Universidade Eduardo Mondlane
Faculdade de Engenharia
Departamento de Engenharia Eletrotécnica
Engenharia Informática

Estrutura de Dados e Algoritmos – EDA

Por:

- **Dr. Alfredo Covele**
- **Eng. Cristiliano Maculuve**

Dezembro 2023

Tópicos

No.	Designação
1	**Complexidade de Algoritmos** ❑ Complexidade de tempo ❑ Interpretação da complexidade de tempo ❑ Cálculo da complexidade de tempo

Introdução

O estudo de Estruturas de Dados é basicamente o **estudo de métodos de organização de grandes quantidades de dados**. Estes métodos incluem:

- formas de organizá-los (estruturas);
- técnicas para manipulá-los (algoritmos básicos).

Introdução

- Pilhas, Listas e Filas são estruturas de dados típicas;
- para manipular estas estruturas - inserir elementos, retirar elementos e ordenar elementos - necessitamos de algoritmos;
- estes algoritmos podem ser implementados de muitas maneiras. Algumas simples e diretas, outras não tão simples - porém engenhosas - e outras ainda complicadas e envolvendo muitas operações;
- quando trabalhamos com quantidades muito grandes de dados, um projeto ruim de algoritmo para manipular uma estrutura de dados pode resultar em um programa que apresenta um tempo de execução inviável.

Introdução

Quando estudamos algoritmos, um aspecto que devemos considerar, além de sua correção, é a **análise da sua eficiência**.

A **análise de algoritmos** pode ser definida como o ***estudo da estimativa do tempo de execução dos algoritmos*** (M. A. Weiss).

Idéias básicas

- Um algoritmo é um conjunto finito de passos que devem ser seguidos com um conjunto de dados de entrada para se chegar à solução de um problema;
- um problema, geralmente, pode ser resolvido por muitos algoritmos diferentes;
- exemplo: cálculo da potência x^n de um número inteiro x.

Idéias básicas

```
16      public int calcularPotencia(int x,int n){
17          int y=1;
18          while (n>0) {
19              y=y*x;
20              n=n-1;
21          }
22          return y;
23      }

25      public int calcularPotencia1(int x, int n) {
26          return x ^ n;
27      }
```

Idéias básicas

O fato de existir um algoritmo para resolver um problema não implica necessariamente que este problema possa realmente ser resolvido na prática. Há restrições de tempo e de espaço de memória.

Exemplo: calcular todas as possíveis partidas de xadrez.

Idéias básicas

Um algoritmo é uma idéia abstrata de como resolver um determinado problema. Ele é, a princípio, independente da máquina que o executará e de suas características.

Um programa é uma implementação de um algoritmo em uma linguagem particular que será executado em um computador particular.

Um programa está sujeito às limitações físicas da máquina onde será executado, como capacidade de memória, velocidade do processador e dos periféricos, entre outras.

Idéias básicas

O tempo que a execução de um programa toma é uma grandeza física que depende:

- do tempo que a máquina leva para executar uma instrução ou um passo de programa;
- da natureza do algoritmo, isto é, de quantos passos são necessários para se resolver o problema para um dado;
- do tamanho do conjunto de dados que constitui o problema.

Problema básico na Análise de Algoritmos

Necessitamos definir uma forma de criar uma medida de comparação entre diferentes algoritmos que resolvem um mesmo problema, para:

- podermos saber se são viáveis;
- podermos saber qual é o melhor algoritmo para a solução do problema.

Problema básico na Análise de Algoritmos

Para fazermos isso, **abstraímos de um computador em particular** e assumimos que a execução de todo e qualquer passo de um algoritmo leva um tempo fixo e igual a uma unidade de tempo:

- o tempo de execução em um computador particular não é interessante;
- muito mais interessante é uma comparação relativa entre algoritmos.

Problema básico na Análise de Algoritmos

Modelo de computação

- as operações são todas executadas sequencialmente;
- a execução de toda e qualquer operação toma uma unidade de tempo;
- a memória do computador é infinita.

Assim nos sobram duas grandezas:

- tempo = número de operações executadas;
- quantidade de dados de entrada.

Complexidade de Tempo

Podemos expressar de forma abstrata a eficiência de um algoritmo descrevendo o seu tempo de execução como uma função do tamanho do problema (quantidade de dados). Isto é chamado de complexidade de tempo.

