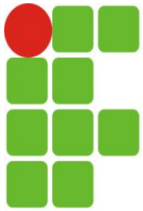


INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE
Campus Passo Fundo

ALGORITMOS II

Prof. Adilso Nunes de Souza

SUB-ROTINAS

- As sub-rotinas ou funções são úteis numa série de problemas relacionados a algoritmos, vamos utilizar funções para realizar uma 'quebra' do algoritmo em partes menores. A quebra será realizada de modo que cada função realize um pedaço do algoritmo a ser resolvido.
- Funções são subalgoritmos que podem retornar um valor.

SUB-ROTINAS

- A definição de uma função deve ocorrer na área destinada as funções: após a definição das variáveis e antes do programa principal.
- Para que uma função seja definida, basta colocarmos o tipo de retorno (int, float, etc...) ou caso não retorne valores “void”, um nome para ela (por exemplo “calculo”) adicionado de um abre e fecha parênteses. Os parênteses serão utilizados para proporcionar maiores funcionalidades.

SUB-ROTINAS

- Após os parênteses deve-se incluir os delimitadores de inicio e fim de bloco, um abre chaves “{” e um fecha chaves “}”, respectivamente.
- Os comandos do subalgoritmo (função) devem ser colocados de forma análoga a um algoritmo.
- Na linguagem C++ não é necessário utilizar a definição function para identificar uma função.

SUB-ROTINAS

- Para a definição dos nomes das funções deve-se seguir as regras dos identificadores e da mesma forma não podem haver nomes de funções repetidos. Os nomes das funções também não podem ser iguais a nomes de variáveis ou constantes. Um dos nomes que não podem ser usados é ‘main’ que dessa forma estaríamos nos referindo a parte principal do algoritmo.

SUB-ROTINAS

- Salienta-se que mesmo que se escreva as funções, o que comanda o algoritmo é o programa principal, dessa forma as funções devem ser chamadas pelo programa principal. Geralmente a chamada das funções é realizada no programa principal (que também pode ser considerado uma função), porém pode-se chamar uma função dentro de outra função, desde que a função que está sendo chamada tenha sido definida antes da função que a chamou.

SUB-ROTINAS

- Após a chamada para execução de uma função, o ponto de execução é deslocado para o primeiro comando da função (subalgoritmo) e segue executando sequencialmente, da mesma forma como um algoritmo. Ao alcançar o final da função, o ponto de execução volta para o comando subsequente da chamada da função.

SUB-ROTINAS

- Para chamar uma função basta colocar o nome da função seguido de abre e fecha parênteses.
- Ex.:

calculo();

- Caso a função necessite de parâmetros de entrada os mesmos devem ser passados quando da chamada da função, escrevendo-os dentro dos parênteses na ordem que foram declarados na função.

VARIÁVEIS

- Como já mencionado existem variáveis globais e locais, destaca-se a atenção durante o trabalho com funções sobre os dois tipo.
- As variáveis definidas na “área de definição de variáveis” de um algoritmo/programa, são consideradas globais. Elas são conhecidas em todas as partes e seu valor também pode ser verificado ou incluso em operações aritméticas nas diferentes partes do programa

VARIÁVEIS

- Ocasionalmente, uma variável é utilizada em apenas uma função, torna-se pouco cômodo deixar sua definição na área das globais. Para isto, podemos incluir uma nova seção de variáveis, logo após a declaração da função, definindo nela , as que serão usadas somente na função em questão. A estas variáveis, declaradas na parte interna de uma função, damos o nome de “variáveis locais”.

VARIÁVEIS

- Ex.

```
/* Comentário com a identificação do algoritmo/programa */
int   x,y;
float r;

calcula(){
   int   i,j,k;
   float p;

   ...
   ...
}
```

Variáveis globais, existem em todas as partes do programa

Variáveis locais, só existem dentro da função calcula()

VARIÁVEIS

- Nenhuma relação existe entre uma variável local e uma global, ainda que definidas com o mesmo nome.
- É totalmente desaconselhável a definição de uma variável local com o mesmo nome de uma global, a fim de evitar confusão por parte do programador.
- O programa não faz confusão, porém o programador pode facilmente se confundir.

VARIÁVEIS

- A grande vantagem das variáveis locais é que o espaço de armazenamento destas na memória, só é efetivamente ocupado quando a função é executada.
- Ao final da execução, esse espaço é liberado para a memória livre. Com isso, os programas ficam mais “leves”, ou seja, ocupam menos memória.

PARÂMETROS

- A passagem de parâmetros é uma necessidade para a programação. Ela possibilita o reaproveitamento de código, de maneira a tornar o programa mais limpo, curto e fácil de entender.
- Nada mais é do que a chamada a uma função, estabelecendo valores iniciais dos dados que este subprograma manipula.
- Rotinas com passagem de parâmetros devem ser escritas sempre que um determinado “bloco de código” se repita.

PARÂMETROS

- Quando uma função é definida para receber parâmetros, a chamada destes deve obrigatoriamente ser acompanhada dos valores dos argumentos, que devem ser passados na mesma ordem e na mesma sequência em que foram definidos.
- A mesma quantidade de parâmetros definidos deve ser passada na chamada da função, sob pena de erros de compilação.

PARÂMETROS

- Existem duas formas de passagem de parâmetro para funções:
- Por valor
  - Existe uma cópia dos valores das variáveis utilizadas na chamada da função para os parâmetros e não é ocupado a mesma posição na memória, por isso, modificações feitas nos parâmetros não alteram o valor das variáveis.

