Universidade Cruzeiro do Sul

Estruturas de Dados

Conteúdo

- Recursividade (recursão).
- Métodos recursivos.
- Método de ordenação Quick Sort.
- Método de ordenação Merge Sort.

Elaboração

Prof. Manuel F. Paradela Ledón

Recursividade (recursão)

Em ciência da computação, a **recursividade** ou **recursão*** é a definição de uma rotina ou método (seja uma função ou um procedimento) que pode invocar (chamar) a si mesmo.

Uma definição recursiva é definida em termos de si própria.

Em um método recursivo deverá existir:

- Ao menos uma parte básica não-recursiva, cujo resultado é imediatamente conhecido (e que garante o fim da recursividade, ou seja, que o processo seja finito).
- Ao menos uma parte recursiva em que se resolve um subproblema do problema inicial invocando (chamando) o próprio método.

*ambos os termos existem nos dicionários Aurélio e Houaiss, mas com definições algo diferentes do significado esperado na programação

Recursividade (recursão)

Podemos construir procedimentos ou funções recursivas. Veja dois exemplos, na linguagem Java, um procedimento e uma função que efetuam uma chamada a si próprio:

```
//Exemplos em Java:

public void procedimento(...) {
    ...
    procedimento (...);   // o procedimento se autoexecuta
}

public float funcao(...) {
    ...
    float b = funcao(...)  +  ... ;  // a função se autoexecuta

}
```

Recursividade (recursão)

Muitas linguagens de programação permitem usar recursividade (Java, C++, C#, Python, Pascal, Prolog, LISP etc.)

```
def contagem (n):
if n == 0:
    print "Acabou!"
else:
   print n
   contagem (n-1)
```

Python

```
% PROLOG

%Regra 1 - não recursiva
antecessor(X,Z) :- progenitor(X,Z).

%Regra 2 – regra recursiva
antecessor(X,Z) :-  progenitor(X,Y),
antecessor(Y,Z).
```

Exemplo já conhecido: a sucessão de Fibonacci (ou sequência de Fibonacci)

A **sequência de Fibonacci** é um problema clássico da Matemática, frequentemente formulado em forma ***recursiva***: trata-se de uma sequência de números naturais, na qual os primeiros dois números são 0 e 1, e cada número subsequente será a soma dos dois números precedentes na sequência de Fibonacci (Leonardo de Pisa, Fibonacci, Pisa, Itália, séc. XIII).

Dependendo da abordagem para resolução deste problema ele poderá ser resolvido:

- em tempo exponencial de **$O(2^n)$** na abordagem **recursiva**;
- em tempo polinomial de O(n) em uma abordagem *iterativa*;
- ou em tempo logarítmico de complexidade computacional O(log n) utilizando uma estratégia de divisão e conquista.

Mencionam-se aplicações da sequência de Fibonacci em problemas da matemática/geometria (triângulo de Pascal, máximo divisor comum, conversão km/milhas), na arquitetura, análise de flutuações do mercado, na natureza (arranjo de folhas, pétalas, espiral de conchas) etc.

A sequência de Fibonacci em forma recursiva

Universidade Cruzeiro do Sul

A **sequência de Fibonacci** aparece originalmente formulada em forma **recursiva**, então mostraremos esta abordagem para exemplificar o conceito da recursão. No exemplo, observe que um número de Fibonacci será calculado somando os dois números anteriores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

A sequência pode ser definida **recursivamente**. Para calcular o n-ésimo número de Fibonacci, podemos definir, em forma geral, a seguinte função:

$$\textbf{fibo}(n) = \begin{cases} 0 & \text{se } n=0 \\ 1 & \text{se } n=1 \\ \textbf{fibo}(n-1) + \textbf{fibo}(n-2) & \text{se } n>1 \end{cases}$$

por definição os dois primeiros números da sequência são 0 e 1

pré-requisito: n >= 0

Em uma análise assintótica, quando $n \rightarrow \infty$, a complexidade deste algoritmo recursivo é **$O(2^n)$**, ou seja, a solução terá tempo exponencial (depende da variável **n** como expoente). É um algoritmo muito crítico quanto a tempo de execução!

A complexidade de espaço, por causa da pilha de execução necessária para recursividade, também deverá ser considerada.

Por exemplo, o quinto número da sequência de Fibonacci será calculado:

fibo(5) = fibo(4) + fibo(3)

fibo(3) + fibo(2) fibo(2) + fibo(1)

fibo(2) + fibo(1) fibo(1) + fibo(0) fibo(1) + fibo(0) 1

1 1 0 1 0

fibo(1)+fibo(0)

1 0

Resumindo, vemos que: fibo(5) vale 5,
resultado de várias chamadas recursivas.

```
System.out.println( "fibo(5): " + fibo(5) );
System.out.println( "fibo(7): " + fibo(7) );
System.out.println( "fibo(8): " + fibo(8) );
System.out.println( "fibo(9): " + fibo(9) );
// para valores grandes de n este método demora muito...

public long  fibo ( int n ) {
   if ( n < 0 ) return -1;  // não definido para valores negativos
   else if (n == 0) return 0;
   else if (n == 1) return 1;
   else return ( fibo(n-1) + fibo(n-2) );
}
```

Veja e teste o exemplo fornecido como projeto NetBeans.