Exemplo: ordenação de um Vetor.

Primeiro caso: Bubblesort

Bubblesort é o mais primitivo dos métodos de ordenação de um vetor. A idéia é percorrer um vetor de *n* posições *n* vezes, a cada vez comparando dois elementos e trocando-os caso o primeiro seja maior que o segundo.

Primeiro caso: Bubblesort

```
25      public void bubbleSort(int[] array) {
26          int n = array.length;
            int temp = 0;
28          for (int i = 0; i < n; i++) {
29              for (int j = 1; j < (n - i); j++) {
30                  if (array[j - 1] > array[j]) {
31                      temp = array[j - 1];
32                      array[j - 1] = array[j];
33                      array[j] = temp;
34
35                  }
36              }
37          }
38      }
```

Primeiro caso: Bubblesort

A comparação **(**array**[j-1] >** array**[j])** vai ser executada **n*(n-1)** vezes. No caso de um vector na ordem inversa, as operações da atribuição triangular poderão ser executadas até 3*n*(n-1) vezes, já que uma troca de elementos não significa que um dos elementos trocados tenha encontrado o seu lugar definitivo.

Segundo caso: StraightSelection

O método da seleção direta é uma forma intuitiva de ordenarmos um vetor: escolhemos o menor elemento do vetor e o trocamos de posição com o primeiro elemento.

Depois começamos do segundo e escolhemos novamente o menor dentre os restantes e o trocamos de posição com o segundo e assim por diante.

Segundo caso: StraightSelection

```
StraightSelection(a[], n)
  inteiro i, j, k, x;
      início
            para i de 1 até n-1 faça
                  k <- i;
                  x <- a[i];
                  para j de i+1 até n faça
                        se (a[j] < x) então
                              k <- j;
                              x <- a[k];
                        fim se
                  fim para
                  a[k] <- a[i];
                  a[i] <- x;
            fim para
      fim
```

Segundo caso: StraightSelection

Neste algoritmo o número de vezes que a comparação **(a[j] < x)** é executada é expresso por (n-1)+(n-2)+...+2+1 = (n/2)*(n-1).

O número de trocas **a[k] <- a[i]; a[i] <- x** é realizado no pior caso, onde o vetor está ordenado em ordem inversa, somente n-1 vezes, num total de 2*(n-1).

Interpretação

Como já foi dito, a única forma de se poder comparar dois algoritmos é descrevendo o seu comportamento temporal em função do tamanho do conjunto de dados de entrada. Assim:

$T_{algoritmo} = f(n)$, onde *n* é o tamanho do conjunto de dados.

Interpretação

Se tomarmos as operações de troca de valores como critério-base, podemos dizer que:

$T_{Bubblesort}$ = **3*n*(n-1)** sempre

$T_{StraightSelection}$ = **2*(n-1)** para o pior caso

Interpretação

O que nos interessa é o comportamento assintótico de f(n), ou seja, como f(n) varia com a variação de n.

Razão: para mim é interessante saber como o algoritmo se comporta com uma quantidade de dados realística para o meu problema e o que acontece quando eu vario esses dados.

Interpretação

Exemplo: eu tenho dois algoritmos (*a* e *b*) para a solução de um problema. Se a complexidade de um é expressa por $f_a(n) = n^2$ e a do outro por $f_b(n) = 100*n$, significa que o algoritmo *a* cresce quadraticamente (uma parábola) e que o algoritmo *b* cresce linearmente (embora seja uma reta bem inclinada).