EXEMPLO

- Ex:

```
int calcula(int a, int b)
{
        int res;
        a += 8;
        res = a + b;
        return res;
}
```

```
main()
{
        int x, y;
        x = 4;
        y = 2;
        cout << "Resultado: " << calcula(x,
        y);
        getchar();
}
```

PARÂMETROS

- Por referência:
  - Neste tipo de passagem de parâmetro as variáveis ocupam a mesma posição na memória, é passado para a função o endereço da variável utilizada na chamada. Assim sendo, ao se modificar o valor do parâmetro está sendo alterado o valor da variável também.

PARÂMETROS

- Ex:

```
int calcula(int &a, int &b)
{
        int res;
        a += 8;
        res = a + b;
        return res;
}
```

```
main()
{
        int x, y;
        x = 4;
        y = 2;
        cout<<"Resultado: "<<calcula(x, y);
        getchar();
}
```

PARÂMETROS

- Os parâmetros dentro de uma função são considerados variáveis locais, ou seja, só existem dentro dele próprio. Ainda por serem variáveis locais, os parâmetros não necessitam de uma definição prévia na área de definições de variáveis, e também não necessitam ser definidos como variáveis locais, porque a própria definição ao lado do nome da função se encarrega disso.

RETORNO DA FUNÇÃO

- Nos exemplos anteriores a função voltava para o seu chamador quando sua chave de fechamento era encontrada.
- Muitas funções não necessitam chegar até a chave de fechamento para encerrar seu processamento, pois podem ter atingido seu objetivo antes de percorrer e executar todas as linhas, para encerrar e retornar a função podemos utilizar o comando ***return.***

RETORNO DA FUNÇÃO

- O comando *return* pode ser utilizado de duas formas:
  - Uma que retorna um valor em funções que possuem tipo diferente de *void*
  - E a outra que não retorna qualquer valor, mas mesmo assim necessita interromper sua execução naquele momento.

RETORNO DA FUNÇÃO

- Função que retorna um valor:

```
int soma(int n1, int n2);
main()
{
   cout << "Soma: " << soma(5,7);
}
int soma(int n1, int n2)
{
    return n1 + n2;
}
```

RETORNO DA FUNÇÃO

- Uma função que retorna um valor deve ser declarada e especificar o tipo do valor retornado.
- O tipo de retorno deve ser compatível com o tipo de dados usado na instrução *return*.
- Caso o tipo não for compatível será gerado um erro em tempo de compilação do programa.

RETORNO DA FUNÇÃO

- Função que não retorna valor

```
void exp(int b, int e);
main()
{
  exp(2,-4);
}
void exp(int b, int e)
{
   if(e < 0)
   {
      cout << "Nao e permitido expoente negativo";
      return;
   }
   else
      cout << pow(b,e);
}
```

RETORNO DA FUNÇÃO

- Uma função poderá conter diversas vezes a instrução *return*. Assim que uma for encontrada, a função retorna e o restante do código é desprezado.
- No entanto fique atento para o fato de que ter muitos *return* pode desestruturar uma função e confundir seu significado ou mesmo dificultar a execução com determinados valores.

RETORNO DA FUNÇÃO

```
void teste(int x)
{
   switch x
   {
      case 0:
         //instruções
         return;
         break;
      case 1:
         //instruções
         return;
         break;
   };
   if(x > 100)
      return;
   else
      //faz outra operação
}
```

ARGUMENTO DEFAULT

- Argumento *default* é a possibilidade de definir um valor padrão para um argumento em uma função.
- O valor *default* é automaticamente usado quando nenhum argumento correspondente àquele parâmetro é especificado na chamada da função.
- O argumento *default* é especificado no protótipo da função de maneira similar à uma inicialização de variável.

ARGUMENTO DEFAULT

```
void calcular(int x = 0, int i = 10);
main()
{
   calcular();
   cout << "\n";
   calcular(5);
   cout << "\n";
   calcular(2,8);
}
```

```
void calcular(int x, int i)
{
   while(x <= i)
   {
      cout << x << ", ";
      x++;
   }
}
```

SAÍDA:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
5, 6, 7, 8, 9, 10,
2, 3, 4, 5, 6, 7, 8,

ARGUMENTO DEFAULT

- A Primeira chamada não passa nenhum argumento, sendo válidos os argumentos *default* para x e i

  calcular();
- A Segunda chamada define somente valor para um argumento, sendo atribuído ao primeiro parâmetro, no caso x recebe 5 e i fica com o valor *default*

  calcular(5);

ARGUMENTO DEFAULT

- A última chamada define os dois valores, desta forma x recebe 2 e i recebe 8.

```
calcular(2,8);
```

- Os argumentos *default* não podem ser redefinidos dentro de um programa ou em tempo de execução do programa.

ARGUMENTO DEFAULT

- Salienta-se que uma vez que você tenha começado a definir parâmetros que adotam valores *default*, todos os parâmetros devem especificar um valor *default*.

  void calcular(int x = 0, int i);

- Esta declaração gera um erro de compilação, pois como x possui valor *default* i também deveria ter definido um valor *default*.

