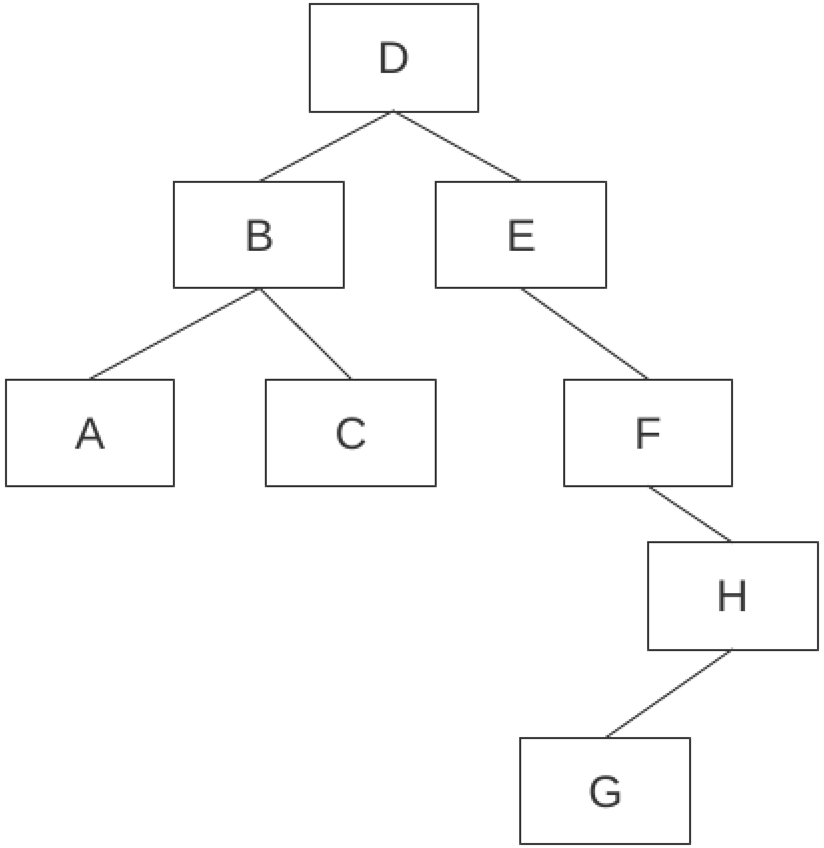
What we need is not the will to believe but the will to find out.
Bertrand Russell

Engenharia Informática

3º ano, 1º semestre



Como vai funcionar a Unidade Curricular?

- **Aulas Teóricas:**
  - Prof. Jorge Barreiros
- **Aulas práticas:**
  - Prof. Jorge Barreiros
  - Prof. Mateus Mendes (mmendes@isec.pt)
- **Horário de atendimento**
- https://videoconf-colibri.zoom.us/j/933906891
- Horário de atendimento
  - Segunda 10:30 – 12:30
  - Quinta 9:00 – 13:00

O que vai ser estudado em Estruturas de Dados?

- **Diferentes formas de representar dados**
  - Pilhas
  - Filas
  - Árvores
  - Tabelas
  - ….
- **Comportamento de diferentes algoritmos, principalmente ao realizar operações *demoradas* sobre essas mesmas estruturas (e em geral)**

Nas aulas práticas...

- **Resolução de fichas de trabalho, em que são propostos exercícios que demonstram o comportamento de determinados algoritmos**
  - Normalmente usa-se Java para implementação
- **Avaliação (a confirmar):**
  - Exame: 10 Valores
  - Seminário: 1.5 Valores
  - Testes Laboratoriais: 8.5 Valores

- **A análise de complexidade é**
  - Independente do hardware
  - Focada no número de operações relevantes para resolver o problema, e principalmente das **operações dominantes**
  - **Operação dominante** é a que demora mais tempo a executar

```
soma = 0; // Esta inicialização realiza-se uma vez

for ( j=0; j < n; j++)
    soma += tab[j]; // O que está dentro do for realiza-se n vezes
```

- **Ignoram-se** operações dependentes do hardware e outros fatores
  - Instruções de leitura e escrita
  - Tempo de acesso à memória
  - Chamadas de subprogramas
- **Contabilizam-se**
  - Comparações;
  - Atribuições;
  - Trocas;
  - Deslocamentos;
  - Operações aritméticas.

