utilizados para o trabalho e os que fazem cópias do conjunto não possuem importância.

Métodos de Ordenação

Categorização dos métodos de ordenação interna:

Métodos Simples

- Adequado para pequenos conjuntos de dados
- Em sua maioria apresentam custo perto de $O(n^2)$
- Produzem programas pequenos
- Em sua maioria, são de fácil implementação

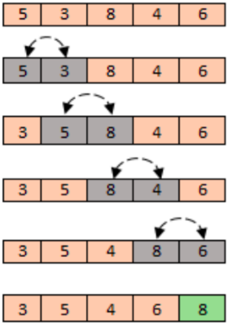
Métodos Eficientes

- Adequados para grandes conjuntos de dados
- Requerem, em sua maioria, $O(nlogn)$ comparações
- As comparações são mais complexas a nível de detalhes
- São mais complexos de implementar e, muitas vezes, fazem uso de recursão durante o processo de ordenação.

Método de Ordenação - Bubble Sort

A classificação por bolha é um método simples de ordenação, possível de ser associado a pequenas entradas de dados.

- É um dos algoritmos mais simples encontrados na literatura
- Seus passos de execução podem ser descritos como:
    1. Inicialize a execução no primeiro elemento, compare-o com os demais.
    2. Troque-o, caso o i-ésimo item for menor, reposicione o ponteiro e continue comparando até o fim.
    3. Repita os passos 1 e 2 com os $n - 1$ elementos, depois com $n - 2, \ldots,$ até que reste um único elemento.



Métodos de Ordenação - Bubble Sort

Algorithm 1: Bubble Sort: Maximum bubbles values first

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ the data vector
output: Ordered data vector

```
while n > 1 do
    Min = 1;
    for i = 2 to n do
        if D[i].value < D[Min].value then
            /* swap method */
            aux = D[Min];
            D[Min] = D[i];
            D[i] = aux;
            Min = i;
        end
    end
    n = n - 1;
end
```

Métodos de Ordenação - Selection Sort

- É um dos algoritmos mais simples encontrados na literatura
- Seus passos de execução podem ser descritos como:
  1. Selecione o menor item do vetor
  2. Troque-o com o item da primeira posição
  3. Repita as operações dos itens 1 e 2 com os $n-1$ elementos, depois com $n-2, \ldots,$ até que reste um único elemento

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
i = 1	**A**	*R*	*D*	*E*	*N*	**O**
i = 2	*A*	**D**	**R**	*E*	*N*	*O*
i = 3	*A*	*D*	**E**	**R**	*N*	*O*
i = 4	*A*	*D*	*E*	**N**	**R**	*O*
i = 5	*A*	*D*	*E*	*N*	**O**	**R**

Métodos de Ordenação - Selection Sort

Algorithm 2: Selection Sort

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ data vector
output: The ordered data vector

```
for i = 1 to n-1 do
    Min = i;
    for j = i + 1 to n do
        if D[j].value < D[Min].value then
            Min = j;
        end
    end
    /* swap method                                    */
    if i ≠ Min then
        aux = D[Min];
        D[Min] = D[i];
        D[i] := aux;
    end
end
```

Métodos de Ordenação - Selection Sort

Custo computacional do método:

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

$$M(n) = 3(n-1)$$

Observação

Segundo Knuth(1973, exercícios 5.2.3.3-6), o comando de atribuição $Min := j$ é executado aproximadamente $nlogn$ vezes.

Métodos de Ordenação - Selection Sort

Vantagens

- Apresenta custo linear no tamanho da entrada para o número de movimentos entre registros
- É um bom algoritmo para ser utilizado para conjuntos de dados que apresentam regitros muito grandes
- É muito interessante para arquivos pequenos

Desvantagens

- O fato de o conjunto de dados estar ordenado não ajuda em nada, pois o custo ainda está em $O(n^2)$
- O algoritmo **não é estável**

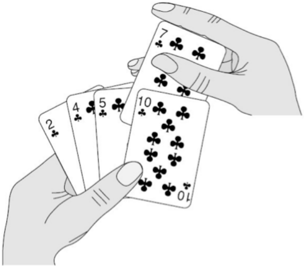
Métodos de Ordenação - Selection Sort

Implemente os métodos de ordenação Bubble Sort e Selection Sort e execute esses para os seguintes problemas.

