

Notas de Aula de Algoritmos e Programação de Computadores

FLÁVIO KEIDI MIYAZAWA

com a colaboração de

TOMASZ KOWALTOWSKI

Instituto de Computação - UNICAMP

Versão 2000.1

Estas notas de aula não devem ser usadas como única fonte de estudo. O aluno deve ler outros livros disponíveis na literatura.

Nenhuma parte destas notas pode ser reproduzida, qualquer que seja a forma ou o meio, sem a permissão dos autores.

Os autores concedem a permissão explícita para a utilização e reprodução deste material no contexto do ensino de disciplinas regulares dos cursos de graduação sob a responsabilidade do Instituto de Computação da UNICAMP.

© Copyright 2000

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas-SP
{fkm,tomasz}@ic.unicamp.br

11 Algoritmos de Ordenação

Nesta seção vamos considerar outros algoritmos para ordenação: *InsertionSort*, *MergeSort* e *QuickSort*. Apresentaremos os algoritmos MergeSort e QuickSort implementados recursivamente e utilizando uma importante técnica chamada *Divisão e Conquista*. Estes dois algoritmos estão entre os algoritmos mais rápidos para ordenação usando apenas comparação entre elementos. Neste tipo de ordenação, o algoritmo QuickSort é o que tem o melhor tempo médio para se ordenar seqüências em geral.

11.1 Algoritmo InsertionSort

O algoritmo InsertionSort também usa a técnica de projeto de algoritmo por indução para resolver o problema, i.e., supondo saber resolver um problema pequeno, resolve se um maior.

Considere um vetor $v = (v_1, v_2, \dots, v_{n-1}, v_n)$. Vamos supor que já sabemos ordenar o vetor $v' = (v_1, v_2, \dots, v_{n-1})$ (com $n - 1$ elementos, e portanto menor que n). A idéia é ordenar o vetor v' e depois inserir o elemento v_n na posição correta (daí o nome InsertionSort). Primeiramente apresentamos um algoritmo recursivo que implementa esta idéia.

```
procedure InsertionSortRecursivo(var v : TipoVetorReal;n:integer);  
var i : integer;  
    aux : real;  
begin  
  if n>1 then begin  
    InsertionSortRecursivo(v,n-1); {Ordena os n-1 primeiros elementos}  
    aux:=v[n]; i:=n;  
    while (i>1) and (v[i-1]>aux) do begin  
      v[i] := v[i-1];  
      i:=i-1;  
    end;  
    v[i] := aux;  
  end;  
end; { InsertionSortRecursivo }
```

O algoritmo tem uma estratégia indutiva e fica bem simples implementá-lo de maneira recursiva. Note que a base da recursão é para o vetor com no máximo 1 elemento (nestes caso o vetor já está ordenado e portanto o problema para este vetor já está resolvido). Uma vez que desenvolvemos a estratégia, podemos observar que podemos descrevê-lo melhor de forma iterativa, uma vez que o algoritmo acima faz uma chamada recursiva no início da rotina (veja seção 10.4). Assim, uma estratégia mais eficiente, usando duas estruturas de repetição, é apresentada a seguir.

```
procedure InsertionSort(var v : TipoVetorReal;n:integer);  
var i,j : integer;  
    aux : real;  
begin  
  for i:=2 to n do begin  
    aux:=v[i]; j:=i;  
    while (j>1) and (v[j-1]>aux) do begin  
      v[j] := v[j-1];  
      j:=j-1;  
    end;  
    v[j] := aux;  
  end;  
end; { InsertionSort }
```

No caso médio este algoritmo também gasta um tempo computacional quadrático em relação a quantidade de elementos. Por outro lado, se a instância estiver *quase ordenada*, este algoritmo é bastante rápido.

11.2 Algoritmo MergeSort e Projeto por Divisão e Conquista

O algoritmo MergeSort também usa a estratégia por indução. Além disso, o algoritmo usa de outra técnica bastante importante chamada Divisão e Conquista. Neste tipo de projeto, temos as seguintes etapas:

- Problema suficientemente pequeno: Resolvido de forma direta.
- Problema não é suficientemente pequeno:

Divisão: O problema é dividido em problemas menores.