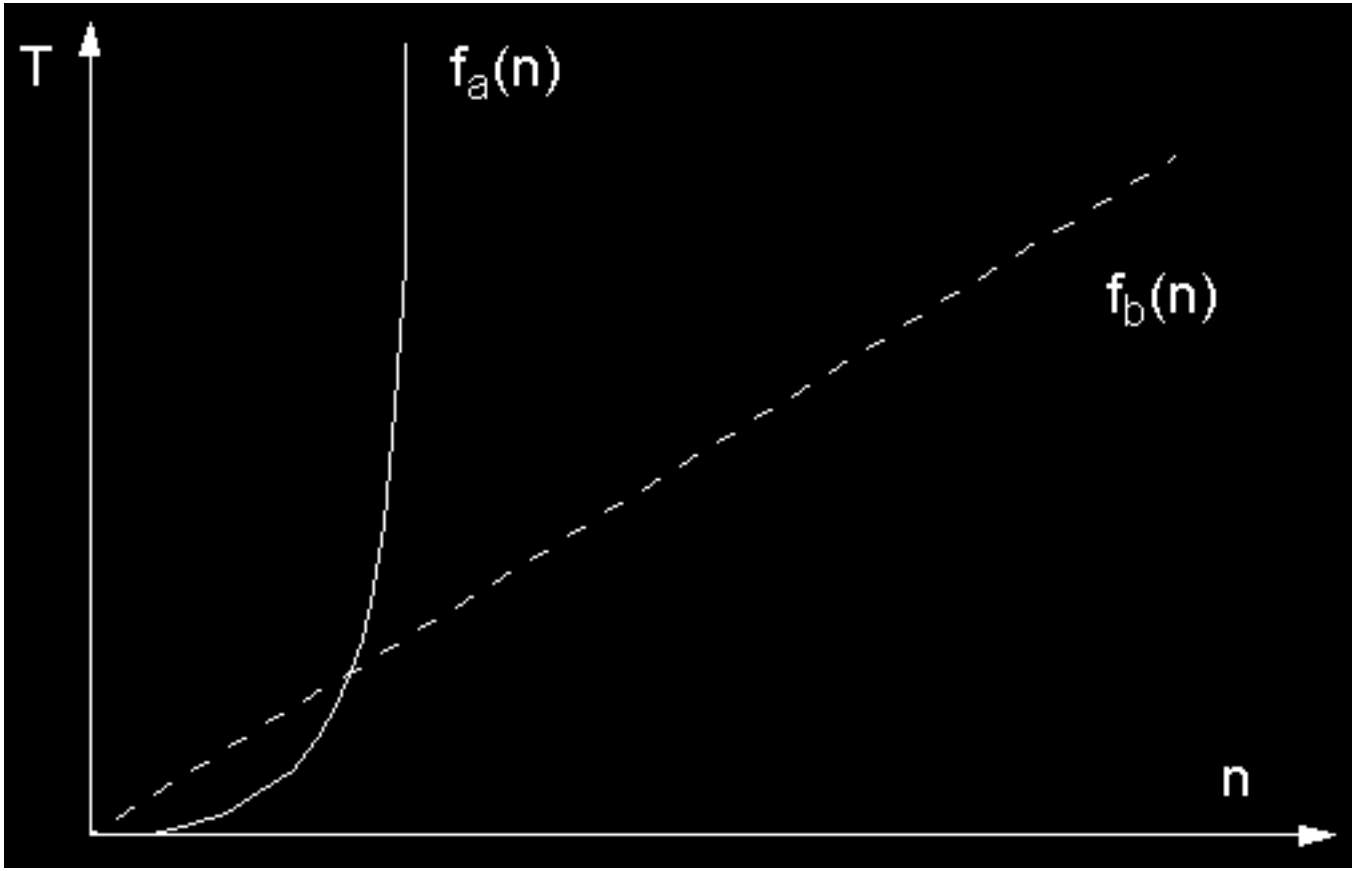
Interpretação

Se eu uso estes algoritmos para um conjunto de 30 dados, o segundo com T_b=3.000 é pior do que o primeiro com T_a=900.

Se eu os uso para um conjunto de 30.000 dados, porém, terei T_a=900.000.000 e T_b=3.000.000.

Isto ocorre porque o **comportamento assintótico** dos dois é bem diferente.

Interpretação



Interpretação da Complexidade de Tempo

Aspecto essencial que deve ser expresso pelo cálculo de complexidade:

Qual é o comportamento assintótico ***predominante*** *de um algoritmo em função do* ***tamanho*** *do conjunto de dados a ser processado.*

Exemplo: se é linear, polinomial (quadrático, cúbico, etc.), logarítmico ou exponencial.

Análise Assintótica

Para a análise do comportamento de algoritmos existe toda uma terminologia própria.

Para o cálculo do comportamento de algoritmos foram desenvolvidas diferentes medidas de complexidade.

A mais importante delas e que é usada na prática é chamada de **Ordem de Complexidade** ou **Notação-O** ou ***Big-Oh***.

Análise Assintótica

Definição (*Big-Oh*): **T(n) = O(f(n))** se existem constantes c e n_0 tais que T(n)c.f(n) quando nn$_0$.