Exemplo: fatorial de um número natural n

O fatorial de um número natural pode ser definido em forma mais clara **recursivamente**. Para calcular o fatorial de um número n, podemos definir, em forma geral, a seguinte função:

$$\textbf{fatorial}\ (n) = \begin{cases} 1 & \text{se } n=0 \text{ ou } n=1 \\ n * \textbf{fatorial}\ (n-1) & \text{se } n>1 \end{cases}$$

- pré-requisito: n >= 0
- fatorial(0) e fatorial(1) valem 1
- a complexidade de **tempo** do algoritmo do fatorial, em forma recursiva, é **O(n)**
- a complexidade de **espaço** também é **O(n)**, por causa da pilha de execução necessária para a recursividade
- o algoritmo para calcular o fatorial em forma não recursiva (iterativa) será melhor, com complexidade de espaço de apenas O(1)

Como o compilador resolve 5! (fatorial de 5) ?

```
0! = 1
1! = 1
3! = 3 x 2 x 1
4! = 4 x 3 x 2 x 1
5! = 5 x 4 x 3 x 2 x 1
5! = 5 x 4!

fat(5)  =  5 x  fat(4)
                4  x fat(3)
                     3  x  fat(2)
                           2 x fat(1)
                               1
```

```
O compilador pegará os valores da pilha de execução (stack)
em sentido contrário: 1 x 2 x 3 x 4 x 5 = 120
```

```
System.out.println( "fatorial(4): " + fatorial (4) );
System.out.println( "fatorial(5): " + fatorial (5) );


public long fatorial ( int n ) {
	if ( n < 0 ) return -1;  // não existe fatorial para valores negativos
	else if (n == 0 || n == 1) return 1;
	else return ( n * fatorial (n-1) );
}
```

Veja este exemplo em Fatorial.java

O algoritmo de Busca Binária em forma recursiva

Veja este exemplo no projeto NetBeans BuscaBinariaRecursiva

```
public class BuscaBinariaRecursiva {

   public static void main(String[] args) {
      new BuscaBinariaRecursiva();
   }

   public BuscaBinariaRecursiva() {
      double vetex[] = {-30, 1, 6, 8, 9, 12, 34, 41, 67, 78, 92};
      System.out.println("8 encontrado na posição " + busca(vetex, 8));
      System.out.println("34 encontrado na posição " + busca(vetex, 34));
      System.out.println("15 encontrado na posição " + busca(vetex, 15));
   }

   public int busca(double vetor[], double chave) {
      return buscaBinaria(vetor, 0, vetor.length - 1, chave);
   }

   private int buscaBinaria(double vetor[], int inf, int sup, double chave) {  // função recursiva
       if (inf > sup) return -1;  //não foi encontrado o valor buscado (chave)
       int centro = (inf + sup) / 2;
       if (chave == vetor[centro]) {
           return centro;  //encontramos o valor procurado (chave) na posição centro ➔ fim recursivo
       } else if (chave < vetor[centro]) {
           return buscaBinaria(vetor, inf, centro - 1, chave);  //buscamos a chave no trecho inferior
       } else {
           return buscaBinaria(vetor, centro + 1, sup, chave);  //buscamos a chave no trecho superior
       }
   }
}
```

Outro exemplo de recursividade: encontrar o **maior valor de um vetor**

```java
public class Maximo {

    public static void main(String args[]) {
        new Maximo();
    }

    public Maximo() {
        int a[] = {12, 21, 89, 99, 45, 89, 12, 24, 6, 70, 12, 56, 78};
        System.out.println( "Maior dos valores: " + maximo(a, 0, a.length-1 ) );
    }

    public int maximo (int vet[], int inicio, int fim) {
            if(inicio == fim) return vet[inicio];
            int meio = (inicio + fim) / 2;
            int a = maximo(vet, inicio, meio);
            int b = maximo(vet, meio+1, fim);
            return ( (a>b) ? a:b );
     }

}
```

Veja este exemplo em Maximo.java ou no projeto NetBeans Maximo.

Métodos de ordenação Quick Sort e Merge Sort

Neste material estudaremos os métodos Quick Sort e Merge Sort.

- Ambos são métodos naturalmente **recursivos**.
- Ambos utilizam alguma estratégia de **divisão e conquista**. O que isto significa?

Curiosidade: René Descartes

René Descartes, filósofo, físico e matemático francês, nasceu em La Haye en Touraine em 31 de março de 1596 e morreu em Estocolmo em 11 de fevereiro de 1650.