Conquista: Cada problema menor é resolvido (recursivamente).

Combinar: A solução dos problemas menores é combinada de forma a construir a solução do problema.

No algoritmo MergeSort, a etapa de divisão consiste em dividir o vetor, com n elementos, em dois vetores (subvetores) de mesmo tamanho ou diferindo de no máximo um elemento. No caso, um vetor $v = (v_1, \dots, v_n)$ é dividido em vetores $v' = (v_1, \dots, v_{\lfloor n/2 \rfloor})$ e $v'' = (v_{\lfloor n/2 \rfloor + 1}, \dots, v_n)$.

Para facilitar, em vez de realmente dividir o vetor em dois outros vetores, usaremos índices para indicar o início e o fim de um subvetor no vetor original. Assim, estaremos sempre trabalhando com o mesmo vetor, mas o subvetor a ser ordenado em cada chamada recursiva será definido através destes índices. Caso o vetor tenha no máximo 1 elemento, o vetor já está ordenado e não precisamos subdividir em partes menores (base da recursão).

Uma vez que o vetor (problema) foi dividido em dois subvetores (problemas menores), vamos ordenar (conquistar) cada subvetor (subproblema) recursivamente.

A combinação das soluções dos subproblemas é feita intercalando os dois subvetores ordenados em apenas um vetor. Para isso, usaremos dois índices para percorrer os dois subvetores e um terceiro para percorrer o vetor que receberá os dois vetores intercalados, vetor resultante. A idéia é começar os índices no início dos dois vetores e comparar os dois elementos localizados por estes índices e atribuir no vetor resultante o menor valor. Em seguida, incrementamos o índice que tinha o menor valor. Este processo se repete até que tenhamos intercalado os dois vetores no vetor resultante.

No quadro seguinte apresentamos o procedimento para intercalar os vetores $V[inicio, \dots, meio]$ e $V[meio + 1, \dots, fim]$ e na figura 36 ilustramos seu comportamento.

```
procedure IntercalaMergeSort(var v : TipoVetorReal; Inicio,Meio,Fim:integer);
var i,j,k : integer; {v1 = [Inicio..Meio], v2 = [Meio+1..Fim]}
begin { supondo a declaração de Vaux no procedimento principal }
  i:=Inicio;    j:=Meio+1;    k:=Inicio;
  while (i<=Meio) and (j<=Fim) do begin
    if (v[i]>v[j]) then begin {Sempre inserindo o menor em Vaux}
      vaux[k]:=v[j];  j:=j+1;
    end else begin
      vaux[k]:=v[i];  i:=i+1;
    end;
    k:=k+1;
  end;
  while (i<=Meio) do begin
    vaux[k]:=v[i]; {Inserindo os elementos do primeiro vetor}
    i:=i+1; k:=k+1;
  end;
  while (j<=Fim) do begin
    vaux[k]:=v[j]; {Inserindo os elementos do segundo vetor}
    j:=j+1; k:=k+1;
  end;
  for k:=Inicio to Fim do v[k] := vaux[k]; {Copiando para o vetor original}
end; { IntercalaMergeSort }
```

A seguir, descrevemos o algoritmo MergeSort.