ARGUMENTO DEFAULT

- Caso em determinada situação alguns argumentos não tenham valores *default* esses devem ser declarados a **esquerda (antes)** dos argumentos que serão declarados com valor *default*:

```
void calcular(int x, int i = 10);
```

- Porém nestes casos a chamada da função sempre deverá conter ao menos um argumento, para ser atribuído a X.

```
calcular(); // chamada inválida
```

VETOR COM FUNÇÃO

- Para manipular um array de uma dimensão (vetor) utilizando uma função devemos criar um parâmetro na função identificando como um vetor através do uso dos colchetes, não necessita informar a dimensão, pois ao acionar a função estaremos referenciando ao endereço de memória do vetor original.
- Exemplo

```
void leitura(int x[])
```

VETOR COM FUNÇÃO

- Para acionar (chamar) a função que vai manipular o vetor devemos informar o endereço de memória onde inicia o vetor, isso pode ser feito passando somente o nome do vetor, ou o endereço dá primeira posição:
- Exemplo:

```
int a[10];
leitura(a);
leitura(&a[0]);
```

VETOR COM FUNÇÃO

```
main(){
   int a[50];
   leitura(a);
}

void leitura(int x[]){
    int i;
    srand(time(NULL));
    for(i = 0; i < 50; i++){
            x[i] = rand() % 50;
    }
}
```

MATRIZ COM FUNÇÃO

- Para manipular um array de duas dimensões (matriz) através de uma função deve-se declarar o parâmetro da função informado o número de colunas que a matriz vai conter.
- Exemplo:

```
void leitura(int mat[][5]);
```

- A chamada deve ser passando somente o nome da variável original, que também estaremos passando o endereço de memória onde inicia a matriz:

```
leitura(a);
```

MATRIZ COM FUNÇÃO

```
main(){
   int a[5][5];
   leitura(a);
}
void leitura(int mat[][5]){
    int i, x;
    srand(time(NULL));
    for(i = 0; i < 5; i++){
        for(x = 0; x < 5; x++){
            mat[i][x] = rand() % 100;
        }
    }
}
```

MATRIZ

- Algumas matrizes possuem características especiais, recebendo denominação própria conforme a característica apresentada.

MATRIZ QUADRADA

- É uma matriz que possui o mesmo número de linhas e colunas, denominada matriz N x N.
- Denomina-se ordem da matriz o número de elementos possíveis de ser inserido em cada linha ou coluna.
- Ex: Matriz A de ordem 2

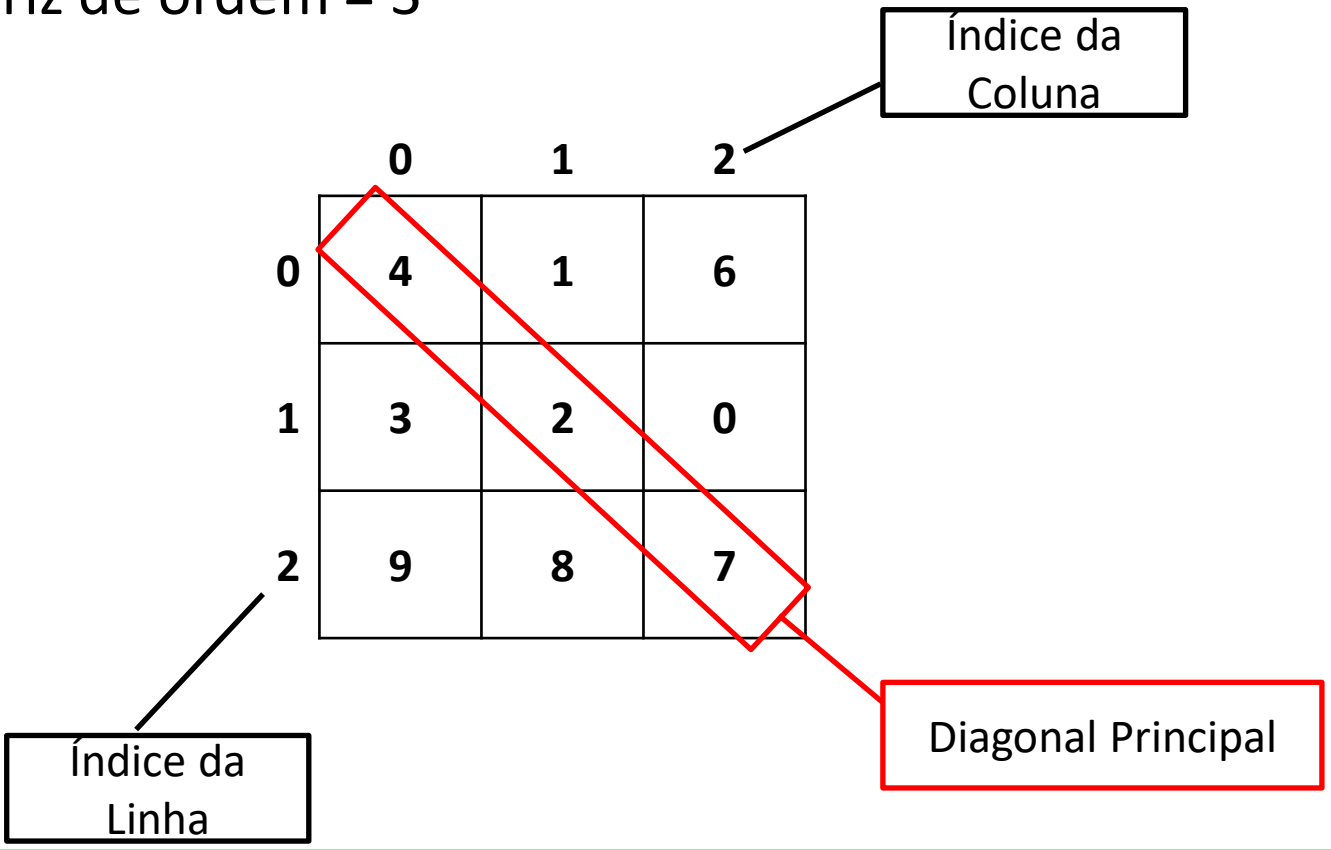
$$A = \begin{bmatrix} 2 & 5 \\ 1 & 9 \end{bmatrix}$$

MATRIZ QUADRADA

- Em uma matriz quadrada alguns elementos estão posicionados em locais com identificação diferenciada, é o caso das diagonais: principal e secundária.
- A diagonal principal (DP) é formada pelos elementos A[L,C] tais que L = C, onde L representa o índice da linha e C o índice da coluna.