A cada uma destas instruções atribui-se uma unidade de tempo e soma-se o número de instruções.

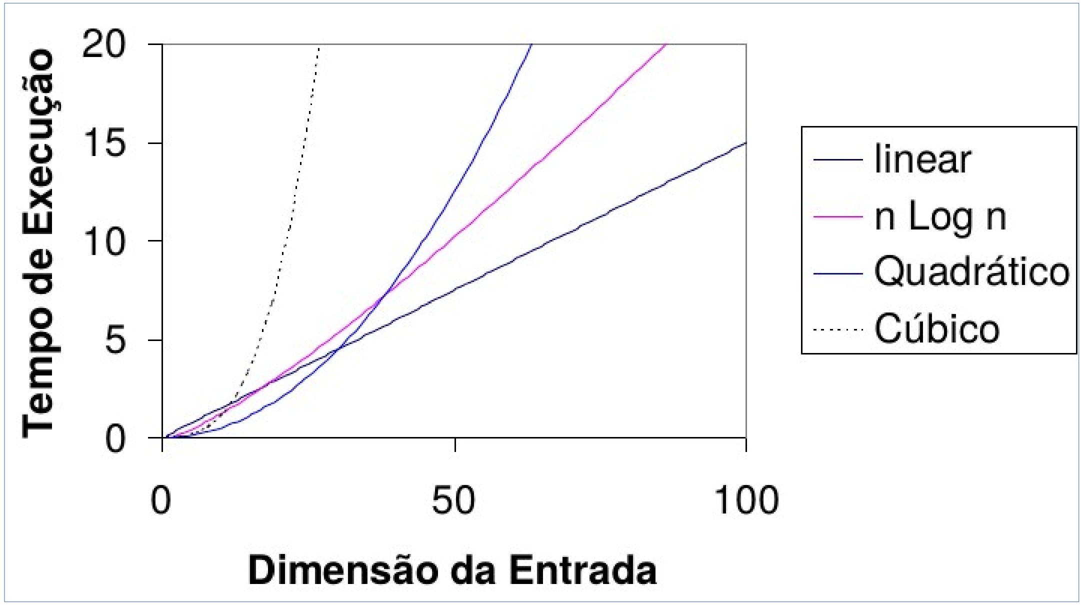
```
soma = 0; // Esta inicialização conta-se, mas realiza-se uma vez
for ( j=0; j<n; j++)
    soma += tab[j]; // Esta soma conta-se e realiza-se n vezes
```

Análise de Complexidade de Algoritmos

- O **mais importante** num algoritmo é saber como se comporta quando a **dimensão dos dados aumenta muito**
  - Complexidade **espacial**: quanta memória vai precisar?
  - Compexidade **temporal**: quanto tempo vai demorar?

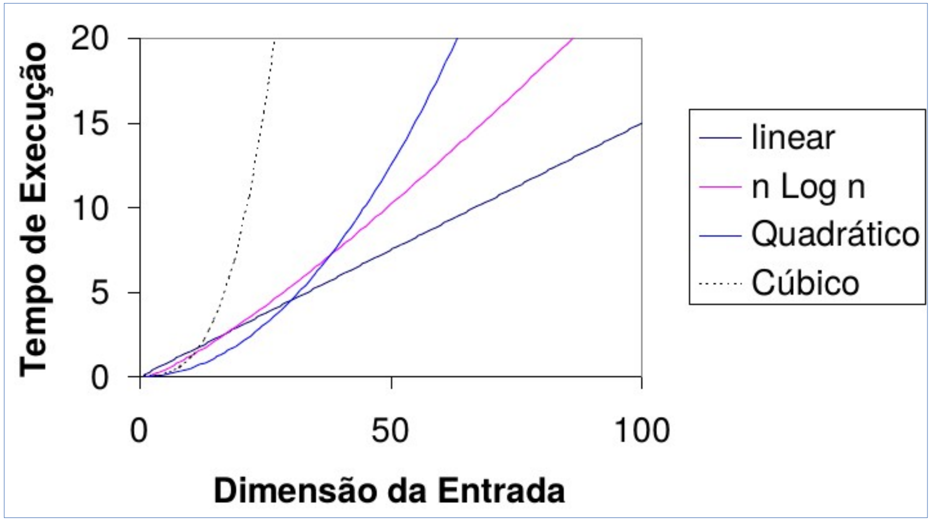
```
soma = 0; // Esta inicialização conta-se, mas realiza-se uma vez

for ( j=0; j<n; j++)
    soma += tab[j]; // Esta soma conta-se e realiza-se n vezes
```