O Discurso do Método (também chamado *O Discurso sobre o Método*) pode ser considerada uma obra importante dentro da Resolução de Problemas.
Veja em (PADOVANI, 1972, pp. 289-295).

Curiosidade: René Descartes - *O Discurso do Método*

O método de raciocínio proposto por Descartes no **Discurso do Método** está composto por quatro partes:

- Receber escrupulosamente as informações, examinando sua racionalidade e sua justificação. Verificar a verdade, a boa procedência daquilo que se investiga. Aceitar somente o que não tenha dúvidas.
- **Análise**, ou divisão do assunto em tantas partes quanto possível e necessário.
- **Síntese**, ou elaboração progressiva de conclusões abrangentes e ordenadas a partir de objetos mais simples e fáceis até os mais complexos e difíceis.
- Enumerar e revisar minuciosamente as conclusões, garantindo que nada seja omitido e que a coerência geral exista.

Análise e síntese (HOUAISS, 2009)

- **Análise**

separação de um todo em seus elementos ou partes componentes

estudo pormenorizado de cada parte de um todo, para conhecer melhor sua natureza, suas funções, relações, causas etc.

- **Síntese**

método, processo ou operação que consiste em reunir elementos diferentes, concretos ou abstratos, e fundi-los num todo coerente

método cognitivo que, partindo da evidência imediata dos fragmentos de um objeto, alcança uma formulação teórica de sua totalidade, indo da constatação de elementos simples à explicação de combinações complexas

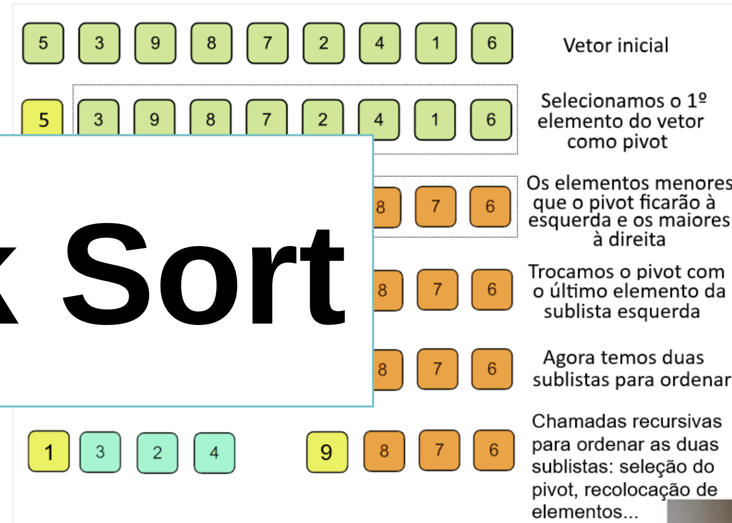
Divisão e conquista

A solução de um problema (algoritmo etc.) poderá utilizar o paradigma da **divisão e conquista**. Esse paradigma (ou estratégia para a resolução de problemas) consiste no seguinte:

- o problema é dividido em dois ou mais problemas menores;
- cada parte menor (ou subproblema) será resolvida usando normalmente o mesmo método de solução que estava sendo utilizado;
- as soluções das instâncias menores são combinadas para produzir uma solução do problema original.

Os métodos de ordenação Quick Sort e Merge Sort utilizam as ideias desta abordagem.