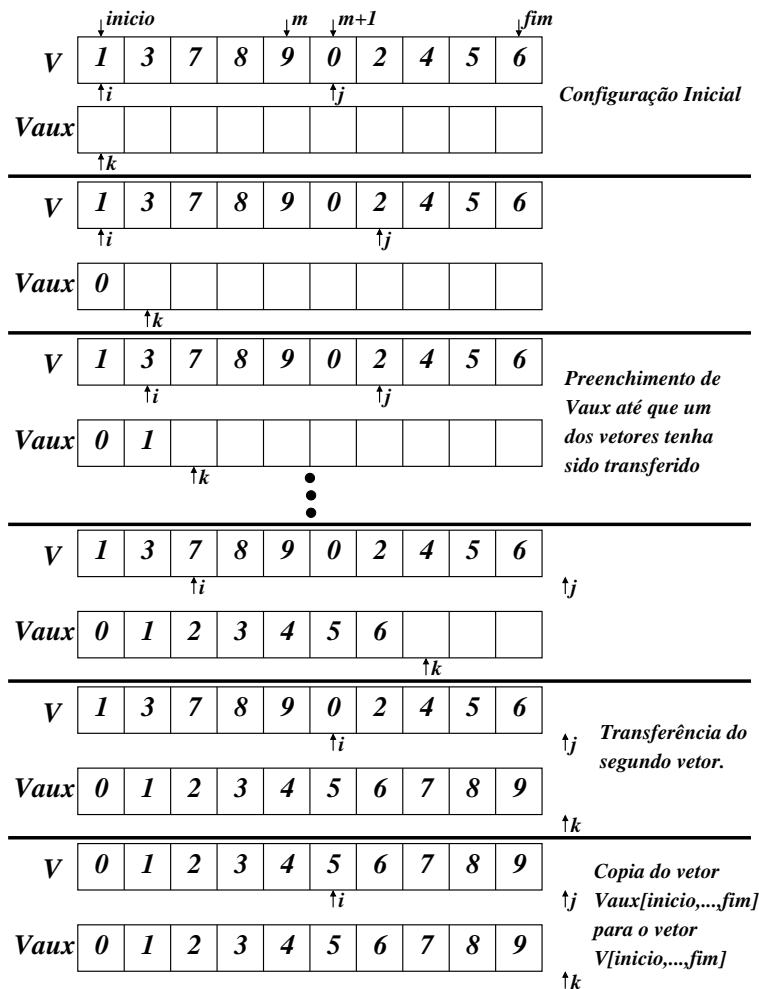


Figura 36: Intercalação de dois sub-vetores.

```

procedure MergeSort(var V : TipoVetorReal; n:integer);
var Vaux : TipoVetorReal; {vetor auxiliar para fazer intercalacoes}

{ --> Inserir rotina Intercala IntercalaMergeSort <-- }

procedure MergeSortRecursivo(var V : TipoVetorReal; inicio,fim:integer);
var meio : integer;
begin
  if (fim > inicio) then begin {Se tiver quantidade suficiente de elementos}
    meio:=(fim+inicio) div 2;      {Dividindo o problema}
    MergeSortRecursivo(V,inicio,meio); {Conquistando subproblema 1}
    MergeSortRecursivo(V,meio+1,fim); {Conquistando subproblema 2}
    IntercalaMergeSort(V,inicio,meio,fim); {Combinando subproblemas 1 e 2}
  end;
end; { MergeSortRecursivo }

begin
  MergeSortRecursivo(V,1,n);
end; { MergeSort }

```

Na figura 37 apresentamos uma simulação do algoritmo MergeSort para o vetor $[7,3,8,1,4,2,6,5]$. Denotamos a chamada da rotina $MergeSort(V)$ por $M(V)$, e a rotina IntercalaMergeSort por $I(v_1, v_2)$. Se uma chamada da rotina $M(V)$ exige chamadas recursivas, esta é substituída pelas chamadas $I(M(V'), M(V''))$, onde V' e V'' são as duas partes de V .