- Normalmente preocupamo-nos apenas com a **ordem de grandeza das operações dominantes**
  - Ex., se houver uma operação de complexidade quadrática, uma operação de complexidade linear pode ser ignorada. Para **n** suficientemente elevado a quadrática sobrepõe-se e a linear pode quase ser desprezada



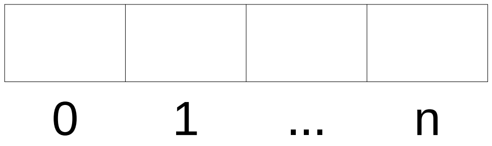

```
soma = 0;
x = 1;
for ( j=0; j<n; j++){
    soma += tab[j];    // Realizado n vezes
    for( i=0; i<n; i++)
        x += tab2[i][j]; // Realizado n * n vezes
    }
```

- Notação **O-grande (Big-O)**, Θ-grande, Ω grande
  - **O: ordem de grandeza melhor ou igual** (o O é como que o “limite superior”)
  - Θ: ordem de grandeza media
  - Ω: limite inferior



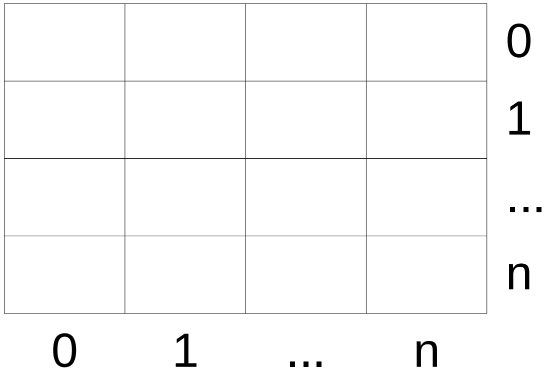

```
Θ(n) – complexidade exatamente linear
O(n) – complexidade linear ou melhor

Θ(n²) – complexidade exatamente quadrática
O(n²) – complexidade quadrática ou melhor
```

- **Ordens de complexidade típicas**
  - O(1) – complexidade constante, não aumenta com n
  - O(log(n)) – logarítmica
  - O(n) – linear
  - O(n.log(n)) – log linear
  - O(n^2), O(n^3), ... – quadrática, cúbica ...
  - O(e^n) – exponencial