Quick Sort

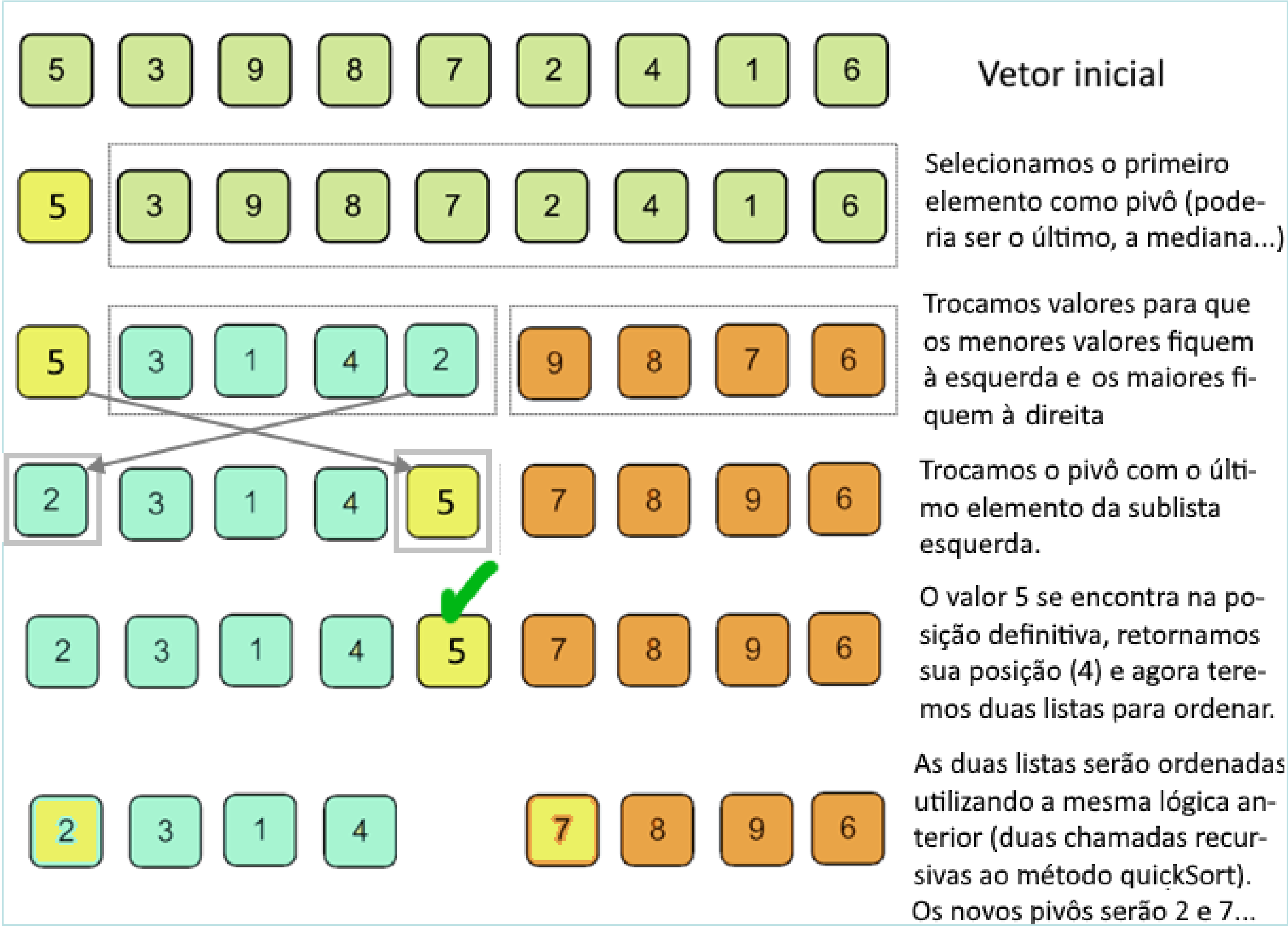


Quicksort, 1960
Charles Antony Richard (**Tony**) **Hoare**
britânico, 1934-...

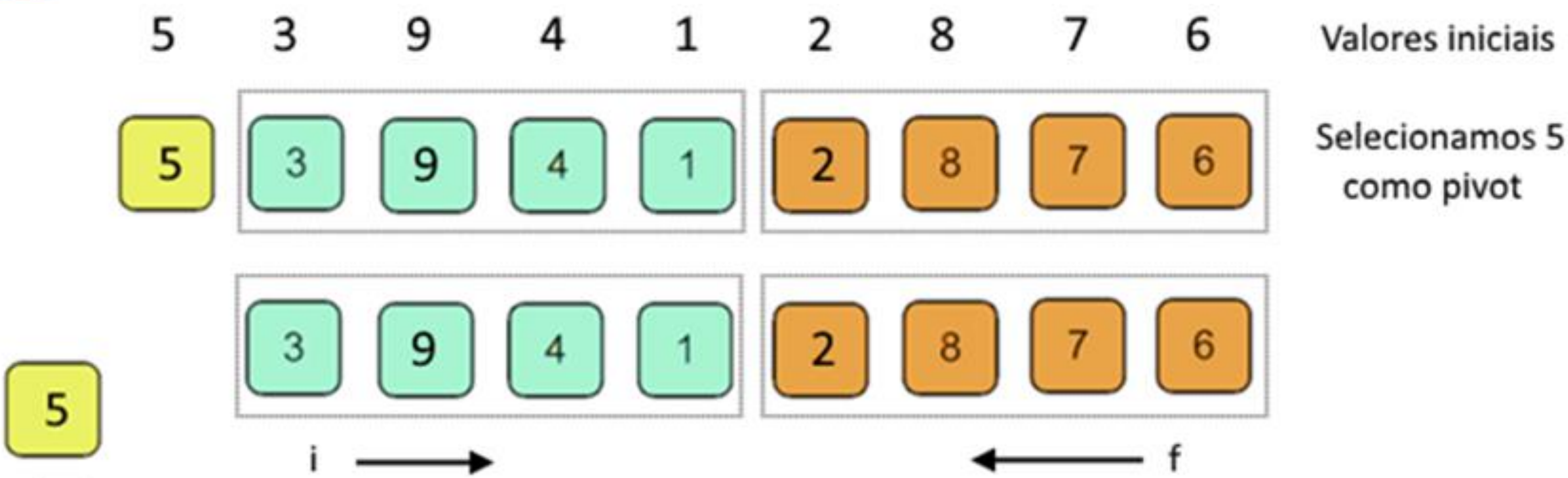
Existem muitas versões e adaptações do método Quick Sort de Tony Hoare. Vamos apresentar aqui uma implementação bem próxima do algoritmo original de Hoare.