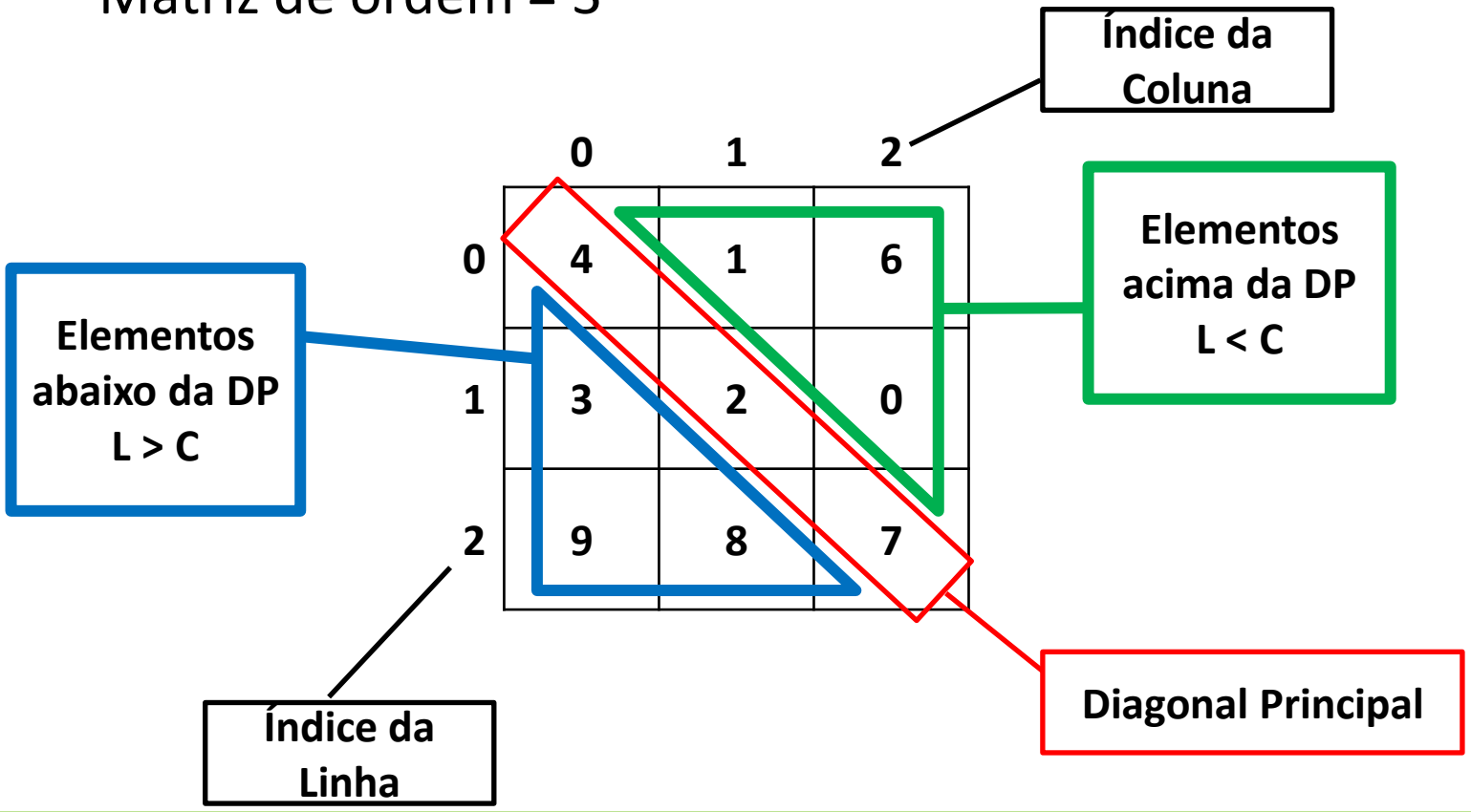
DIAGONAL PRINCIPAL

- L = C
- Matriz de ordem = 3



DIAGONAL PRINCIPAL

- Elementos acima ou abaixo da DP
- Matriz de ordem = 3

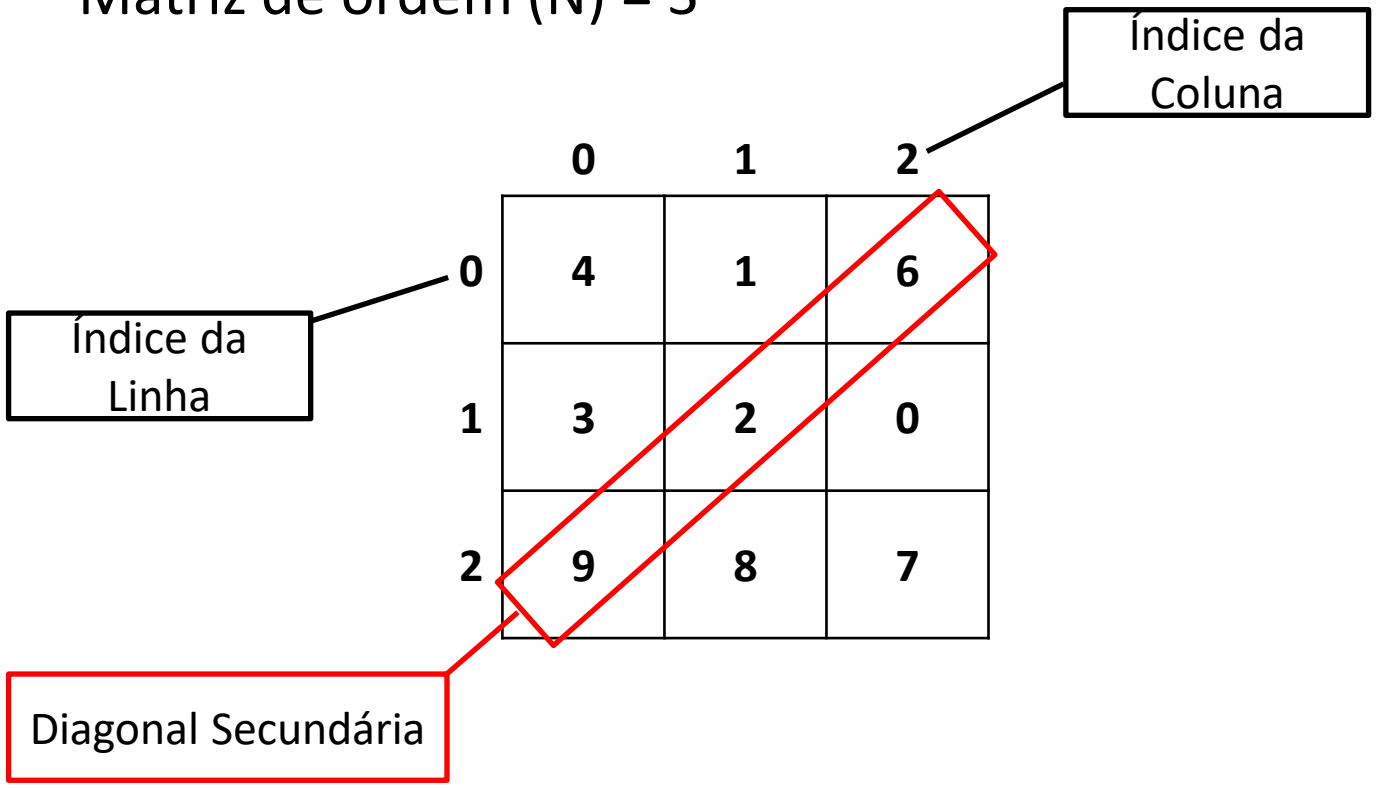


DIAGONAL SECUNDÁRIA

- A diagonal secundária (DS) é formada pelos elementos A[L,C] tais que L + C = N - 1, onde