- Considere um vetor linear de 15 posições, cada uma com um número inteiro aleatório. Quais seriam os passos do método para ordenar crescentemente esse vetor?
- Aproveite as exemplificações realizadas para discutir o que muda de um método para outro. Algum deles executa mais operações ou aparenta ser mais rápido visualmente? Por quê?
- Faça adaptações em ambos os métodos para receber uma lista simplesmente encadeada dinâmica, a qual deve ser ordenada utilizando-se apenas movimentação de ponteiros e não de conteúdo.

Métodos de Ordenação - Insertion Sort

- Algoritmo preferido dos jogadores de carta. A ideia básica está em ordena-las por distribuição



- Em cada passo, a partir de $i = 1$ selecione o i-esimo item de cada sequência e coloque-o no lugar apropriado de acordo com o critério de ordenação adotado.

	1	2	3	4	5	6
Chaves iniciais:	**O**	*R*	*D*	*E*	*N*	*A*
i = 2	**O**	**R**	*D*	*E*	*N*	*A*
i = 3	**D**	**O**	**R**	*E*	*N*	*A*
i = 4	**D**	**E**	**O**	**R**	*N*	*A*
i = 5	**D**	**E**	**N**	**O**	**R**	*A*
i = 6	**A**	**D**	**E**	**N**	**O**	**R**

Métodos de Ordenação - Insertion Sort

Algorithm 3: Insertion Sort

```
input  : n → number of items in data vector
input  : D → data vector
output: The ordered data vector

for i = 1 to n do
    aux := D[i];
    j := i - 1;

    while j ≥ 0 and aux.value < D[j].value do
        D[j+1] := D[j];
        j := j - 1;
    end

    D[j + 1] := aux;
end
```

Observação: Há implementações desse método que utilizam o conceito de sentinela para eliminar a necessidade de uma variável de auxílio (i.e., aux) durante a execução.

Métodos de Ordenação - Insertion Sort

Quando que o algoritmo Insertion Sort irá parar ?

- Um valor menor que o atribuido para **aux.value** é encontrado, então o valor de teste é inserido na sua frente na sequência do vetor
- Ou, a posição sentinela é alcançada.

Observação

Note que a posição sentinela é utilizada para facilitar as condições de parada e/ou como alternativa para a variável *aux*

Métodos de Ordenação - Insertion Sort

Custo Computacional / Análise Assintótica

$$\textbf{melhor caso: } C(n) = (1 + 1 + \cdots + 1) = n - 1$$

$$\textbf{pior caso: } C(n) = (2 + 3 + \cdots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$$

$$\textbf{caso médio: } C(n) = \frac{1}{2}(3 + 4 + \cdots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$$

Métodos de Ordenação - Insertion Sort

Custo Computacional / Análise Assintótica

melhor caso: $M(n) = (3 + 3 + \cdots + 3) = 3(n - 1)$

pior caso: $M(n) = (4 + 5 + \cdots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$

caso médio: $M(n) = \frac{1}{2}(5 + 6 + \cdots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$

Métodos de Ordenação - Insertion Sort

Observações Gerais

- O número mínimo de comparações e movimentações ocorre quando os valores estão ordenados de forma crescente
- O número máximo de comparações ocorre quando os valores estão originalmente em ordem reversa
- O *Insertion Sort* é um bom método para ser aplicado a arquivos quase ordenados
- O método é uma boa opção para a inserção de poucos registros em arquivos já ordenados, apresentando nesse caso custo linear no tamanho da entrada
- O algoritmo é **estável**

Métodos de Ordenação - Shell Sort

Contextualização:

- Método proposto por *Shell* em 1959
- Pode ser visto como uma extensão do método de inserção - *Insertion Sort*
- Problemas tratados se comparado com o método de Inserção:
    1. Facilita a troca de itens adjacentes, identificando o ponto de inserção em saltos maiores
    2. Evita as $n - 1$ comparações quando o menor item está na posição mais a direita no vetor

<u>Ideia:</u> A base do Shell Sort para contornar os problemas do método de inserção é permitir a troca de itens distantes no vetor

Métodos de Ordenação - Shell Sort

A ideia por trás do método

- Os itens em uma distância *h* são comparados e se necessário rearranjados.
- Todo *h-ésimo* item deve levar a uma sequência ordenada. Essa sequência pré-ordenada é dita *h-ordenada*