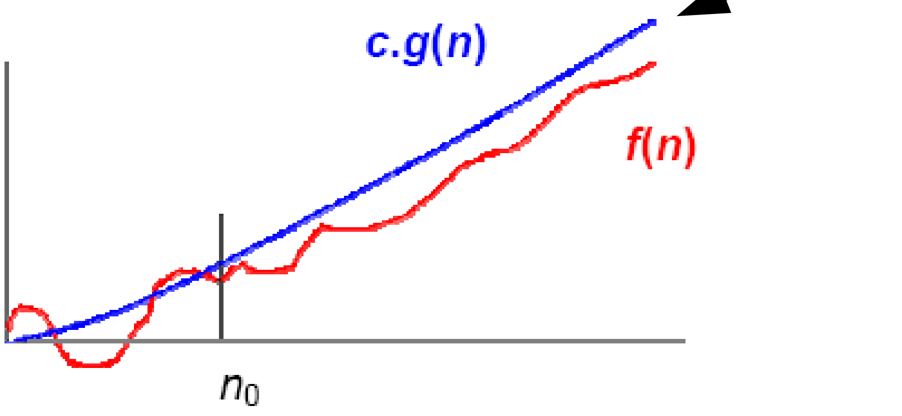


- Constante – O(1)
  - Verificar se o primeiro valor de um *array* é igual a uma chave
  - É só uma operação, independentemente do tamanho do *array*
- Linear -> O(n)
  - Procurar valor em *array* não ordenado de dimensão **n** (um ciclo)
  - É necessário verificar todos os números do *array*
- Quadrático O(n^2)
  - Procurar um valor numa matriz bidimensional não ordenada com lado n (um ciclo dentro de outro ciclo)
  - Por cada linha a mais tem

    de se percorrer todas as colunas, e vice-versa



- Um algoritmo de ordem inferior **não** é necessariamente “mais rápido”.
    - A ordem de complexidade indica o que acontece para valores “suficientemente elevados” de N (ver limites das funções, comportamento assintótico na matemática).
- Qualquer algoritmo de ordem superior irá tornar-se mais lento do que um algoritmo de ordem inferior “para N **suficientemente grande**”
- Para N pequeno o algoritmo mais eficiente pode ser diferente de N grande

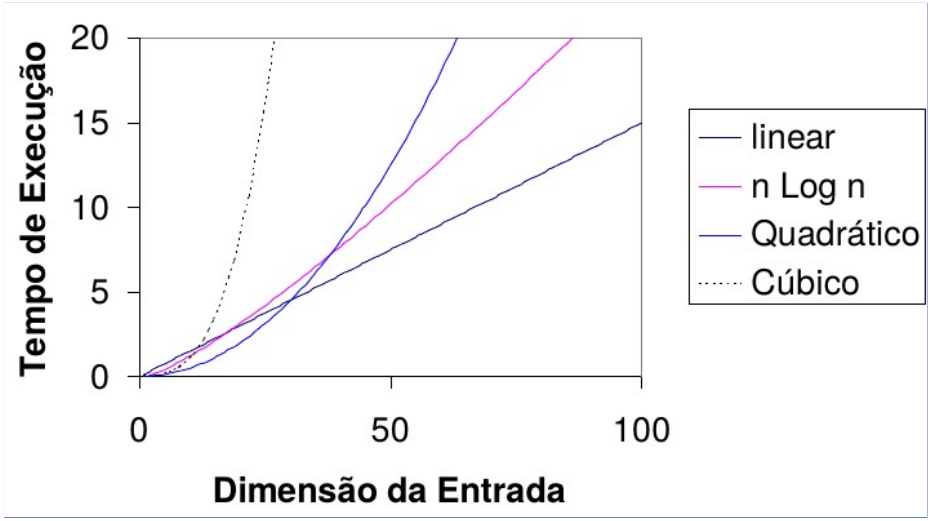



```
Exemplo de limite:
   f(n) tende para c.g(n), para
valores muito grandes de n.
   Para valores pequenos tem um
comportamento errático.
```

- Existem algoritmos muito eficientes que requerem muito pré-processamento ou setup inicial, sendo ineficientes no início
- O contrário também se verifica

Comportamento assintótico

- Num algoritmo de complexidade linear, duplicam os dados, o tempo de execução duplica
- Num algoritmo de complexidade quadrática, se os dados duplicam o tempo de execução aumenta o seu quadrado...



1 – Para cada um dos programas seguintes:

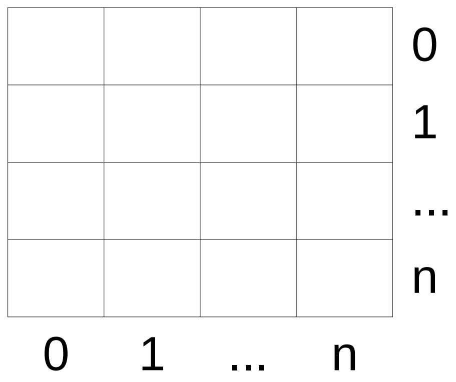
- Efetue a análise de complexidade.
- Calcule analiticamente quanto o tempo de execução deverá aumentar caso a dimensão de *N* seja aumentada *4x*.

a)

```
for(long i=0;i<n;i++)
   for(long j=0;j<n;j++)
      soma++;  // Quantas vezes é executada esta linha?
```

Ficha 1 - Exercícios

1 – Para cada um dos programas seguintes:



a)

```
for(long i=0; i<n; i++)
    for(long j=0; j<n; j++)
        soma++;  // Quantas vezes é executada esta linha?
```

*– A instrução **soma++** é executada n*n vezes (ciclo exterior * ciclo interior).*

– O programa é de complexidade O(n²).