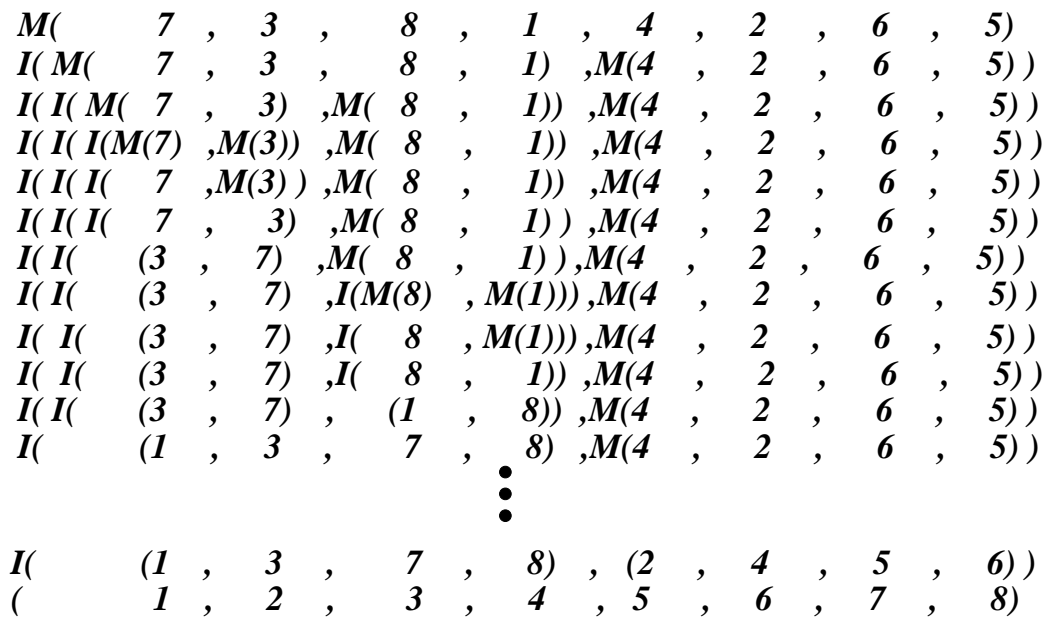


Figura 37: Simulação das chamadas recursivas do procedimento MergeSort para o vetor 7,3,8,1,4,2,6,5.

11.3 Algoritmo QuickSort

O algoritmo QuickSort também usa a técnica de divisão e conquista, dividindo cada problema inicial em duas partes. Neste caso, a etapa de divisão em dois subproblemas é mais sofisticada mas por outro lado a etapa de combinação é simples (lembrando que no algoritmo MergeSort, dividir é simples enquanto combinar é mais sofisticado).

Para dividir um vetor $v = (v_1, \dots, v_n)$ em dois subvetores $v' = (v'_1, \dots, v'_p)$ e $v'' = (v''_1, \dots, v''_q)$, o algoritmo usa de um pivô X tal que $v'_i \leq X, i = 1, \dots, p$ e $X < v''_j, j = 1, \dots, q$.

Uma vez que o vetor v foi dividido nos subvetores v' e v'' , estes são ordenados recursivamente (conquistados).

Por fim, os vetores são combinados, que nada mais é que a concatenação dos vetores ordenados v' e v'' .

A etapa mais complicada neste algoritmo é a etapa de divisão, que pode ser feita eficientemente no mesmo vetor $v = (v_1, \dots, v_n)$. Primeiro escolhemos um pivô X . Uma das vantagens de se escolher este pivô do próprio vetor a ser ordenado é que podemos implementar a estratégia garantindo sempre que cada subvetor é estritamente menor que o vetor original. O que faremos é tomar o pivô $X \in v$ e dividir o vetor v em subvetores v' e v'' de tal maneira que a ordenação final se torne o vetor $v' \parallel (X) \parallel v''$, onde o operador \parallel é a concatenação de vetores.