	1	2	3	4	5	6
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
h = 4	*N*	*A*	*D*	*E*	*O*	*R*
h = 2	*D*	*A*	*N*	*E*	*O*	*R*
h = 1	*A*	*D*	*E*	*N*	*O*	*R*

Métodos de Ordenação - Shell Sort

Como escolher bons valores para *h*

Knuth, em 1973, mostrou que uma boa sequência para esse modelo de método seria:

$$h(s) = 3h(s-1) + 1 \text{ para } s > 1$$

$$h(s) = 1 \qquad \text{para } s = 1$$

Métodos de Ordenação - Shell Sort

Algorithm 4: Shell Sort: Based on the Knuth theory for the h variable

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ data vector
output: The ordered data vector

```
h = 1;
do h = 3 * h + 1; while h < n;
do
    h = |h ÷ 3|;
    for i = h to n do
        aux = D[i];
        j = i;
        while D[j - h].value > aux.value do
            D[j] = D[j-h];
            j = j - h;
            if(j ≤ h or j ≤ 0){ break; }
        end
        D[j] := aux;
    end
while h ≠ 1;
```

Métodos de Ordenação - Shell Sort

- A implementação do Shell Sort **não** faz uso de posições sentinelas
- Seriam necessárias h posições sentinelas para que o algoritmo pudesse aplicar o mesmo conceito do Insertion Sort.
- Um outro ponto a considerar é que não se sabe ao certo qual é exatamente a eficiência desse algoritmo, pois, ninguém ainda foi capaz de analiza-lo.

Custo Computacional / Analise Assintótica

$$\text{Conjectura 1: } C(n) = O(n^{1.25})$$

$$\text{Conjectura 2: } C(n) = O(n(lnn)^2)$$

Métodos de Ordenação - Shell Sort

Vantagens

- O Shell Sort é um excelente método para arquivos de tamanho moderado
- Sua implementação é bem simples e requer uma quantidade de código relativamente pequena

Desvantagem

- O tempo de execução do algoritmo é sensível a ordem inicial do arquivo
- O método **não é estável**

Métodos de Ordenação - Shell Sort

Observações Gerais:

- Avaliamos até agora métodos que utilizam aninhamento de for e técnicas que utilizam saltos para resolver o problema da ordenação
- Os algoritmos apresentados até o momento estão definidos como sendo da ordem quadrática, ou próximos disso - $O(n^2)$
- Outra característica comum é que as implementações são de fácil codificação e não fazem uso de recursos complexos da linguagem.

Métodos de Ordenação - Merge Sort

Contextualização

- Criado em 1945 pelo matemático *John Von Neumann*, o algoritmo considera:
    1. Que um número menor de passos é utilizado para ordenar um subconjunto de dados
    2. Que é fácil criar uma lista ordenada a partir de duas já ordenadas

- Método que faz uso de *intercalações* e *chamadas recursivas*

- Apresenta como característica a divisão e conquista[1] como meio para ordenar o conjunto de dados

[1] Divide o problema em subproblemas para, só depois, resolvê-los

Métodos de Ordenação - Merge Sort

Passos executados pelo algoritmo:

1º: Verificar se não é um **caso base**, ou seja, os indicadores de início e fim do vetor apontam para a mesma posição. Isso nos diz que nosso conjunto só possui um elemento e que, neste caso, o vetor já se encontra ordenado

2º: Dividir a lista em duas sublistas de tamanho semelhante. Feito isso, chamar o algoritmo recursivamente até alcançar o caso base.

3º: Executar, para cada sublista obtida no 2º passo, uma ordenação e uma junção (mesclagem) com o procedimento *Merge*

Métodos de Ordenação - Merge Sort

Observação: O algoritmo é dividido em dois procedimento, *Merge Sort* e *Merge*

Algorithm 5: Merge Sort : Recursive start stage

input : *start* → start position in the data vector
input : *last* → last position in the data vector
input : *D* → data vector
output: The ordered data vector

```
if start < last then
    middle = (start + last) ÷ 2;
    MergeSort(D, start, middle);
    MergeSort(D, middle + 1, last);
    Merge(D, start, middle, last);
end
```