*– Se o n for aumentado para 4n, o soma++ vai ser executado 4n*4n vezes. Portanto, o tempo de execução aumenta 16 vezes.*

Ficha 1 - Exercícios

```
b)for (long i=0;i<n; i++)
    soma++; // Quantas vezes é executada esta linha?
```

```
b)for (long i=0;i<n; i++)
      soma++; // Quantas vezes é executada esta linha?
```

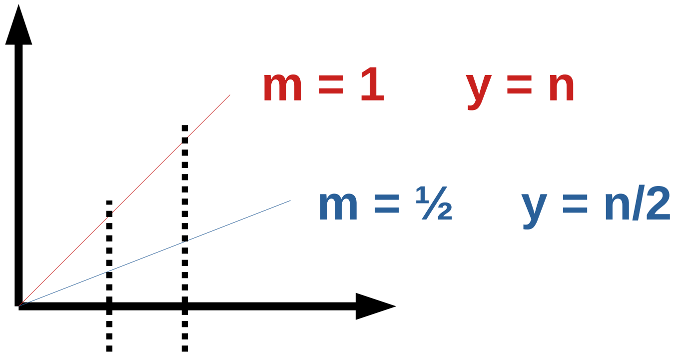
- Trata-se apenas de um ciclo, em que a instrução dominante executa n vezes. Portanto a complexidade é linear, O(n).

- Quanto n aumenta 4 vezes, o tempo de execução aumenta de n para 4n, portanto 4 vezes.

c)
```
for (long i=0;i<n;i+=2)
    soma++;  // Quantas vezes é executada esta linha?
```

```
c)   for (long i=0;i<n;i+=2)
        soma++;  // Quantas vezes é executada esta linha?
```

- O i é incrementado em passos de 2. Portanto o tempo de execução é n/2. Mas metade de algo linear é também linear. O algoritmo continua a ser de ordem **O(n)**.
- Aumentando o número de execuções de n para 4n, o tempo de execução aumenta 4 vezes. Passa a ser 4n/2.



d)
```
for(long i=0;i<1000;i++)
     for(long j=0;j<n;j++)
        soma++; // Quantas vezes é executada esta linha?
```

- Complexidade?
- Quanto aumenta o tempo com 4 vezes mais dados?

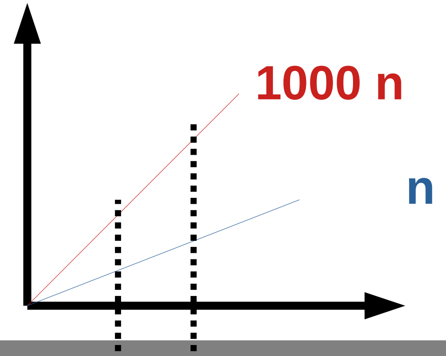
```
d) for(long i=0;i<1000;i++)
       for(long j=0;j<n;j++)
          soma++; // Quantas vezes é executada esta linha?
```

- Aqui, no pior dos casos, o ciclo externo executa 1000 vezes. Mil vezes algo linear é também linear.

- O facto de serem dois ciclos encadeados não implica automaticamente que seja um algoritmo quadrático.

*- A instrução dominante é executada, no pior dos casos, 1000 * n vezes, portanto o ciclo exterior atua como uma constante. O algoritmo continua a ser de ordem linear O(n).*

*- Quando n aumenta 4 vezes, o tempo passa a ser no pior dos casos 1000 * (4n) = 4000n, portanto aumenta de forma linear 4 vezes.*



e)
```
for(long i=0;i<n;i++)
    soma++;
for(long j=0;j<n;j++)
    soma++;
```

- Complexidade?
- Quanto aumenta o tempo com 4 vezes mais dados?