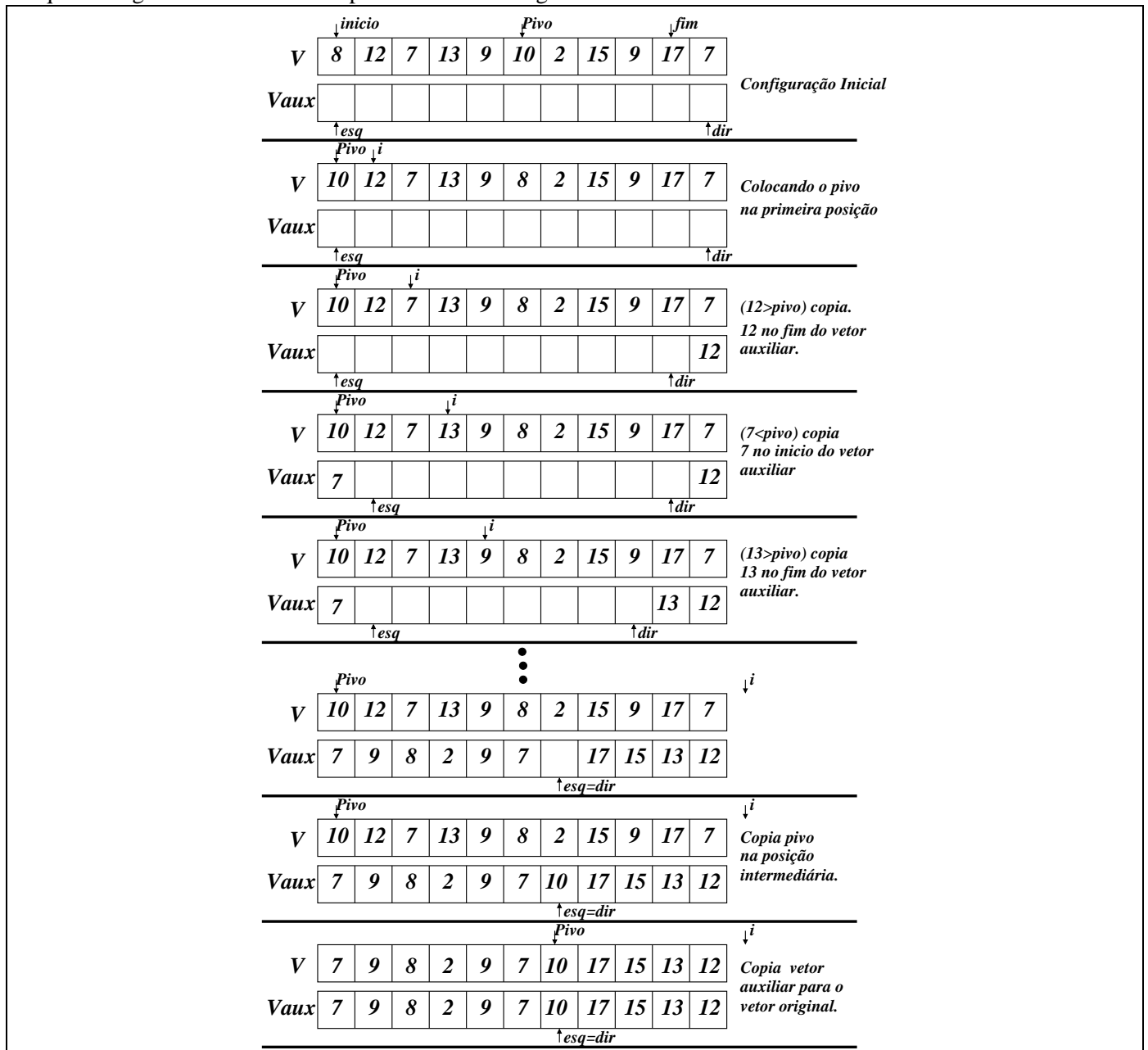
Antes de apresentar o algoritmo de particionamento em um vetor, vamos apresentá-lo usando um vetor auxiliar. A implementação é mais simples e fica mais fácil de se entender a idéia do programa. Primeiro a rotina escolhe um pivô (posição $(IndPivo \leftarrow \lfloor inicio + fim \rfloor / 2)$). Troca com o primeiro elemento ($V[inicio]$). Em seguida percorre-se todos os demais elementos $V[inicio + 1..fim]$ e insere os elementos menores ou iguais ao pivô, lado a lado, no início do vetor auxiliar. Os elementos maiores que o pivô são inseridos lado a lado a partir do fim do vetor auxiliar. Por fim, o pivô é inserido na posição intermediária e o vetor auxiliar é copiado para o vetor auxiliar.

```

function ParticionaQuickSort(var v : TipoVetorReal; inicio, fim: integer): integer;
var esq, dir, IndicePivo: integer;   Pivo: real;
    { Supondo a declaração de Vaux (vetor auxiliar p/ partição) no procedimento principal }
begin
    troca(v[inicio], v[(inicio+fim) div 2]); { escolha de um pivo }
    Pivo := v[Inicio]; { Pivo fica na posição Inicio }
    esq:=Inicio;   dir:=Fim;
    for i:=Inicio+1 to fim do begin
        if (V[i] > Pivo) then begin
            Vaux[dir] := V[i];   dec(dir);
        end else begin
            Vaux[esq] := V[i];   inc(esq);
        end;
    end;
    V[esq] := Pivo; { esq=dir }
    for i:=Inicio to fim do V[i] := Vaux[i]; { Copia Vaux[inicio..fim] para V[inicio..fim] }
    ParticionaQuickSort := esq;
end;