A definição indica que existe uma constante c que faz com que c.f(n) seja sempre pelo menos tão grande quanto T(n), desde que n seja maior que um n_0.

Em outras palavras: a Notação-O fornece a Ordem de Complexidade ou a Taxa de Crescimento de uma função.

Para isso, não consideramos os termos de ordem inferior da complexidade de um algoritmo, apenas o termo predominante.

Análise Assintótica

Exemplo: algoritmo com complexidade **$T(n) = 3n^2 + 100n$**.

Nesta função, o segundo termo tem um peso relativamente grande, mas a partir de $n_0 = 11$, é o termo n^2 que "dá o tom" do crescimento da função: ***uma parábola***. A constante 3 também tem uma influência irrelevante sobre a taxa de crescimento da função após um certo tempo.

Por isso dizemos que este algoritmo é da ordem de n^2 ou que tem complexidade **$O(n^2)$**.

Diferentes Tempos de Execução

Problema da Subsequência de Soma Máxima: dada uma sequência de números $a_1, a_2, \ldots, a_n$, positivos ou negativos, encontre uma subsequência $a_j, \ldots, a_k$ dentro desta sequência cuja soma seja máxima. A soma de uma sequência contendo só números negativos é por definição 0.

Exemplo: para a sequência **-2, 11, -4, 13, -5, -2,** a resposta é **20 (a_2,a_3,a_4)**.

Este problema oferece um bom exemplo para o estudo de como diferentes algoritmos que resolvem o mesmo problema possuem diferentes comportamentos, pois para ele existem muitas soluções diferentes.

Cálculo da Complexidade de Tempo

```
16      public int somaCubos(int val) {
17          int soma = 0;
18          for (int i = 1; i <= val; i++) {
19              soma = soma + i * i * i;
20          }
21          return soma;
22      }
```

Cálculo da Complexidade de Tempo

Análise:

- as declarações não tomam tempo nenhum;
- a linha 20 também não toma tempo nenhum;
- as linhas 17 e 21 contam uma unidade de tempo cada;
- a linha 19 conta 4 unidades de tempo (2 multiplicações, uma adição e uma atribuição) e é executada *n* vezes, contando com um total de 4*n* unidades de tempo;
- a linha 18 possui custos implícitos de inicializar i, testar se é menor que *n* e incrementá-lo. Contamos 1 unidade para sua inicialização, *n* + 1 para todos os testes e *n* para todos os incrementos, o que perfaz 2*n* + 2 unidades de tempo;
- o total perfaz 6*n* + 4 unidades de tempo, o que indica que o algoritmo é O(*n*), da Ordem de Complexidade *n*, ou seja, linear.

Regras para o cálculo

Laços: o tempo de execução de um laço é, no máximo, a soma dos tempos de execução de todas as instruções dentro do laço (incluindo todos os testes) multiplicado pelo número de iterações.

Laços aninhados: analise-os de dentro para fora. O tempo total de execução de uma instrução dentro de um grupo de laços aninhados é o tempo de execução da instrução multiplicado pelo produto dos tamanhos de todos os laços.

Regras para o cálculo

Instruções Consecutivas: estes simplesmente somam, sendo os termos de ordem menor da soma ignorados.

Exemplo: O(n) + O(n^2) = O(n^2)

```
public void instrucoesConsecutivas(int val) {
    int soma = 0;

    for (int j = 1; j <= val; j++) {
        soma += j;
    }

    for (int j = 1; j <= val; j++) {
        for (int k = 1; k <= val; k++) {
            soma += k;
        }
    }
}
```

Regras para o cálculo

Se / Então / Senão: considere o fragmento de código abaixo.

```
se cond então
    expresssão1
senão
    expressão2
fim se
```

O tempo de execução de um comando Se / Então / Senão nunca é maior do que o tempo de execução do teste **cond** em si mais o tempo de execução da maior dentre as expressões **expressão**1 e **expressão**2. Ou seja: se **expressão**1 é $O(n^3)$ e **expressão**2 é $O(n)$, então o teste é $O(n^3) + 1 = O(n^3)$.

Regras para o cálculo

Chamada a Funções: segue a mesma regra dos laços aninhados - analise tudo de dentro para fora. Ou seja: para calcular a complexidade de um programa com várias funções, calcula-se primeiro a complexidade de cada uma das funções e depois considera-se cada uma das funções como uma instrução com a complexidade de função.