Método de ordenação Quick Sort

O método Quick Sort ordena uma lista de valores escolhendo um pivot (pivô), trocando valores e ordenando as sublistas esquerda e direita utilizando o próprio método.

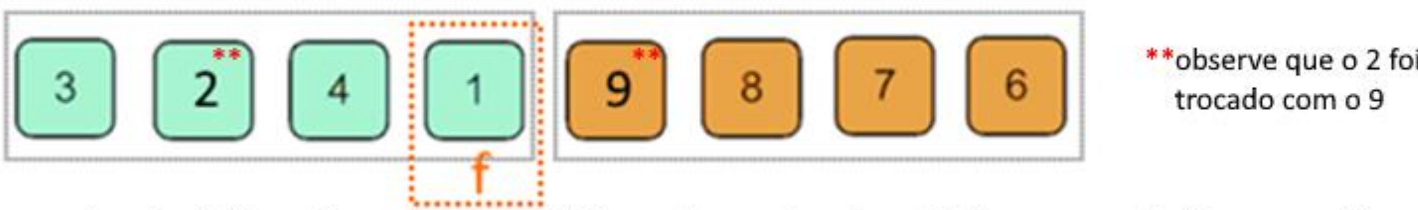


Outro exemplo com mais detalhes



pivot

Avançamos o ponteiro i e retrocedemos o ponteiro f até encontrar um elemento vetor[i] que deveria ficar à direita (por ser maior que o pivô) e encontrar um elemento vetor[f] que deveria estar à esquerda (por ser menor ou igual que o pivô): nesse momento efetuaremos a troca de vetor[i] com vetor[f].** O ciclo continua: f-- e i++ enquanto i <= f.



efetuamos a troca do pivot (5 na figura) com o último elemento da sublista esquerda(1, que está em f):

Ao concluir esta etapa temos o valor 5 onde deverá ficar, os valores menores do lado esquerdo, os valores maiores do lado direito e temos duas novas sublistas para serem ordenadas.

Método de ordenação Quick Sort (parte 1)

```
public boolean quickSort (double[] vetor, int inicio, int fim) {
    if (vetor == null) return false;
    if (inicio < fim) {
      int posicaoPivo = separar (vetor, inicio, fim);
      quickSort(vetor, inicio, posicaoPivo - 1);
      quickSort(vetor, posicaoPivo + 1, fim);
    }
    return true;
}
```

O método **separar** move os elementos menores que o pivô para esquerda e os maiores para direita. Este método retornará a nova posição do pivô (o lugar final, definitivo, onde ficou o valor que foi usado antes como pivô).

Depois de **separar** os elementos, os trechos à esquerda e à direita do pivô serão ordenados utilizando o próprio método **quickSort** (realizamos duas chamadas recursivas).

Método de ordenação Quick Sort (parte 2)

```
private int separar (double[] vetor, int inicio, int fim) {
    double pivo = vetor[inicio];
    int i = inicio + 1, f = fim;
    while (i <= f) {
      if (vetor[i] <= pivo) i++;
      else if (vetor[f] > pivo) f--;
      else {
         double troca = vetor[i];
         vetor[i] = vetor[f];
         vetor[f] = troca;
         i++;
         f--;
      }
    }
    vetor[inicio] = vetor[f];
    vetor[f] = pivo;
    return f;
  }
```

o primeiro elemento da lista será o pivot

comparamos o elemento com o pivô:
se encontramos um valor no lado correto
apenas vamos avançar i ou retroceder f

efetuamos a troca do elemento vetor[i] com vetor[f], de forma a deixar um item menor que o pivô no lado esquerdo e um elemento maior que o pivô no lado direito;
depois avançamos i e retrocedemos f;
o ciclo terminará quando i > f

efetuamos a troca do pivot (que estava em inicio) com o último elemento da sublista esquerda (que estava em f);
e retornamos a posição final do pivot, que é a posição correta onde ficou esse valor na lista

Analisando o método de ordenação **Quick Sort**

- O método Quick Sort , como foi mostrado nos slides anteriores (com seleção do primeiro ou último elemento da lista como pivot), tem complexidade em tempo, no pior caso (quando a lista está completamente ordenada) de **$O(n^2)$**.
- O critério utilizado na seleção do pivot é o ponto crítico deste método.
- Nos algoritmos básicos encontrados na literatura selecionam o primeiro ou o último elemento como pivô. Contraditoriamente, este critério provocaria o pior caso $O(n^2)$ quando a lista está ordenada ou quase ordenada (crescente ou dec.).
- Existem alguns ajustes que melhoram a eficiência do método, por exemplo:
    - selecionar como pivô a "mediana de três valores" do vetor a ser ordenado (os elementos são trocados de posição, veja slide a seguir);
    - selecionar aleatoriamente a posição do elemento que será selecionado como pivô, a mediana de três itens aleatórios etc.
- Autores demostram que estas otimizações tem uma alta probabilidade de garantir uma eficiência melhor para este método, de **$O(n \log n)$**.