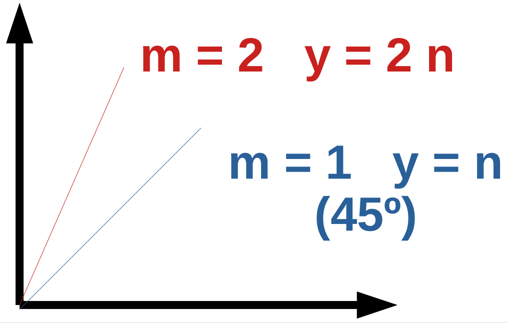
e)
```
for(long i=0;i<n;i++)
    soma++;
for(long j=0;j<n;j++)
    soma++;
```

- O tempo de execução é n + n = 2n.

- O dobro de algo linear é também linear.

- Vários "algoritmos" ou trechos de código linear consecutivos resultam em código linear.

- Quando n aumenta 4 vezes, o tempo passa a ser 2(4n), portanto aumenta linearmente 4 vezes.*



Ficha 1 - Exercícios

```
f) if(n>20000) n=20000;
     for(long i=0;i<n;i++)
       for(long j=0;j<n;j++)
          soma++;
```

- Complexidade?
- Quanto aumenta o tempo se os dados quadruplicarem?

```
f) if(n>20000) n=20000; // Esta condicao limita o valor de n
    for(long i=0;i<n;i++)
      for(long j=0;j<n;j++)
         soma++;
```

- Algoritmo de ordem constante, O(1).

- O aumento de n, no limite, não interfere com o tempo de execução, porque n é truncado em 20000.

*- Os ciclos encadeados têm um comportamento aparentemente quadrático. No entanto, o valor de n é limitado a 20000 pela primeira linha. Isso significa que o ciclo irá ter, sempre 20000*20000 iterações de "soma++", para valores suficientemente elevados de n (neste caso, valores de n>=20000).*

- Assim, o código tem o mesmo tempo de execução à medida que n aumenta para valores de n elevados, e dessa forma é de complexidade CONSTANTE.

- Respostas como "É O(N^2) para n<20000 e O(1) para n>=20000" não fazem muito sentido dado que, por definição, a notação de O descreve apenas o que acontece para valores de N ***suficientemente elevado****.*

Ficha 1 - Exercícios

```
g) for(long i=0;i<n;i++)
       for(long j=0;j<n*n;j++)
         soma++;  //Quantas vezes é executada esta linha?
```

- Complexidade?
- Quanto aumenta o tempo se os dados quadruplicarem?

Ficha 1 - Exercícios

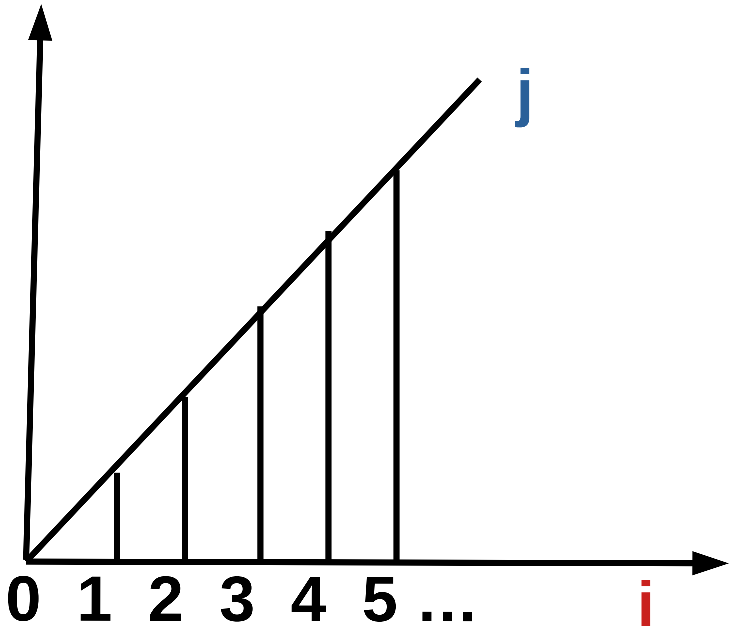
```
g) for(long i=0;i<n;i++)
       for(long j=0;j<n*n;j++)
           soma++;  //Quantas vezes é executada esta linha?
```