```

No quadro seguinte mostramos comportamento deste algoritmo.



Agora vamos descrever o procedimento para particionar um vetor, através de um pivo e sem uso de vetor auxiliar. Primeiramente colocaremos o pivo na primeira posição do vetor. Para obter os subvetores v' e v'' , usaremos de dois índices no vetor original, esq e dir . Inicialmente esq começa no segundo elemento do vetor (logo depois do pivo) e dir começa no último elemento do vetor. Em seguida, fazemos o índice esq “andar” para a direita enquanto o índice apontar para um elemento menor ou igual ao pivo. Fazemos o mesmo com o índice dir , “andando-o” para a esquerda enquanto o elemento apontado por ele for maior que o pivo. Isto fará com que o índice esq vá pulando todos os elementos que vão pertencer ao vetor v' e o índice dir pula todos os elementos que vão pertencer ao vetor v'' . Quando os dois índices pararem, estes vão estar parados em elementos que não são os pertencentes aos correspondentes vetores e neste caso, trocamos os dois elementos e continuamos o processo. Para manter a separação entre v' e v'' viável, vamos fazer este processo sempre enquanto $esq < dir$. Quando todo este processo terminar (i.e., quando $esq \geq dir$), iremos inserir o Pivo (que estava na primeira posição) separando os dois vetores. O algoritmo QuickSort está descrito na página 112. Existem outras implementações do algoritmo QuickSort que são mais eficientes que a versão que apresentamos, embora são também um pouco mais complicadas.

```

function ParticionaQuickSort(var v : TipoVetorReal; inicio, fim: integer): integer;
var esq, dir, IndicePivo : integer;
    Pivo          : real;
begin
    troca(v[inicio], v[(inicio+fim) div 2]); {escolha de um pivo}
    Pivo := v[Inicio]; {Pivo fica na posição Inicio}
    esq := Inicio + 1; {primeiro elemento mais a esquerda, pulando o pivo}
    dir := Fim;      {primeiro elemento mais a direita}
    while (esq < dir) do begin
        while (esq < dir) and (v[esq] <= Pivo) do inc(esq);
        while (esq < dir) and (pivo < v[dir]) do dec(dir);
        if esq < dir then begin
            troca(v[esq], v[dir]);
            inc(esq);    dec(dir);
        end;
    end; {dir <= esq, e |esq - dir| <= 1}
    if v[dir] <= Pivo then IndicePivo := dir {dir = esq - 1 ou dir = esq}
    else IndicePivo := dir - 1;      {esq = dir}
    troca(v[IndicePivo], v[Inicio]); {Coloca o pivo separando os dois subvetores}
    ParticionaQuickSort := IndicePivo;
end;