L representa o índice da linha, C o índice da coluna e N a ordem da matriz.

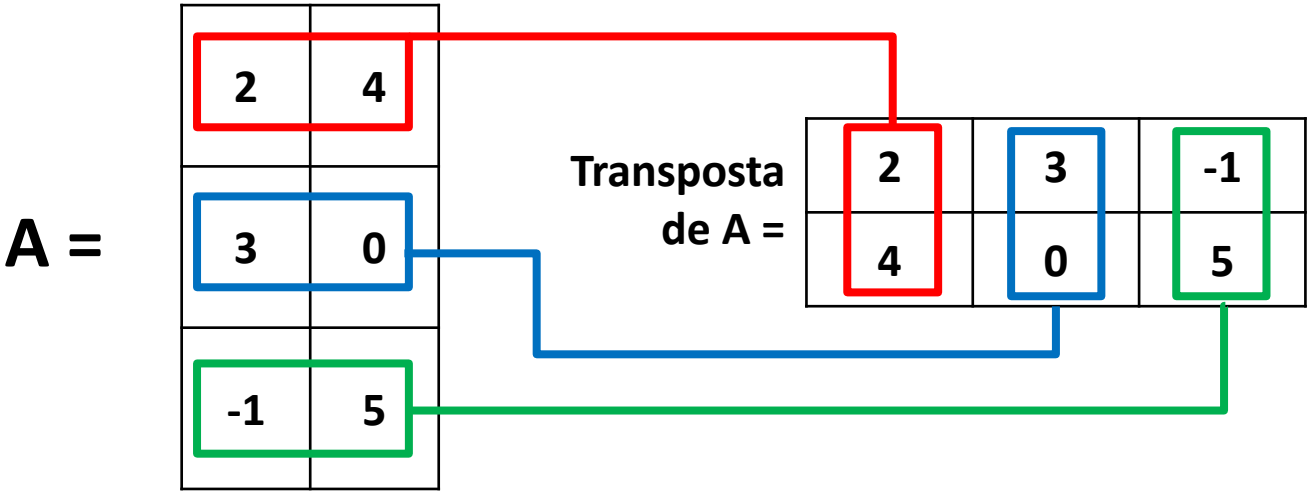
DIAGONAL SECUNDÁRIA

- L + C = N - 1
- Matriz de ordem (N) = 3



MATRIZ TRANSPOSTA

- Matriz Transposta é obtida a partir da troca ordenada das linhas de uma Matriz pelas colunas ou as colunas por linhas.