Mediana e média aritmética (são conceitos diferentes)

Suponhamos os valores:
1 5 6
a média aritmética é (1 + 5 + 6) / 3 = 4
a mediana é 5

Em um vetor (grupo de valores) com quantidade ímpar:
2 4 5 12 21 34 39
a média aritmética é (2 + 4 + 5 + 12 + 21 + 34 + 39) / 7 = 16.71
a mediana é 12

Em um vetor (grupo de valores) com quantidade par:
2 4 5 12 18 21 34 39
a mediana é (12 + 18) / 2 = 30 / 2 = 15

A mediana pode ser mais útil que a média: a mediana pode representar melhor um valor típico, porque não é tão distorcida por valores muito altos ou baixos. Também, é mais rápido calcular uma mediana que uma média aritmética.

Analisemos: mediana dos salários de uma empresa, mediana de três valores para selecionar o pivô no método de ordenação Quick Sort.

Seleção de pivô no método de ordenação Quick Sort

Uma estratégia seria selecionar como pivô a mediana de três valores do trecho do vetor a ser ordenado (os itens serão trocados de posição): mediana dos valores esquerdo, central e direito.

```
public double medianaDeTres (double vetor[], int esq, int dir) {
    int centro = (esq + dir) / 2;
    if ( vetor[esq] > vetor[centro] ) troca ( vetor, esq, centro );
    if ( vetor[esq] > vetor[dir] ) troca ( vetor, esq, dir );
    if ( vetor[centro] > vetor[dir] ) troca ( vetor, centro, dir );
    double pivo = vetor[centro];
    // mas colocamos o pivô à esquerda,
    // para manter a lógica original usada
    // no método separar:
    troca( vetor, esq, centro );
    return pivo;  // retorna a "mediana de três"
}
```

```
public void troca (double vet[], int i, int j) {
    if( i==j ) return;
    double temp = vet[i];
    vet[i] = vet[j];
    vet[j] = temp;
}
```

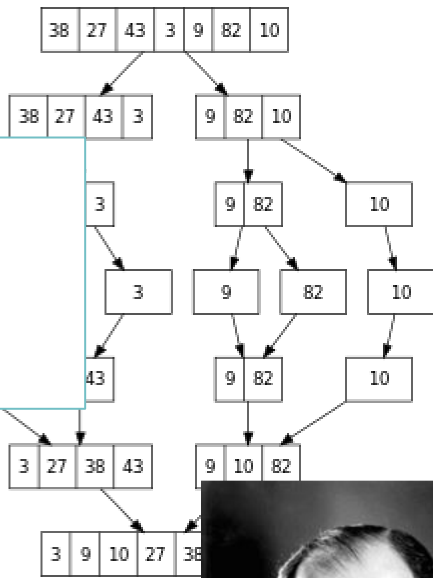
Obs: a) existem outras versões e otimizações desta lógica apresentada, b) o método *separar* deverá ser alterado.

Eficiência do método de ordenação Quick Sort

- Existe utilização de memória adicional por causa da recursão. Alguns autores afirmam que este método é de O(log n) quanto à complexidade de espaço (relativamente aceitável, mas é um critério a ser considerado).
- O método Quick Sort poderá chegar a ter uma eficiência (tempo) na média de **O(n log n)**. Desta forma, seria uma opção melhor para ordenação que os algoritmos estudados anteriormente.