```

A seguir, apresentamos a descrição do algoritmo QuickSort.

```

procedure QuickSort(var V : TipoVetorReal; n: integer);
{ --> Declarar Vaux e inserir rotina Intercala ParticionaQuickSort <-- }
procedure QuickSortRecursivo(var V : TipoVetorReal; inicio, fim: integer);
var IndicePivo : integer;
begin
    if (fim > inicio) then begin {se tem pelo menos 2 elementos}
        IndicePivo := ParticionaQuickSort(v, Inicio, Fim); {Dividindo o problema}
        QuickSortRecursivo(V, Inicio, IndicePivo - 1); {Conquistando subproblema 1}
        QuickSortRecursivo(V, IndicePivo + 1, Fim);   {Conquistando subproblema 2}
    end;      {Não é preciso fazer nada para combinar os dois subproblemas}
end;
begin
    QuickSortRecursivo(V, 1, n);
end;

```

11.4 Exercícios

1. Faça uma rotina recursiva para ordenar um vetor usando a estratégia do algoritmo SelectionSort.
2. Faça um algoritmo para ordenar um vetor usando uma estratégia parecida com a usada pelo algoritmo MergeSort só que em vez de dividir em duas partes, divide em três partes. (I.e., se o vetor for suficientemente pequeno, ordena-se de forma direta. Caso contrário este algoritmo divide o vetor em 3 partes iguais (ou diferindo de no máximo um elemento). Ordena recursivamente cada um dos três subvetores e por fim o algoritmo intercala os três vetores, obtendo o vetor ordenado.)
3. Seja (v_i, \dots, v_f) elementos consecutivos de um vetor v do índice i ao índice f . Projete um procedimento recursivo com o cabeçalho

procedure minmax(**var** v : *TipoVetorReal*; i, f : **integer**; **var** minimo, maximo: **real**);

que retorna em *minimo* e *maximo* o menor e o maior valor que aparecem no vetor v dos índices i ao índice f , respectivamente. O procedimento deve dividir o vetor em dois subvetores de tamanhos quase iguais (diferindo de no máximo 1 elemento).

4. (*K-ésimo*) O problema do K -ésimo consiste em, dado um vetor com n elementos, encontrar o k -ésimo menor elemento do vetor. Note que se o vetor já estiver ordenado, podemos resolver este problema obtendo o elemento que está na posição k do vetor. Isto nos dá um algoritmo que usa a ordenação para resolver o problema do k -ésimo. Entretanto, é possível se resolver este problema de maneira mais eficiente, no caso médio. A idéia é combinar duas estratégias:

- (a) Estratégia da busca binária, de descartar uma parte do vetor.
- (b) Estratégia do algoritmo *ParticionaQuickSort*, que divide o vetor V em três partes, $V = (V' || (X) || V'')$, onde V' contém os elementos menores ou iguais a X e V'' contém os elementos maiores que X .

Use estas duas estratégias para desenvolver uma função que dado V , n e k , devolve o k -ésimo menor elemento do vetor.

5. Faça um programa contendo procedimentos com os seguintes cabeçalhos:

- **procedure** MergeSort(**var** v :*TipoVetor*; n :**integer**);
- **procedure** QuickSort(**var** v :*TipoVetor*; n :**integer**);
- **procedure** SelectionSort(**var** v :*TipoVetor*; n :**integer**);
- **procedure** InsertionSort(**var** v :*TipoVetor*; n :**integer**);
- **procedure** aleatorio(**var** v :*TipoVetor*; n :**integer**);

onde *TipoVetor* é um vetor de números:

type TipoVetor=**array** [1..30000] of **integer**;

As rotinas MergeSort, QuickSort, InsertionSort e SelectionSort são rotinas para ordenar usando os algoritmos do mesmo nome. A rotina aleatorio coloca no vetor v a quantidade de n elementos gerados (pseudo) aleatoriamente pela função random(MAXINT).