Algorithm 6: Merge procedure: The second stage used to merge data vector parts

input : *start* → start position in the data vector
input : *middle* → middle position in the data vector
input : *last* → last position in the data vector
input : *D* → data vector
output: The ordered data vector

```
i = start;
j = middle + 1;
k = 0;
while i ≤ middle and j ≤ last do
    if D[i] < D[j] then
        aux[k] = D[i];
        i = i + 1;
    end
    else
        aux[k] = D[j];
        j = j + 1;
    end
    k = k + 1;
end
/* Continua aqui ...                                    */
```

Métodos de Ordenação - Merge Sort

Algorithm 7: Merge procedure: The second stage used to merge data vector parts

```
/* Continuação ...                                    */
while i ≤ middle do
    aux[k] = D[i];
    k = k + 1;
    i = i + 1;
while j ≤ last do
    aux[k] = D[j];
    k = k + 1;
    j = j + 1;
for k = start to last do
    D[k] := aux[k − start];
/* Fim da continuação ...                             */
```

Métodos de Ordenação - Merge Sort

Custo computacional / Análise Assintótica

- Por se tratar de um método recursivo, o Merge Sort nos fornece como análise uma **equação de recorrência**. A mesma pode ser solucionada por meio do **teorema mestre**.

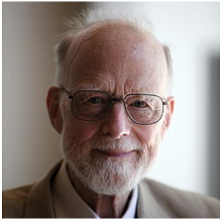
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

Observação

Resonvendo a equação de recorrência pelo teorema mestre tem-se que o método Merge Sort apresenta custo de $O(nlogn)$ pelo caso 2.

Métodos de Ordenação - Quick Sort

Contextualização



- Algoritmo proposto por Sir Charles Antony Richard Hoare em 1960 e publicado em 1962 após refinamentos
- O Quick Sort é o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações

A ideia ...

A ideia básica é dividir o problema de ordenar um conjunto de n itens em dois problemas menores.

Observação

O Quick Sort não é um algoritmo de divisão e conquista, mas sim, conquista e divisão

Métodos de Ordenação - Quick Sort

Contextualização

- A parte mais delicada do algoritmo é o processo de particionamento
- O vetor *D*, nesse algoritmo, é rearranjado a partir da escolha de um item "arbitrário" chamado de pivô

Observação

O vetor *D* é particionado em dois segmentos:

- O segmento esquerdo com chaves menores ou iguais ao pivô
- O segmento direito com chaves maiores ou iguais ao pivô

Métodos de Ordenação - Quick Sort

O processo de ordenação

1. Escolha arbitrariamente um **pivô**
2. Percorra o vetor a partir da esquerda até que o item na posição *D[i]* seja maior ou igual ao **pivô**
3. Percorra o vetor a partir da direita até que o item na posição *D[j]* seja menor ou igual ao **pivô**
4. Troque as posições *D[i]* e *D[j]* encontradas nos itens 2 e 3
5. Continue o processo até os apontadores **i** e **j** se cruzarem

Métodos de Ordenação - Quick Sort

A forma de execução

- O **pivô** é escolhido, no modelo padrão, como sendo o *D[(i + j) ÷ 2]*
- No exemplo ao lado, como i = 1 e j = 6, então o pivô será o elemento da posição 3 - Elemento D
- Ao termino do processo de partição, **i** e **j** se cruzam

1	2	3	4	5	6
O	R	**D**	E	N	A
A	R	**D**	E	N	O
A	**D**	R	E	N	O

Métodos de Ordenação - Quick Sort

Algorithm 8: Quick Sort: partition stage

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ data vector
input : *left* $\rightarrow$ left vector pointer
input : *right* $\rightarrow$ right vector pointer
output: The ordered data vector

```
i = esq;    j = dir;
pivo = D[(i+j) ÷ 2]; // Escolha do pivô
do
    while pivo.value > D[i].value do
        i = i + 1;
    end
    while pivo.value < D[j].value do
        j = j - 1;
    end
    if i <= j then
        aux = D[i];
        D[i] = D[j];
        D[j] = aux;
        i = i + 1;    j = j - 1;
    end
while i > j;
```

Métodos de Ordenação - Quick Sort

Algorithm 9: Quick Sort: start stage

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ data vector
input : *left* $\rightarrow$ left vector pointer
input : *right* $\rightarrow$ right vector pointer
output: The ordered data vector

```
Partition(left, right, i, j);
if left < j then
    QuickSort(left, j); // i e j ocultados 3⁰ e 4⁰ parâmetros
end
if i < right then
    QuickSort(i, right); // i e j ocultados 3⁰ e 4⁰ parâmetros
end
```