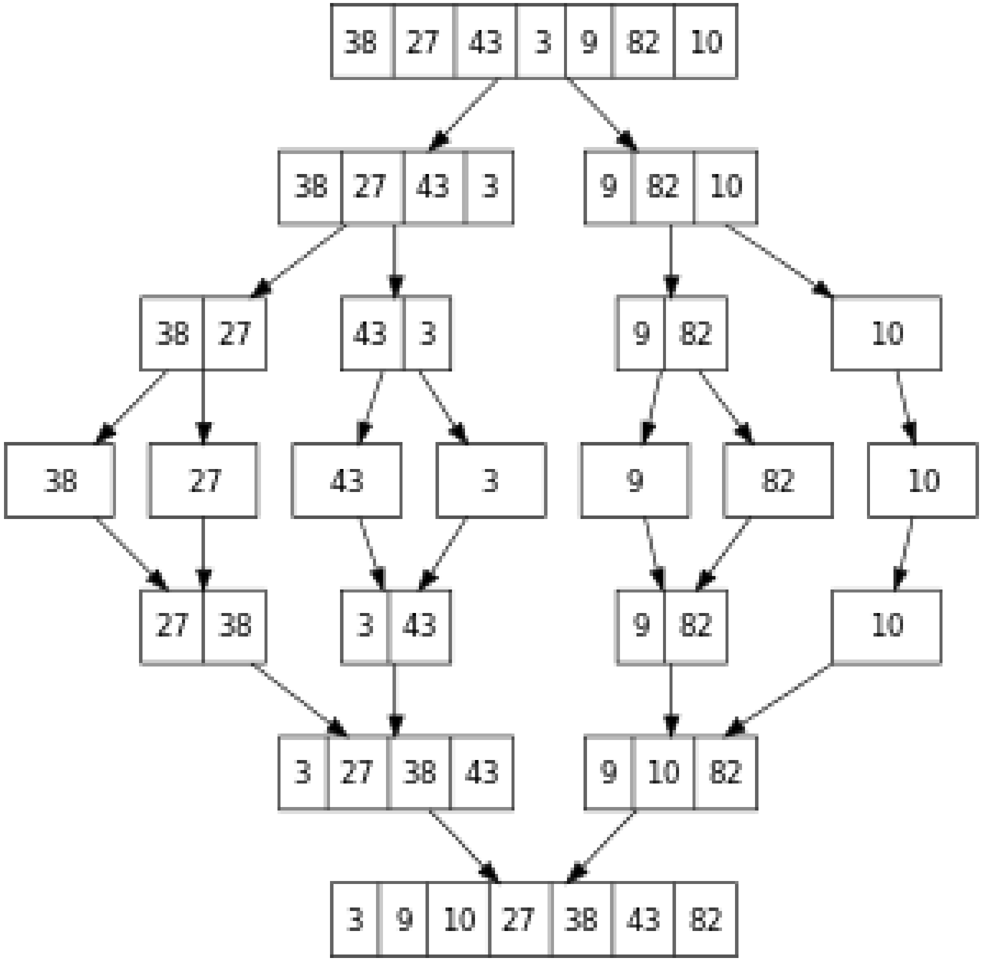
Algoritmo	Eficiência média (tempo)
Bubble sort	$O(n^2)$
Selection sort	$O(n^2)$
Insertion sort	$O(n^2)$
Quick sort	$O(n \log n)$

Merge Sort



John von Neumann: método proposto em 1945
Matemático, cientista da
computação húngaro, 1903-1957

Método de ordenação Merge Sort



Fonte da figura:
https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg

O Merge Sort (merge: fundir, misturar) se baseia no método de divisão e conquista (divide-and-conquer, análise e síntese).

Podemos considerar três etapas ou fases:

1. **Divisão**: se o tamanho da entrada for menor que um certo limite (normalmente consideramos um ou dois elementos), resolvemos diretamente e retornamos a solução obtida. Em qualquer outro caso, os dados de entrada são divididos, normalmente em uma ou duas partes.

2. **Recursão**: Cada parte obtida no passo anterior será resolvida, utilizando o mesmo método, em forma recursiva.

3- **Conquista**: as soluções dos subproblemas são unidas em uma única solução, também em etapas para cada tamanho de partição, fundindo até chegar no vetor final ordenado.

```java
package ordenacaomergesort;
// Programação: Ledón (implementação baseada no algoritmo de Mark Allen Weiss)
public class OrdenacaoMergeSort {
   public static void main(String[] args) {
        new OrdenacaoMergeSort();
   }

   public OrdenacaoMergeSort() {
      double vet[] = {71.2, 0.3, 6.3, -1.2, 5.4, 0.5, 0.2, 91.5, 33.3, 0.9};  //este é o vetor que queremos ordenar
      double tempVet [] = new double[vet.length];  // vetor auxiliar
      System.out.println("Vetor desordenado:");
      visualizarVetor(vet);
      mergeSort(vet, tempVet, 0, vet.length-1);  // ordenamos o vetor vet completo
      System.out.println("Vetor ordenado:");
      visualizarVetor(vet);
   }

   public void mergeSort( double vet[], double tempVet[], int esq, int dir ){
      if (esq < dir) { // caso contrário (se o trecho do vetor tiver mais de um elemento) abandonaremos este método (fim da recursão)
        int centro = (esq + dir)/2;
        mergeSort(vet, tempVet, esq, centro);          rdena
        mergeSort(vet, tempVet, centro+1, dir);        dena
        merge(vet, tempVet, esq, centro+1, dir);       istura
      }
   }
```

- determinamos o **centro** do trecho analisado;
- solicitamos ordenar as partes à esquerda e direita, efetuando duas chamadas recursivas ao próprio método **mergeSort**;
- misturamos, fundimos (método **merge**) os trechos analisados entre **esq** e **dir**, cujo ponto central é **centro+1**