Regras para o cálculo

Recursão: é a parte mais difícil da análise de complexidade. Na verdade existem dois casos: muitos algoritmos recursivos mais simples podem ser "linearizados", substituindo-se a chamada recursiva por alguns laços aninhados ou por uma outra subrotina extra e eventualmente uma pilha para controlá-la. Nestes casos, o cálculo é simples e pode ser feito depois da "linearização". Em muitos algoritmos recursivos, porém, isto não é possível. Nestes casos obtemos uma relação de recorrência que tem de ser resolvida e é uma tarefa matemática menos trivial.

Regras para o cálculo

Exemplo de cálculo de complexidade em recursão: Fatorial

```
public int factorialCalculo(int val){
    if(val==0)
        return 1;
    return val*factorialCalculo(val-1);
}
```

Regras para o cálculo

O exemplo anterior é realmente um exemplo pobre de recursão e pode ser "iterativisado" de forma extremamente simples com apenas um laço **for**:

```
public int factorial2(int n) {
    int fact = 1;
    for (int i = 2; i < n; i++) {
        fact=fact*i;
    }
    return fact;
}
```

A complexidade de **Fatorial** pode então ser facilmente calculada e é evidente que é **O(*n*)**.

Regras para o cálculo

O caso dos números de Fibonacci abaixo não é tão simples e requer a resolução de uma relação de recorrência:

```
static long fibonacci(int n) {
    if (n < 2) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Observando o algoritmo, vemos que para n >= 2 temos um tempo de execução T(n) = T(n-1)+T(n-2)+2. A resolução desta relação nos mostra que Fibonacci é O(n).

Logaritmos e outros Tempos de Execução

O aspecto mais complexo na análise de complexidade centra-se em torno do logaritmo. Para analisar-se um algoritmo de complexidade logarítmica e chegar-se a um resultado correto sobre a sua ordem exata de complexidade é necessária uma certa experiência e algum "jeito" matemático. Algumas regras de aproximação podem ser dadas: algoritmos seguindo a técnica Dividir-Para-Conquistar são muitas vezes ***n log n***.

Logaritmos e outros Tempos de Execução

- Quando um algoritmo, em uma passada de uma iteração toma o conjunto de dados e o divide em duas ou mais partes, sendo cada uma dessas partes processada separada e recursivamente, este algoritmo utiliza a técnica dividir-para-conquistar e será possivelmente ***n log n***;
- um algoritmo é ***log n*** se ele toma um tempo constante O(*1*) para dividir o tamanho do problema, usualmente pela metade;
- a pesquisa binária é um exemplo de ***log n***.

Logaritmos e outros Tempos de Execução

Se o algoritmo toma tempo constante para reduzir o tamanho do problema em um tamanho constante, ele será O(*n*).

Algoritmos combinatórios são exponenciais: se um algoritmo testa todas as combinações de alguma coisa, ele será exponencial.

Exemplo: Problema do Caixeiro Viajante

Checando a sua análise

- Uma vez que a análise de complexidade tenha sido executada, é interessante verificar-se se a resposta está correta e é tão boa quanto possível.
- Uma forma de se fazer isto é o procedimento pouco matemático de se codificar o trecho de algoritmo cuja complexidade se tentou descrever e verificar se o tempo de execução coincide com o tempo previsto pela análise.
- Quando *n* dobra, o tempo de execução se eleva de um fator 2 para algoritmos lineares, fator 4 para quadráticos e fator 8 para cúbicos.

Checando a sua análise

Programas logarítmicos só aumentam o seu tempo de execução de uma constante a cada vez que *n* dobra. Algoritmos de O(*n log n*) tomam um pouco mais do que o dobro do tempo sob as mesmas circunstâncias.

Estes aumentos podem ser difíceis de se detectar se os termos de ordem inferior têm coeficientes relativamente grandes e *n* não é grande o suficiente. Um exemplo é o pulo no tempo de execução de *n*=10 para *n*=100 em algumas implementações do problema da subseqüência de soma máxima.

Checando a sua análise

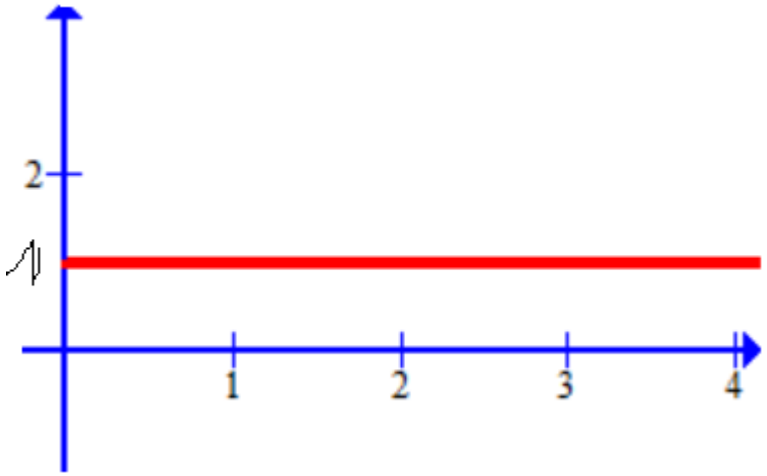
Distinguir programas *n log n* de programas lineares só em evidência de tempo de execução pode também ser muito difícil.

Uma boa pratica é, caso o programa não seja exponencial (o que você vai descobrir muito rápido), fazer alguns experimentos com conjuntos maiores de dados de entrada.

Resumindo