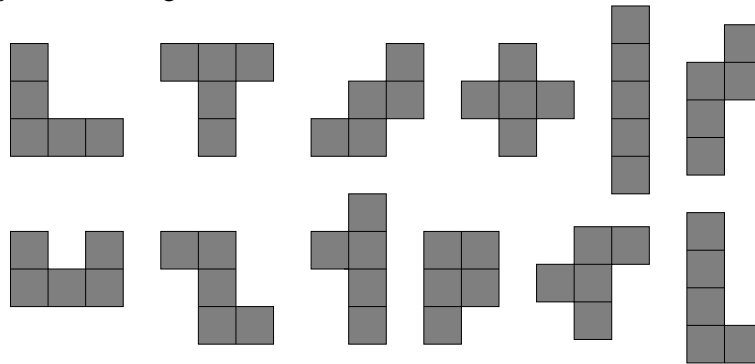
O programa deve ter rotinas auxiliares para leitura e impressão de vetores. O programa deve ter um menu com opcoes:

- (a) Ler um vetor com n inteiros (n também é lido). Inicialmente $n = 0$.
- (b) Gerar vetor aleatório com n elementos (n é lido).
- (c) Ordenar por MergeSort.
- (d) Ordenar por QuickSort.
- (e) Ordenar por InsertionSort.
- (f) Ordenar por SelectionSort.
- (g) Imprimir o vetor.
- (h) Sair do programa.

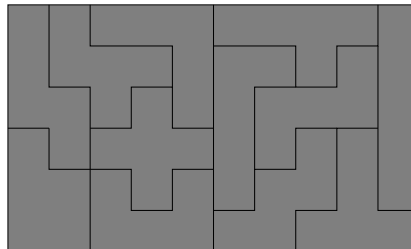
O aluno deve cronometrar os tempos gastos pelas rotinas de ordenação para ordenar n inteiros (pseudo) aleatórios, onde n pode ser $n = 1000, 5000, 10000, 15000, 20000, 25000, 30000$ (ou o quanto o computador suportar). Caso algum método seja muito lento, ignore sua execução. Não é necessário entregar estes tempos.

Exercícios mais elaborados, usando recursão

6. (*Passeio do Cavalo*) Considere um tabuleiro retangular com $n \times m$ casas. O cavalo é uma peça que “anda” neste tabuleiro, inicialmente em uma posição específica. O cavalo pode andar duas casas em um dos sentidos, horizontalmente (verticalmente), e uma casa verticalmente (horizontalmente). Dado um tabuleiro $n \times m$, e uma posição inicial do cavalo (i, j) , $1 \leq i \leq n$ e $1 \leq j \leq m$, faça um programa que resolve os seguintes itens:
- Diga se o cavalo consegue fazer um passeio passando por todas as $n \cdot m$ casas do tabuleiro.
 - Dado uma posição (a, b) , $1 \leq a \leq n$ e $1 \leq b \leq m$, diga se existe um passeio da casa de origem (i, j) até a casa (a, b) . Em caso positivo, imprima uma seqüência de movimentos que o cavalo precisa fazer para ir de (i, j) para (a, b) .
7. (*Problema das 8 Rainhas*) O problema das 8 rainhas consiste em encontrar uma disposição de 8 rainhas em um tabuleiro de xadrez. Uma rainha pode atacar qualquer outra que esteja na mesma linha, coluna ou diagonal do tabuleiro. Portanto o objetivo é dispor as 8 rainhas de maneira que qualquer duas delas não se ataquem. Faça um programa que encontra a solução para este problema, com n rainhas em um tabuleiro $n \times n$.
8. (*Pentominos*) Um Pentomino é uma figura formada juntando 5 quadrados iguais pelos lados. As 12 figuras possíveis, exceto por rotações, são as seguintes:



Dado um retângulo $n \times m$, o objetivo é preencher este retângulo com os pentominos, sem sobreposição entre pentominos e sem que um pentomino ultrapasse a borda do retângulo. Por exemplo, a figura seguinte mostra um retângulo 6×10 preenchida com pentominos. Além disso, um pentomino pode ser girado antes de ser encaixado no retângulo.



Faça um programa recursivo para resolver este problemas, dado um retângulo $n \times m$.