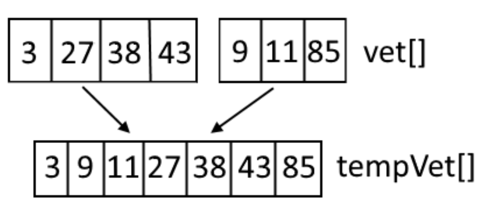
```java
public void merge( double vet [], double tempVet [] , int esq, int centro, int dir ) {
    int fimTrechoEsquerdo = centro - 1;
    int i = esq;
    int qtdeElementos = dir - esq + 1;
    //------------ Mistura, fusão inicial de elementos:
    while( esq <= fimTrechoEsquerdo && centro <= dir )
        if( vet[ esq ] <= vet[ centro ] )
            tempVet[ i++ ] = vet[ esq++ ];
        else
            tempVet[ i++ ] = vet[centro++];
    //------------- Ciclo para copiar o resto da metade esquerda:
    while( esq <= fimTrechoEsquerdo )
        tempVet[ i++ ] = vet[ esq++ ];
    //------------- Ciclo para copiar o resto da metade direita:
    while( centro <= dir )
        tempVet[ i++ ] = vet[centro++];
    //-------- Finalmente, copiamos o trecho do vetor temporário para o vetor original:
    for( i = 0; i < qtdeElementos; i++, dir-- )
        vet[dir] = tempVet[dir];
}

public void visualizarVetor(double vetor[]) {
    for (int i = 0; i < vetor.length; i++) {
        System.out.print(vetor[i] + "   ");
    }
    System.out.println();
}
}
```

adiciona no vetor temporário **tempVet** o menor elemento de **vet** achado, seja da parte esquerda ou da direita;
o índice do item copiado (esq ou centro) será incrementado e também o índice i de **tempVet**

com estes dois ciclos copiamos de **vet** para **tempVet** os itens que poderiam ter ficado na parte esquerda ou na parte direita do trecho analisado

- este último ciclo copia os elementos do vetor auxiliar **tempVet** para o vetor original **vet**;
- é um ciclo que repete **qtdeElementos** vezes;
- observe que o índice utilizado para copiar é **dir** (extremo direito do trecho que este método está misturando [merge], e que é o único índice que não foi alterado nos ciclos anteriores);
- **dir** será decrementado com **dir--** até chegar no início do trecho analisado.



Analisando os métodos de ordenação estudados

Existem diferentes fatores a serem considerados: quantidade de dados a serem ordenados, ordenação ou organização prévia dos dados, eficiência quanto a velocidade, eficiência quanto à memória utilizada, complexidade da implementação e ajustes específicos que melhoram cada método.

Da lista a seguir, os dois algoritmos mais eficientes quanto a desempenho são o Quick Sort e o Merge Sort. O Merge Sort, possui uma limitação importante quanto a utilização de memória adicional (vetor temporário). Ambos algoritmos são recursivos e utilizam memória adicional por causa da recursão (pilha, stack). O Quick Sort utiliza pouca.

Apesar de ser bastante mais ineficientes, Bubble, Insertion e Selection se caracterizam pela simplicidade e pouco requerimento de memória adicional.

Algoritmo	**Complexidade (tempo)**			**Complexidade (espaço)**
	melhor caso	**médio**	**pior caso**	**---**
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

Bibliografia da disciplina

BIBLIOGRAFIA BÁSICA	BIBLIOGRAFIA COMPLEMENTAR
CORMEN, Thomas H.; CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, Campus, 2002. GOODRICH, Michael T.; TAMASSIA, Roberto. Estruturas de dados e algoritmos em Java. 2. ed. Porto Alegre: São Paulo: Bookman, 2002. PREISS, B. R. Estruturas de Dados e Algoritmos: Padroes de Projetos Orientados a Objetivos Com Java. Rio de Janeiro: Campus, 2001.	ASCENCIO, A. F. G.; ARAÚJO, G. S. Estruturas de Dados. São Paulo: Pearson, 2011. [eBook] EDELWEISS, N.; GALANTE, T. Estruturas de Dados. Porto Alegre: Bookman, 2009. [eBook] MORIN, P. Open Data Structures (in Java) Creative Commons, 2011. Disponível em http://opendatastructures.org/ods-java.pdf [eBook] PUGA, S.; RISSETTI, G. Estruturas de Dados com aplicações em Java, 2a ed. São Paulo: Pearson, 2008. [eBook] SHAFFER, C. A.; Data Structures and Algorithm Analysis. Virginia Tech, 2012. Disponível em

Referências adicionais

- AURÉLIO. Aurélio Buarque de Holanda Ferreira. **Mini Aurélio. Dicionário da Língua Portuguesa**. Curitiba: Positivo, 2004.

- HOUAISS. **Mini Houaiss. Dicionário da Língua Portuguesa**. Rio de Janeiro: Objetiva, 2009.

- PADOVANI, U.; CASTAGNOLA, L. **História da Filosofia**. São Paulo: Melhoramentos, 1972.

- WEISS, M. A. **Data Structures and Algorithm Analysis**. 2nd Edition. The Benjamin/Cummings Publishing Company: California, 1994.