```
40      public void imprimirNumeros(int num){
41          System.out.println("O número é " + num);
42      }
```

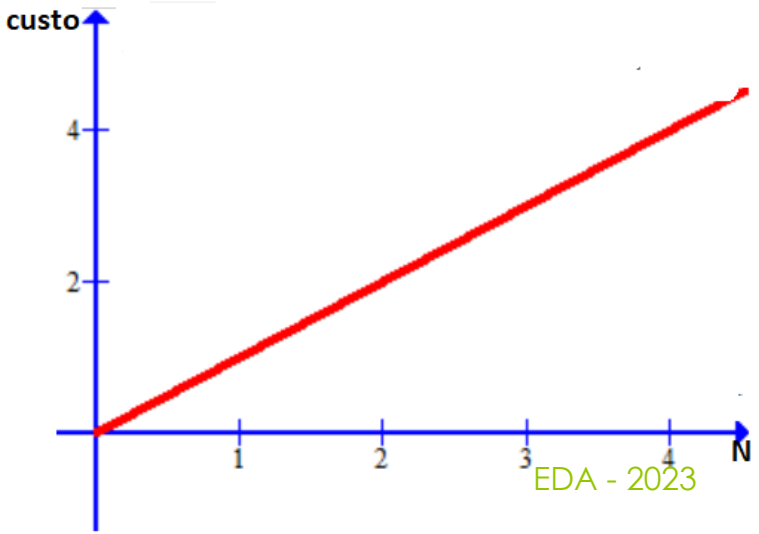
Este algoritmo tem o mesmo custo computacional independentemente da entrada. **O(1)**



Resumindo

```
public void imprimirNumeros(int num){
    for(int i=1;i<=num;i++){
        System.out.println(i);
    }
    System.out.println("O número é " + num);
}
```

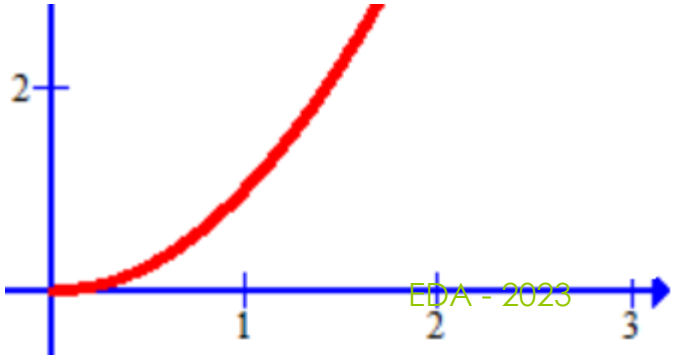
O custo deste algoritmo depende da entrada devido oa laço **O(N)**



Resumindo