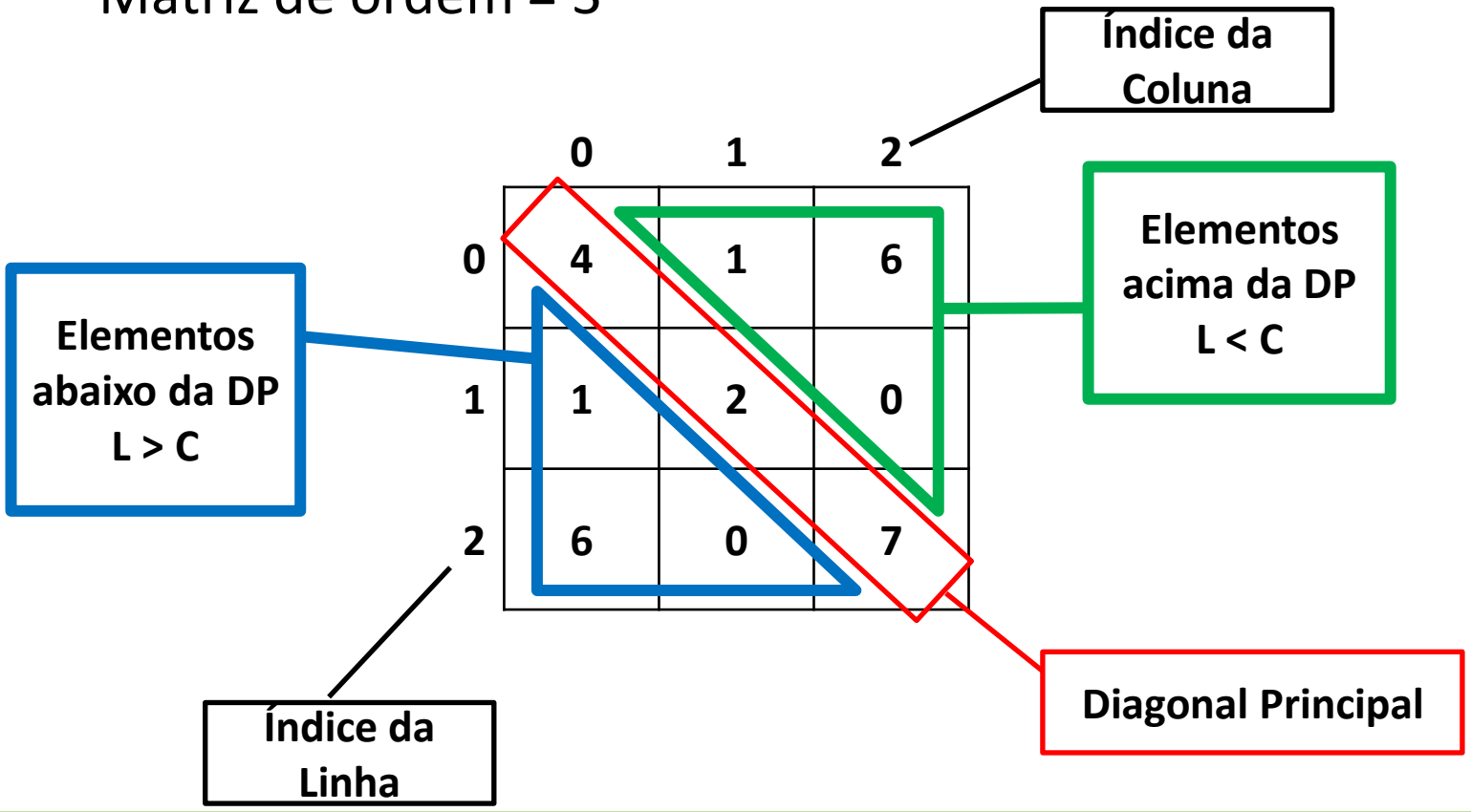


MATRIZ SIMÉTRICA

- Matriz simétrica é a matriz quadrada onde A[L,C] = A[C,L], para todo par (L,C), ou seja, os elementos que estão acima da diagonal principal são iguais aos respectivos elementos que estão abaixo da diagonal principal.

