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

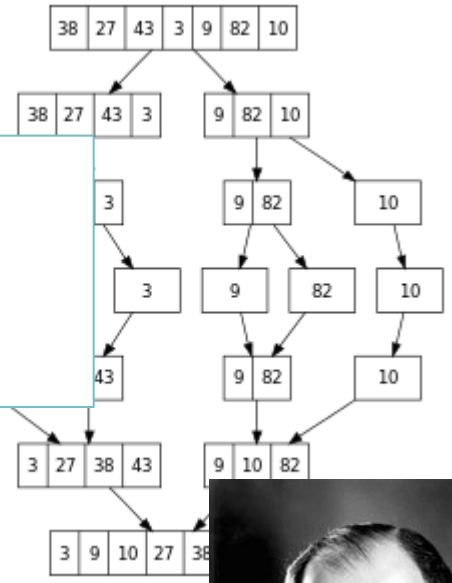
Divisão e conquista

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 utilizando normalmente o mesmo método de solução sendo utilizado;
- as soluções das instâncias menores são combinadas para produzir uma solução do problema original.

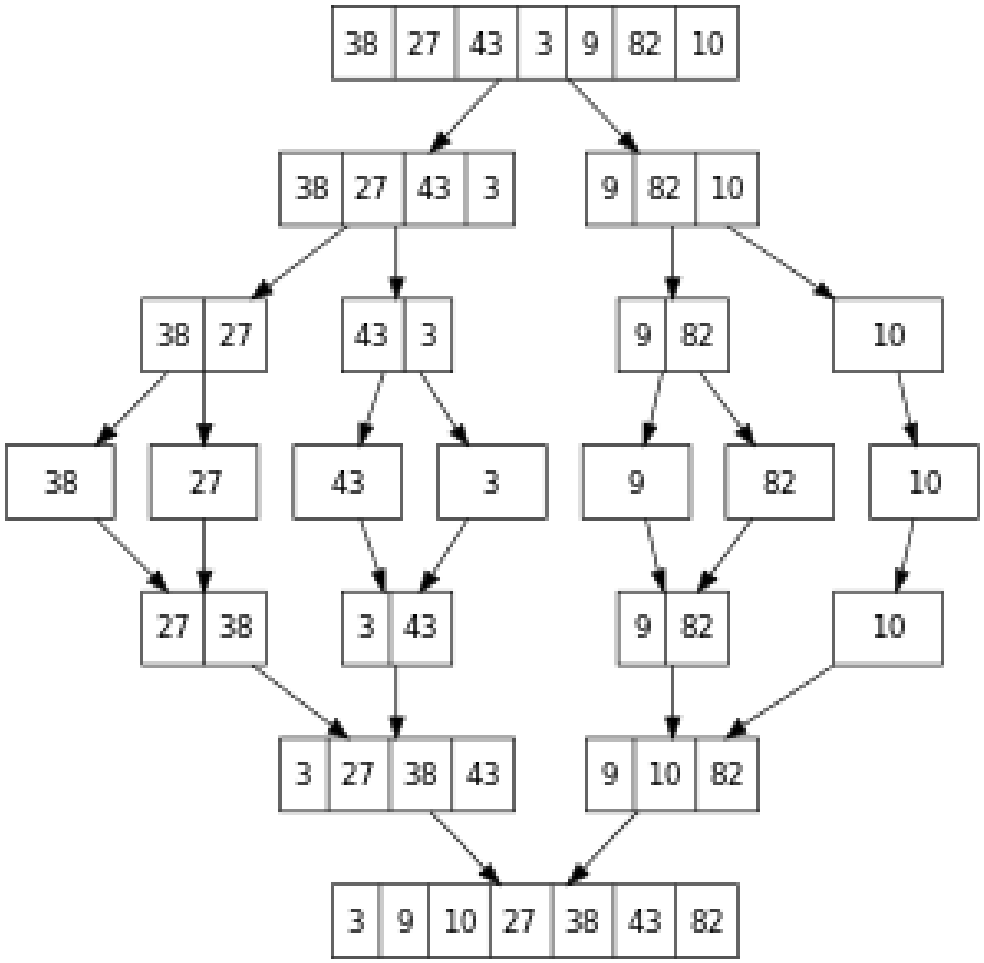
Os métodos de ordenação Quick Sort e Merge Sort são "naturalmente" **recursivos** e utilizam a abordagem da **divisão e conquista** para resolver o problema.

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.

```
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);
         mergeSort(vet, tempVet, centro+1, dir);
         merge(vet, tempVet, esq, centro+1, dir);
      }
   }
```

- 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**

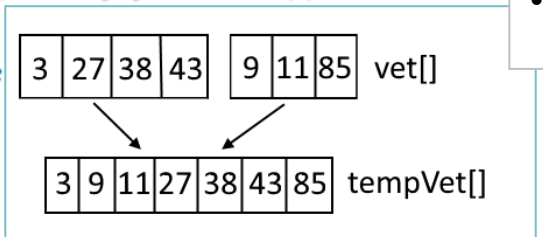
```java
public void merge( double vet [], double tempVet [] , int esq, int centro, int dir ) {
    int fimTrechoEsquerdo = centro - 1;
    int i = esq;
    int qtdeElementos = dir - esq + 1;
    //------------ Mistura, fusão inicial de elementos:
    while( esq <= fimTrechoEsquerdo && centro <= dir )
        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 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 Shell Sort (omitido) estaria na terceira posição. O Merge Sort, muito eficiente quanto a tempo, 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 memória, mas o Merge Sort é O(n) quanto a memória auxiliar.

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)