*- O primeiro ciclo executa n vezes. O segundo executa n*n. Portanto a instrução dominante soma++ executa n*(n*n) = n^3. O algoritmo é de ordem $O(n^3)$, complexidade cúbica.*

- Quando n aumenta 4 vezes, o tempo de execução passa a ser 4n(4n*4n) = 4^3n^3 = 64 n^3.*

- Dois ciclos encadeados não resultam necessariamente em complexidade quadrática – a complexidade pode ser menor ou maior.

```
h) for(long i=0;i<n;i++)
     for(long j=0;j<i;j++)
       soma++;
```



```
h) for(long i=0;i<n;i++)
     for(long j=0;j<i;j++)
       soma++;
```

- Quantas vezes é executado o ciclo interior por em cada iteração do ciclo exterior?

  Para i=0, o ciclo interior é executado 0 iterações

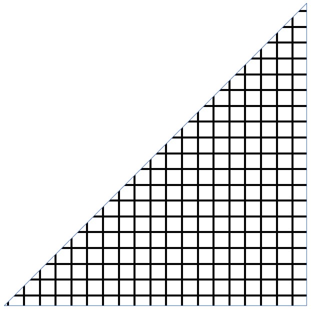
  Para i=1, o ciclo interior é executado 1 iterações

  Para i=2, o ciclo interior é executado 2 iterações

  Para i=3, o ciclo interior é executado 3 iterações

  …

  Para i=**M**, o ciclo interior é executado **M-1** iterações



- O número de iterações do ciclo interior é dado por 1+2+3+4+…+ (N-2)+(N-1)

  Esta soma pode ser calculada matematicamente pela expressão N * sum{i=0..N}= ½(N)(N+1) que é $O(N^2)$.

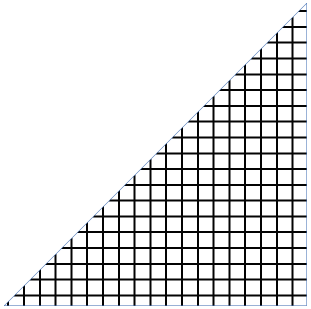
- Graficamente, a soma de todos os números entre 1 e N pode ser representada como sendo a àrea de um triângulo retângulo (da esquerda para a direita, uma coluna de altura 1, seguida de uma coluna de altura 2, etc). Dessa forma, torna-se claro que a soma (àrea) é igual a aproximadamente metade da àrea do quadrado (N^2). Em vez de uma matriz, processa-se "meia matriz".

- O termo linear que pode ser observado na fórmula torna-se irrelevante para valores elevados de N, quando comparado como o termo quadrático.

- Quando n aumenta 4 vezes, o tempo de execução aumenta na mesma 16 vezes.

```
i)for(long i=0;i<n*n;i++)
    for(long j=0;j<i;j++)
      soma ++;
```

```
i)for(long i=0;i<n*n;i++)
    for(long j=0;j<i;j++)
      soma ++;
```



- O ciclo interno vai iterar $1+2^2+3^2+4^2+\ldots(n-1)^2$ vezes

- A expressão matemática para este somatório é (N * (N + 1) * (2N + 1)) / 6, pelo que o resultado é de ordem cúbica.

- Quando n aumenta 4 vezes, o tempo de execução aumenta $4^3 = 64$ vezes.

```
j) for(long i=1;i<n;i*=2)
        soma++;
```

A cada passo, o i duplica.