Métodos de Ordenação - Quick Sort

Curso computacional / Análise assintótica

- O Quick Sort, em média, apresenta comportamento semelhante ao Merge Sort. Logo, apresenta custo de $O(nlogn)$

Pior Caso

O pior caso ocorre quando sistematicamente o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado (o maior elemento do conjunto). Nesse caso o algoritmo apresenta custo quadratico $O(n^2)$

Observação

O pior caso pode ser evitado empregando pequenas modificações no algoritmo. Para isso basta escolher três itens quaisquer do vertor e usar a **mediana dos três** como pivô. Essa solução é chamada de <u>mediana de três</u>

Métodos de Ordenação - Quick Sort

Vantagens

- É extremante eficiente para ordenar arquivos de dados
- Necessita de apenas uma pequena pilha como memória auxiliar
- Requer cerca de *nlogn* comparações em média para ordenar n itens

Desvantagens

- Tem um pior caso $O(n^2)$ quando o pivô é escolhido nos extremos de um arquivo já ordenado
- Sua implementação é muito delicada e difícil. Um pequeno engano pode levar a efeitos inesperados para algumas entradas em particular
- o método **não é estável**

Métodos de Ordenação - Heap Sort

Contextualização

- Apresenta o mesmo princípio de funcionamento que o método de seleção

- No contexto do algoritmo de seleção o custo para encontrar o menor ou maior elemento é de $n-1$ comparações.

- O tempo de execução do Heap Sort pode ser melhorado utilizando fila de prioridade.

Fila de prioridade

Conceito visto com frequência em sistemas operacionais:

- Para identificar o tempo que cada evento deve ocorrer.
- Para identificar em memória quando uma página deve ser substituida
- Para gerenciar processos

Métodos de Ordenação - Heap Sort

Utilizando Lista de Prioridades

A representação da TAD pode ser feita por meio de uma lista linear ordenada.

- Custo para construir é $O(nlogn)$
- Custo para inserir é $O(n)$
- Custo para retirar é $O(1)$
- Custo para unir é $O(n)$

A representação da TAD pode ser feita por meio de uma lista linear não ordenada

- Custo para construir é $O(n)$
- Custo para inserir é $O(1)$
- Custo para retirar é $O(n)$
- Custo para unir por apontador $O(1)$ e por arranjo $O(n)$

Métodos de Ordenação - Heap Sort

Observação: A melhor forma de se representar a estrutura é utilizando-se um *Heap*

- Custo para construir é $O(n)$
- Custo para inserir é $O(nlogn)$
- Custo para retirar é $O(nlogn)$
- Custo para unir $O(nlogn)$

Observação

Formas mais eficiêntes podem utilizar alguns TADs sofisticados, como: Árvores Binomiais. Para mais detalhes consulte `https://www.ic.unicamp.br/~meidanis/courses/mo417/2003s1/aulas/2003-04-23.html`

Métodos de Ordenação - Heap Sort

O que são heaps ?

- É uma sequência de itens com chaves $c[1], c[2], \ldots, c[n]$, tal que:

$$c[i] \geq c[2i]$$
$$c[i] \geq c[2i+1]$$

- Para todo $i = 1, 2, \ldots, \frac{n}{2}$
- Essa definição deixa a estrutura similar a de uma árvore binária completa.

Métodos de Ordenação - Heap Sort

Observações:

- Para que as chaves satisfaçam a condição do *Heap*, a chave de um nó pai deve ser maior que a chave aplicada à seus filhos.