MATRIZ SIMÉTRICA

- A[L,C] = A[C,L]
- Matriz de ordem = 3

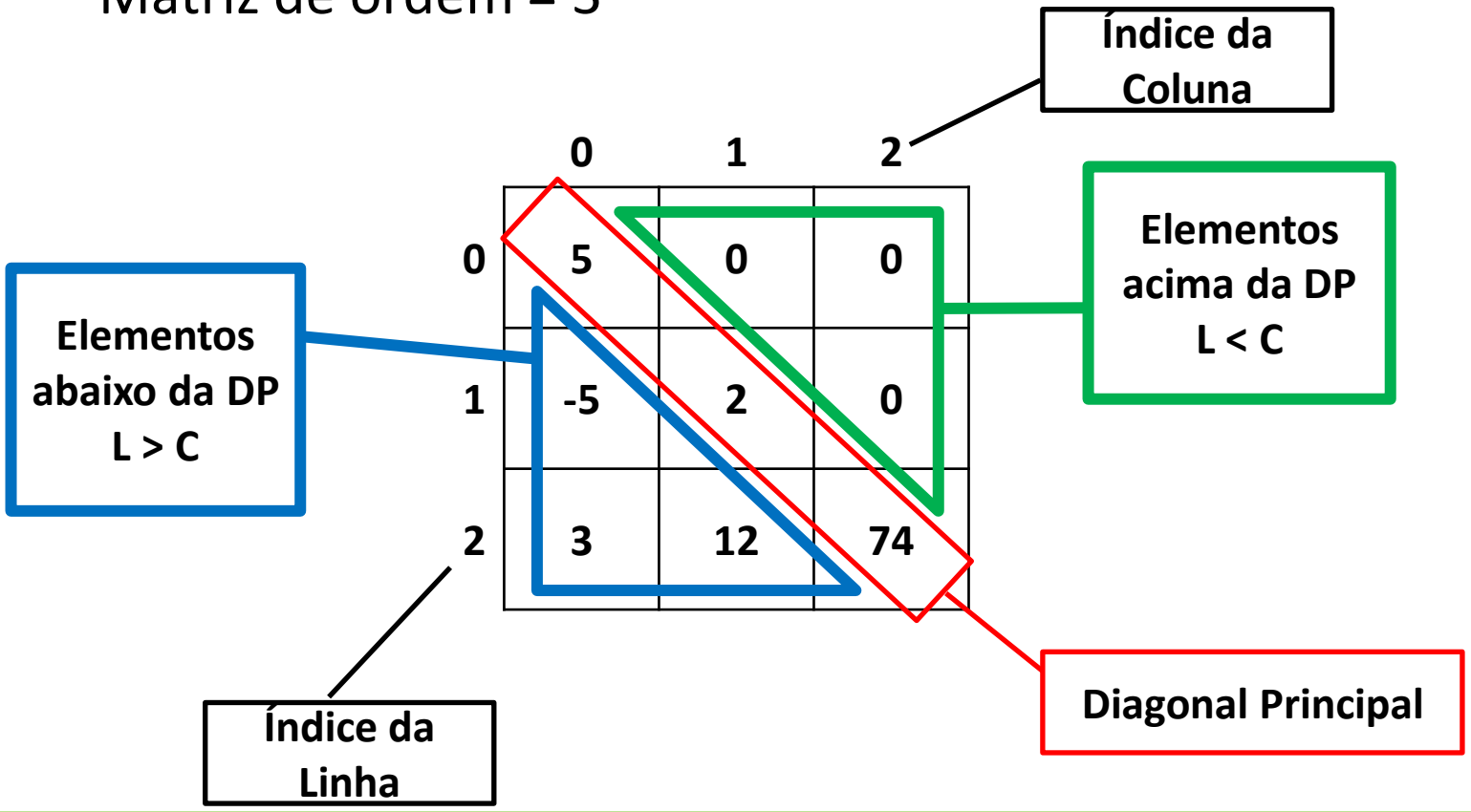


MATRIZ TRIANGULAR

- Uma matriz quadrada pode ser triangular inferior ou triangular superior.
- É triangular inferior quando A[L,C] = 0 para todo elemento L < C, ou seja, todos os elementos acima da diagonal principal são nulos.

MATRIZ TRIÂNGULAR INFERIOR

- SE L < C e A[L,C] = 0
- Matriz de ordem = 3

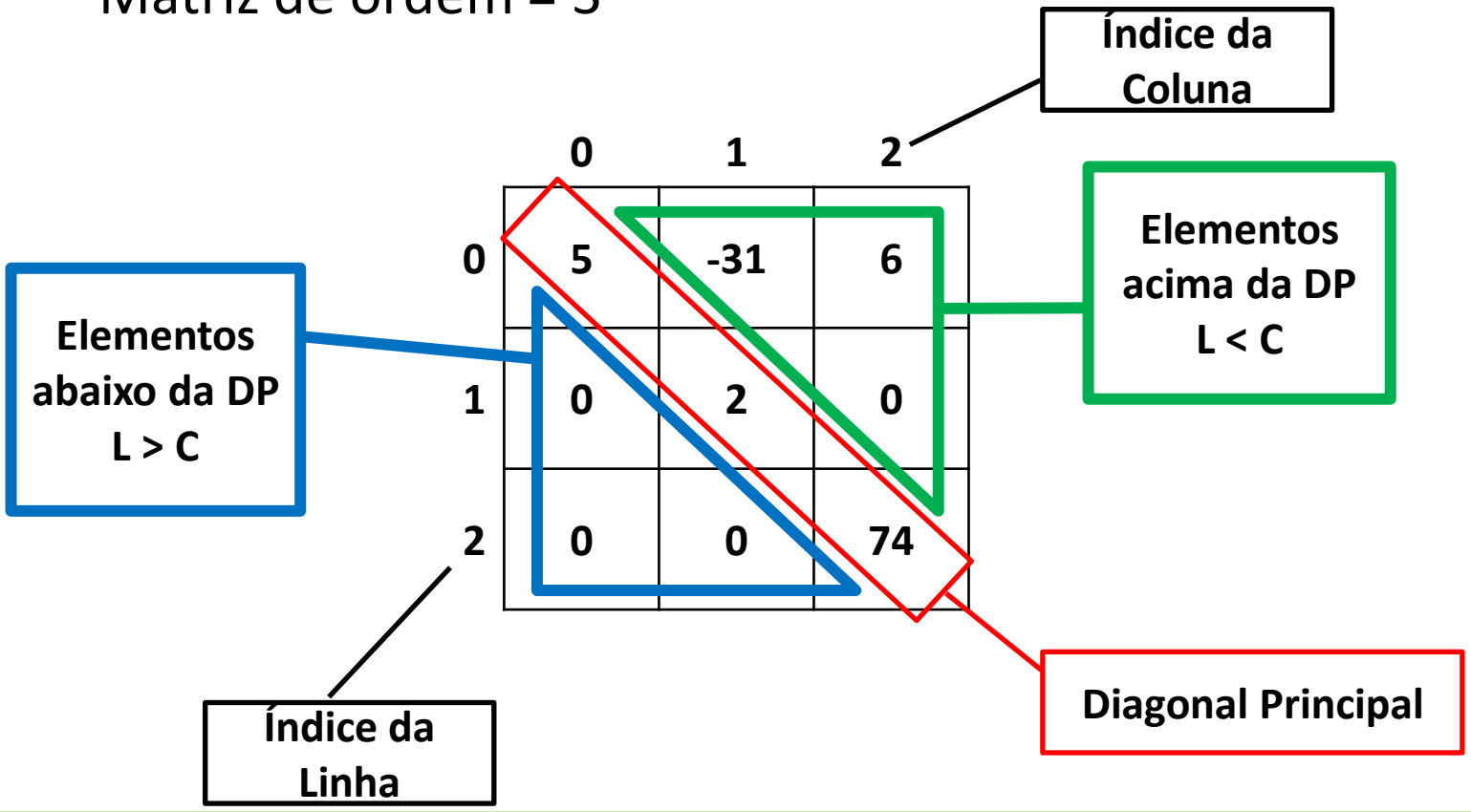


MATRIZ TRIÂNGULAR SUPERIOR

- Uma Matriz quadrada é considerada triangular superior quando A[L,C] = 0 para todo L > C, ou seja, todos os elementos abaixo da diagonal principal são nulos.