- Qual a ordem do algoritmo?
- Quanto aumenta o tempo com o quádruplo dos dados?

```
j) for(long i=1;i<n;i*=2)
     soma++;
```

- O i **duplica** a cada iteração. Exemplo de ordem **logaritmica**, O(log (N))
- Um aumento de **4x** de N obriga a mais **duas** iterações.
- Na matemática, um logaritmo é o "inverso" do expoente : $\log_2(n) = x \Leftrightarrow 2^x=n$
- *É o número de bits necessário para representar o inteiro N.*

  $\log_2(N)$ é o número de vezes que é necessário dividir N por 2 para chegar a 1. (esta interpretação será importante quando for abordada a pesquisa binária)

  $\log_2(N)$ é o número de vezes que é preciso multiplicar por 2 (começando em 1) até chegar a N. Como no exemplo de código desta alínea.

- Quantas iterações adicionais são necessáras se o valor de N aumentar para o **dobro**? **1 iteração**. Quando N quadruplica, são realizadas mais duas iterações.
  - - A base do logaritmo não importa para a complexidade porque o logaritmo de N numa determinada base é proporcional ao logaritmo do mesmo valor noutra base, $\log_B(N)=\log_X(N)/\log_X(B)$. Ex $\ln(N) = \log_2(N) / \log_2(e)$. O valor $1/\log_2(e)$ é constante e independente em relação a N. Graficamente a curva mantém a "mesma forma" pelo que a caraterística de complexidade é a mesma.
- Compare a complexidade logaritmica com a linear e a exponencial. Qual cresce mais depressa? E mais devagar?

2 – Relativamente a cada uma das alíneas da pergunta anterior, implemente o código e verifique o tempo de execução (use *System.nanoTime()* para obter um valor *long* com o tempo actual em *ns* (1 *ns* $=10^{-9}$ segundos)). Compare as suas previsões com os resultados obtidos. Deve procurar encontrar valores de *n* que resultem em tempos de execução significativos (alguns segundos). Pode definir uma constante long acrescentando ao valor constante um L; por exemplo `long n=127343734L`.

Dependendo do computador, poderá ser complicado cumprir isso para algumas alíneas.

- Ver ficheiro java no moodle para exemplo do que é pretendido.

- O tempo de execução pode variar por causa de muitos fatores, pelo que é expectável que os tempos de execução variem ligeiramente e acompanhem as previsões apenas de forma aproximada. Alguns exemplos (lineares, constantes, logarítmicos) poderão ser muito rápidos e difíceis de observar.

2 – Relativamente a cada uma das alíneas da pergunta anterior, implemente o código e verifique o tempo de execução (use *System.nanoTime()* para obter um valor *long* com o tempo actual em *ns* (1 *ns* =10^{-9} segundos)). Compare as suas previsões com os resultados obtidos. Deve procurar encontrar valores de *n* que resultem em tempos de execução significativos (alguns segundos). Pode definir uma constante long acrescentando ao valor constante um L; por exemplo `long n=127343734L`.

Preencha uma tabela semelhante à seguinte. Faça gráficos com os tempos para melhor visualizar o resultado.

Alínea	Ordem do algoritmo	Tempo para N	Tempo para 4N
a (1º teste)	quadrático		
a (2º teste)	quadrático		
b (1º teste)	linear		
...	...		

Ficha 1 - Exercícios

```java
public class Main {
  private static long stopTime;
  private static long startTime;
  static void ex1b(long n){
    long soma=0;
    startTimer();
    for(long i=0;i<n;i++)
      soma++;
    stopTimer();
    System.out.println("Soma="+soma);
    showTime();
  }
  static void ex1a(long n){
    long soma=0;
    startTimer();
    for(long i=0;i<n;i++)
      for(long j=0;j<n;j++)
        soma++;
    stopTimer();
    System.out.println("Soma="+soma);
    showTime();
  }
```

```java
private static void showTime() {
  long interval=stopTime-startTime;
  long secs=interval/1000000000L;
  long decs=interval-secs*1000000000L;
  decs/=100000000L;
  System.out.println("secs="+secs+"."+decs);
  }
private static void startTimer() {
  startTime=System.nanoTime();
  }
private static void stopTimer() {
  stopTime=System.nanoTime();
  }
public static void main(String[] args) {
  long n = 40000;
  ex1a( n );   // Executar código
  ex1a( 4 * n); // Executar com 4x mais dados
  }
}
```