- A chave do nó raiz, primeira posição do *Heap*, é a maior chave contida no conjunto

1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*

- Filhos de i são: $2i$ e $2i + 1$
- Pai de i é i div 2

Observação

Um bom algoritmo para implementação de *heaps* foi apresentado em 1964 por Floyd

Métodos de Ordenação - Heap Sort

Processo de execução do *Heap Sort*

	1	2	3	4	5	6	7
Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*	*S*
Esq = 3	*O*	*R*	**S**	*E*	*N*	*A*	**D**
Esq = 2	*O*	*R*	*S*	*E*	*N*	*A*	*D*
Esq = 1	**S**	*R*	**O**	*E*	*N*	*A*	*D*

Métodos de Ordenação - Heap Sort

Algorithm 10: Heap Sort: Método principal, HeapSort(Dados, n)

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ the data vector
output: Ordered data vector

```
left = 0;    right = n;
while right > 1 do
    right = right - 1;
    BuildHeap(Dados, right);
    aux := Dados[0];
    Dados[0] := Dados[right];
    Dados[right] := aux;
end
/* Método 2x mais lento que o Quick Sort */
```

Métodos de Ordenação - Heap Sort

Procedimento que se encarrega de criar a estrutura de *heap*

Algorithm 11: Heap Sort: Método auxiliar, BuildHeap(Dados, right)

input : *right* → right position in the data vector
input : *D* → the data vector
output: Ordered data vector

```
left := right div 2;
while left > 0 do
    left = left - 1;
    RebuildHeap(Dados, left);
end
```

Algorithm 12: Heap Sort: Método auxiliar, RebuildHeap(Dados, left)

input : $n \rightarrow$ number of items in data vector
input : *left* $\rightarrow$ left position in the data vector
input : $D \rightarrow$ the data vector
output: Ordered data vector

```
child = 2 * left; // Em C, left pode ser 0
if child+1 <= n and Dados[child] < Dados[child+1] then
    child = child + 1;
end
if Dados[left] < Dados[child] then
    aux = Dados[left];
    Dados[left] := Dados[child];
    Dados[child] := aux;
end
```

Métodos de Ordenação - Heap Sort

Métodos adicionais para tratamento da estrutura de dados *Heap*

Algorithm 13: Heap Sort: Método auxiliar, RemoveMaxValue(Dados, n)

input : $n \rightarrow$ number of items in data vector
input : $D \rightarrow$ the data vector
output: The maximum item in D (removeMax)

```
if n < 0 then
    writeln('Erro : Heap vazio');
end
else
    removeMax = Dados[0];
    Dados[0] = Dados[n];
    n = n - 1;
    BuildHeap(Dados, n);
end
```

Métodos de Ordenação - Heap Sort

Vantagens:

- O comportamento do *Heap Sort* é sempre $O(nlogn)$, qualquer que seja a entrada.
- Não utiliza memória adicional já que não faz uso de recursão

Desvantagens:

- O anel interno do algoritmo é bastante complexo se comparado com o Quick Sort
- O *Heap Sort* **não é estável**

Recomentado para ?

- Aplicações que não podem tolerar um caso desfavorável e/ou exigem precaução com o gasto de memória com chamadas recursivas
- Não é recomendado para arquivos com poucos registros devido o tempo gasto na construção do *Heap*

Métodos de Ordenação - Radix Sort

Abordagem alternativa para ordenação que processa as chaves por partes

- Por exemplo, comparação realizada em um nome a partir das letras 1, 2, 3, . . . , n.

Ideia

Quebrar a chave em vários pedaços

- 312 tem os dígitos 3, 1 e 2 na base 10
- 312 tem os dígitos 100111000 na base 2
- "exemplo"tem 6 caracteres na base 256

Observação: A idéia a partir da base é ordenar por meio do valor do número mais a esquerda do conjunto.

Métodos de Ordenação - Radix Sort

Vejamos um exemplo na base 10. Nessa base teremos dígitos de 0 a 9 para processar.

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**

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

Métodos de Ordenação - Radix Sort

1º Passo: Contabilizar quantos elementos existem de cada valor

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**

Digito	Contador
0	0
1	0
2	**2**
3	**3**
4	**1**
5	0
6	0
7	**1**
8	0
9	0

Métodos de Ordenação - Radix Sort

2º Passo: Calcular a posição de cada conjunto contabilizado no vetor

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**

Dig	C	Posicao
0	0	0
1	0	0
2	2	0
3	3	2
4	1	5
5	0	0
6	0	0
7	1	6
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

123	142	087	263	233	014	132
		123				

Dig	C	Posicao
0	0	0
1	0	0
2	2	0
3	3	3
4	1	5
5	0	0
6	0	0
7	1	6
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**
14**2**		12**3**				

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	3
4	1	5
5	0	0
6	0	0
7	1	6
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

123	142	087	263	233	014	132
142		123				087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	3
4	1	5
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