MATRIZ TRIÂNGULAR SUPERIOR

- SE L > C e A[L,C] = 0
- Matriz de ordem = 3

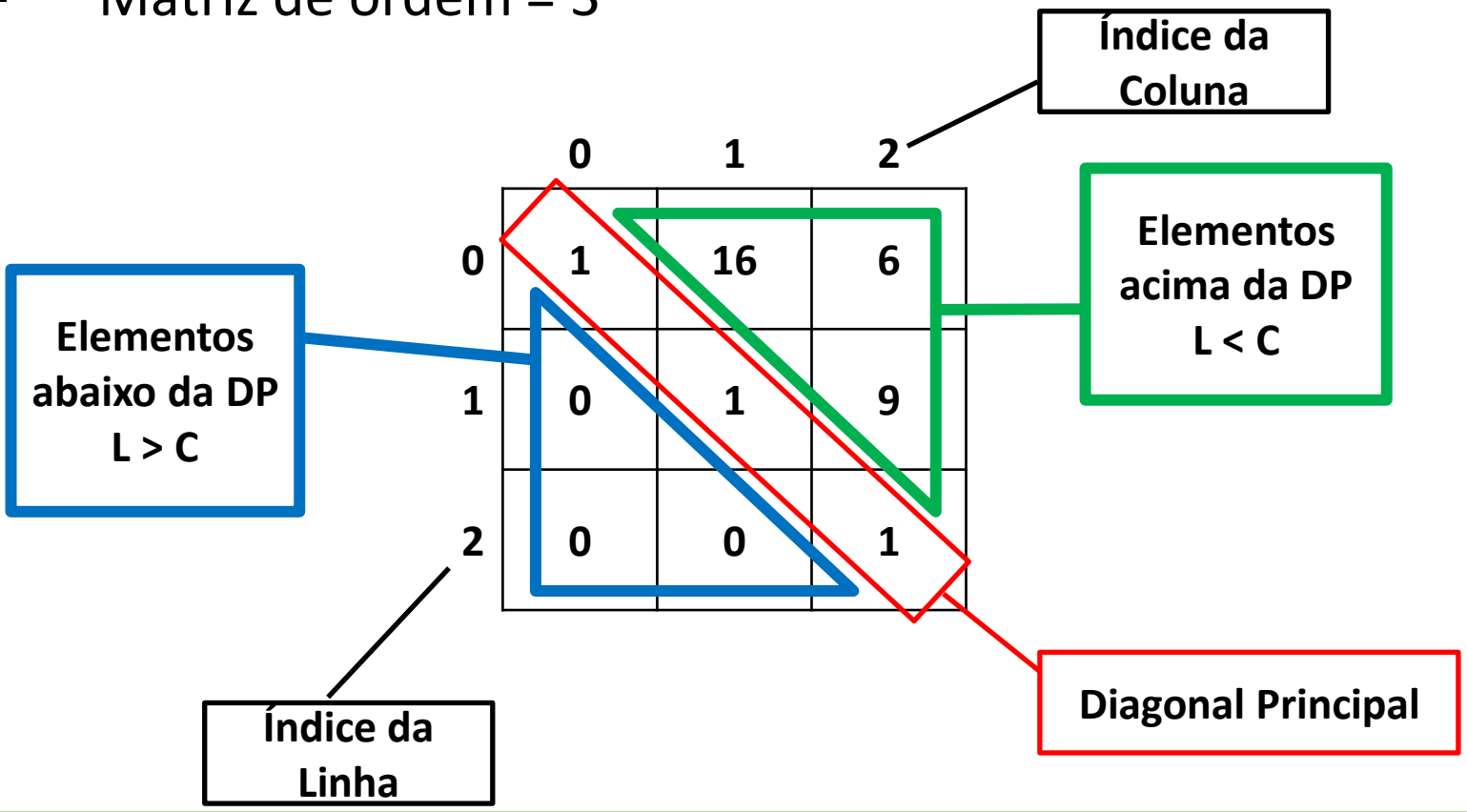


MATRIZ TRIÂNGULAR UNITÁRIA

- Uma Matriz quadrada triangular superior ou inferior é considerada unitária quando além de atender os requisitos para ser triangular, todos os elementos da diagonal principal são iguais a 1.

MATRIZ TRIÂNGULAR SUPERIOR UNITÁRIA

- Se L > C e A[L,C] = 0 e também L = C e A[L,C] = 1
- Matriz de ordem = 3



REFERÊNCIAS

- REBONATTO, Marcelo T. – Notas de Aula.
- ARAÚJO, Jáiro – Dominando a Linguagem C. Editora Ciência Moderna.
- PEREIRA, Silvio do Lago. Estrutura de Dados Fundamentais: Conceitos e Aplicações, 12. Ed. São Paulo, Érica, 2008.
- LORENZI, Fabiana. MATTOS, Patrícia Noll de. CARVALHO, Tanisi Pereira de. Estrutura de Dados. São Paulo: Ed. Thomson Learning, 2007.
- VELOSO, Paulo. SANTOS, Celso dos. AZEVEDO, Paulo. FURTADO, Antonio. Estrutura de dados. Rio de Janeiro: Ed. Elsevier, 1983 27ª reimpressão.