```
public void imprimirNumeros(int num) {
    for (int i = 1; i <= num; i++) {
        String linha = "";
        for (int j = 1; j <= num; j++) {
            linha = linha + " " + num;
        }
        System.out.println(linha);
    }
    System.out.println("O número é " + num);
}
```

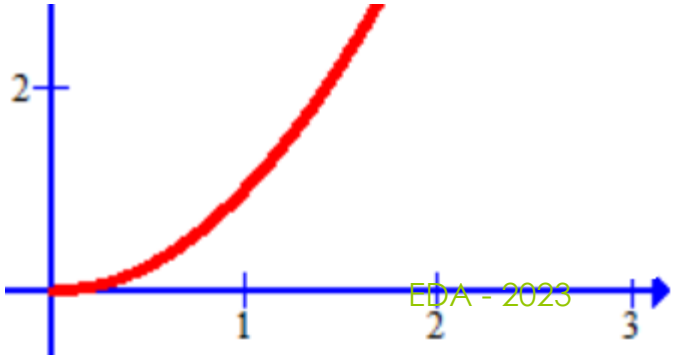
- custo deste algoritmo sera de $\mathbf{O(N^2)}$



Resumindo

```
public void imprimirNumeros(int num) {
    for (int i = 1; i <= num; i++) {
        String linha = "";
        for (int j = 1; j <= num; j++) {
            linha = linha + " " + num;
        }
        System.out.println(linha);
    }
    System.out.println("O número é " + num);
}
```

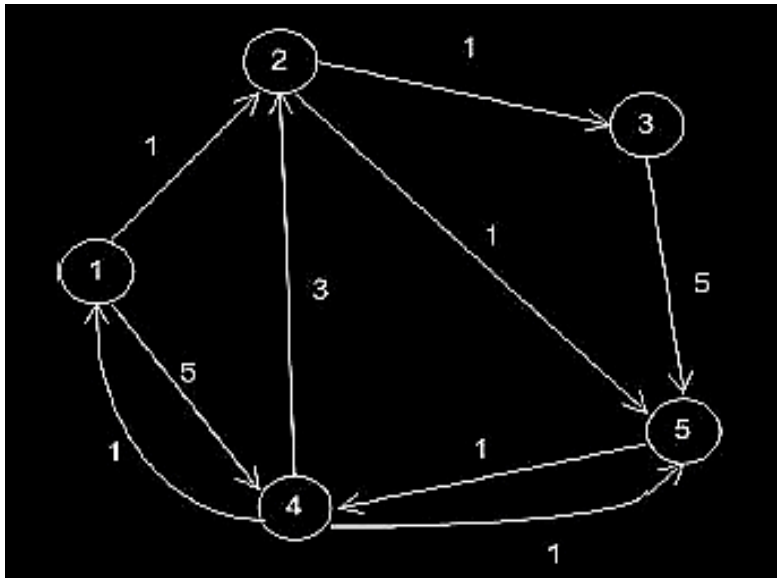
- custo deste algoritmo sera de $\mathbf{O(N^2)}$



Exercício: cálculo de complexidade

1. Floyd

Considere o grafo e sua matriz de custos D, abaixo:



$$D = \begin{bmatrix} 0 & 1 & \infty & 5 & \infty \\ \infty & 0 & 1 & \infty & 1 \\ \infty & \infty & 0 & \infty & 5 \\ 1 & 3 & \infty & 0 & 1 \\ \infty & \infty & \infty & 1 & 0 \end{bmatrix}$$

Exercício: cálculo de complexidade

```
public void floyd(int n, int[][] W, int[][] P, int[][] D) {
    D = W;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            P[i][j] = 0;
        }
    }
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (D[i][k] + D[k][j] < D[i][j]) {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = k;
                }
            }
        }
    }
}
```

Exercício: cálculo de complexidade

2. Torres de Hanoi

2.1. Considere o Problema das Torres de Hanoi com 3 Pinos na sua versão iterativa:

- É um programa que itera sempre sobre duas jogadas: a menor peça e outra possível de ser movida
- Você viu que para 2 discos, houve 3 movimentações de disco e para 3 discos o algoritmo executou 7 movimentações.
- Qual é essa tendência ? Será que esse comportamento continuará seguindo essa tendência?
- Calcule a complexidade do algoritmo das torres de Hanoi

Pesquisar

- **Big O (O())** descreve o **limite superior** da complexidade.
- **Omega (Ω())** descreve o **limite inferior** da complexidade.
- **Theta (Θ())** descreve o **limite exato** da complexidade.
- **Little O (o())** descreve o **limite superior excluindo o limite exato**.

Por hoje é tudo