123	142	087	263	233	014	132
142		123	263			087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	4
4	1	5
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

12**3**	14**2**	08**7**	26**3**	23**3**	01**4**	13**2**
14**2**		12**3**	26**3**	23**3**		08**7**

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	5
4	1	5
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

123	142	087	263	233	014	132
142		123	263	233	014	087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	5
4	1	6
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

3º Passo: Inserir cada conjunto nos seus respectivos locais

123	142	087	263	233	014	132
142	132	123	263	233	014	087

Dig	C	Posicao
0	0	0
1	0	0
2	2	1
3	3	5
4	1	6
5	0	0
6	0	0
7	1	7
8	0	0
9	0	0

Métodos de Ordenação - Radix Sort

Repetimos o mesmo processo para os próximos dígitos como no exemplo abaixo:

123	142	087	263	233	014	132
142	132	123	263	233	014	087
014	123	132	233	142	263	087
014	123	132	233	142	263	087
014	087	123	132	142	233	263

Observação

Observe que ao terminar a execução os números se encontram exatamente na posição correta em ordem crescente de valores.

Métodos de Ordenação - Radix Sort

input : *n* → number of items in data vector
input : *base* → indicator for the worked base
input : *ndigits* → maximum number of digits
input : *D* → the data vector
output: The maximum item in D (removeMax)

```
for position = 0 to ndigits do
    for j = 0 to base do count[j] = 0;
    for i = 0 to n do
        d = getDigits(Dados[i], position, base);
        count[d+1] := count[d+1] + 1;
    end
    for j = 1 to base do count[j] := count[j] + count[j-1];
    for i:=0 to n do
        d = getDigits(Dados[i], position, base);
        idx := count[d];
        count[d] = count[d] + 1;
        aux[idx] := Dados[i];
    end
    for i := 0 to n do   Dados[i] := aux[i];
end
```

Métodos de Ordenação - Radix Sort

Custo computacional / Análise do Algoritmo

- Observe que não houve nenhuma comparação ao decorrer das execuções
- O custo para verificações de dígitos é de $2 * n * ndigits$
- O custo para trocas é de $n * ndigits$
- Se *ndigits* for pequeno ou constante, então o radix Sort apresenta custo linear $O(n)$

Observação

- O quick sort é compatível com o radix sort porque o número de dígitos é da ordem de $log(n)$
- Algumas literaturas dizem que o número máximo de dígitos a serem considerados no modelo é 10. Valores acima disso geram depreciação do método

Métodos de Ordenação - Radix Sort

Vantagens

- O algoritmo é estável
- Não compara chaves para ordenar o conjunto de dados

Desvantagens

- Nem sempre é fácil otimizar a inspeção de dígitos, esse processo depende muito do *hardware*
- Só é bom se o número de dígitos for pequeno
  - Em geral o número de dígitos tem crescimento a um fator de $log(n)$

Observações Gerais

Apresentamos duas classes de algoritmos para ordenação em memória primária, os simples com custo $O(n^2)$ e os mais elaborados com custo $O(nlogn)$. Veja a divisão na tabela abaixo:

Métodos	**Complexidade**
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shell Sort	$O(n^{1.25})$
Merge Sort	$O(nlogn)$
Quick Sort	$O(nlogn)$
Heap Sort	$O(nlogn)$

Observações Gerais

- O **Quick Sort**, geralmente, se apresenta como melhor solução (tempo de execução) para a maioria dos casos avaliados e reais.
- O **Heap Sort** e **Quick Sort** mantêm a mesma relação de tempo para arquivos de mesmo tamanho
- Para arquivos pequenos, o **Shell Sort** apresenta melhor desempenho que o **Heap Sort**
- O **Inserção** é o mais rápido para arquivos que apresentam como característica um conjunto de dados já ordenado. Ele também se apresenta como melhor opção dentre os algoritmos de custo quadrático - $O(n^2)$

Observações Gerais

A influência do conjunto de dados

- O **Shell Sort** e o **Quick Sort** são bastante sensíveis à ordem ascendente ou descendente da entrada.
- O **Shell Sort** executa mais rápido para arquivos já ordenados
- O **Heap Sort** não é sensível a ordem ascendente ou descendente da entrada e também não faz uso excessívo de pilha recursiva.

PERGUNTAS?