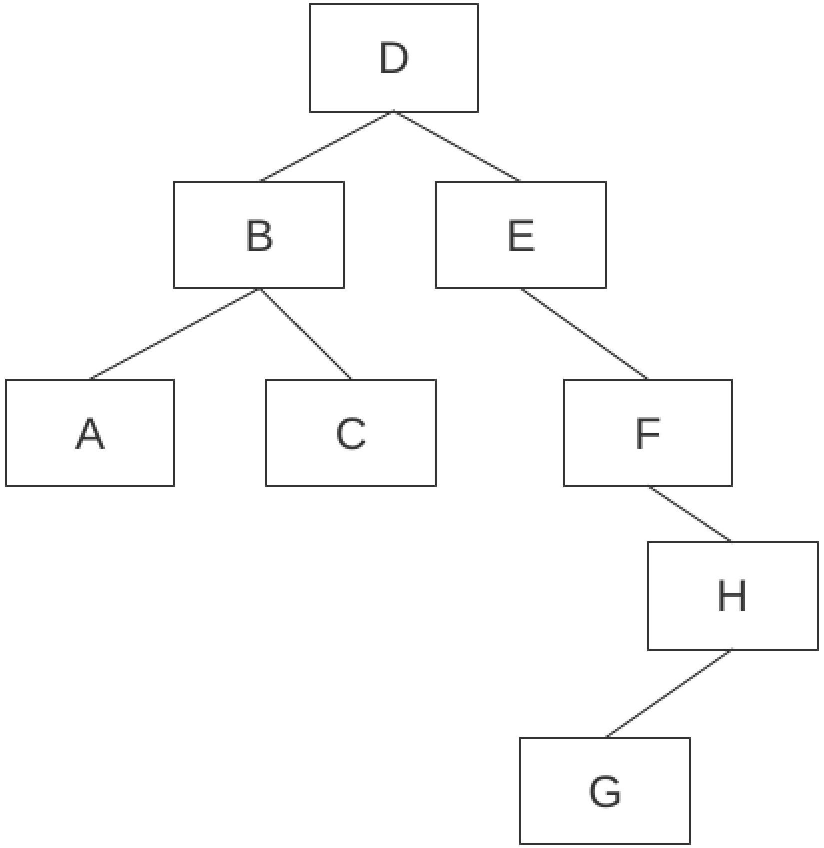
What we need is not the will to believe but the will to find out.
Bertrand Russell

Engenharia Informática

3º ano, 1º semestre



Docentes:
Jorge Barreiros
Mateus Mendes

Como vai funcionar a Unidade Curricular?

- **Aulas Teóricas:**
  - Prof. Jorge Barreiros
- **Aulas práticas:**
  - Prof. Jorge Barreiros (jmsousa@isec.pt
  - Prof. Mateus Mendes (mmendes@isec.pt)
- **Horário de atendimento**
  - Quarta 9:00 – 11:00
  - Quinta 9:00 – 13:00

Calendário previsto

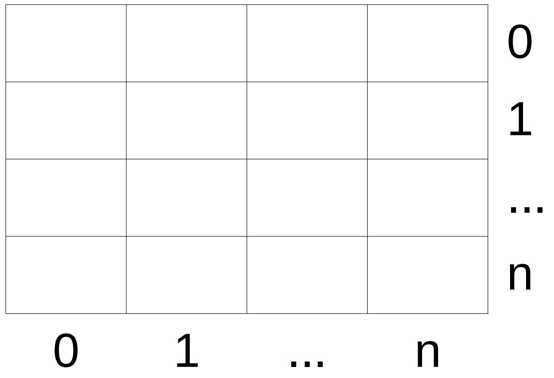
- **Sujeito a alterações...**

Semana	P3 – segunda L1.2
2023-09-11	
2023-09-18	F1 – Complexidade
2023-09-25	F2 – Pesquisa binária e variações
2023-10-02	F2,. 10
2023-10-09	F3 – Genéricos
2023-10-16	F4 – Iteradores
2023-10-23	F5 – Listas
2023-10-30	Listas, mapas
2023-11-06	Teste 1
2023-11-13	F6 – Priority Queue
2023-11-20	F7 – Árvores
2023-11-27	Árvores
2023-12-04	Árvores
2023-12-11	Teste 2
2023-12-18	seminário
2023-12-25	Natal

Semana	P1 – quinta L1.7
2022-09-14	
2022-09-21	F1 – Complexidade
2022-09-28	F2 – Pesquisa binária e variaçõe
2022-10-05	feriado
2022-10-12	F2,. 10
2022-10-19	F3 – Genéricos
2022-10-26	F4 – Iteradores
2022-11-02	F5 – Listas
2022-11-09	Teste 1
2022-11-16	Listas, mapas
2022-11-23	F6 – Priority Queue
2022-11-30	F7 – Árvores
2022-12-07	Árvores
2022-12-14	Teste 2
2022-12-21	seminário
2022-12-28	Natal

O que vai ser estudado em Estruturas de Dados?

- **Diferentes formas de representar dados**
  - Pilhas
  - Filas
  - Listas
  - Árvores
  - Outras estruturas
- **Comportamento de diferentes algoritmos, ao realizar operações sobre grandes quantidades de dados**



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
  - IDE à escolha
- **Avaliação (a confirmar na FUC):**
  - Exame: 10 Valores
  - Seminário: **1.5 Valores**
  - Dois testes Laboratoriais: **8.5 Valores** – nota que pode ser substituída no exame de recurso se fizerem a parte prática

- **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, não necessariamente por ser mais exigente, mas por ser realizada mais vezes

```
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, como
  - 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.

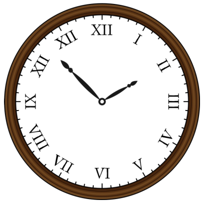
```
soma = 0; // Esta inicialização conta-se, mas realiza-se uma vez
for ( j=0; j<n; j++)
    soma += tab[j]; // Esta soma conta-se e realiza-se n vezes
```

- O **mais importante** num algoritmo é saber como se comporta quando a **dimensão dos dados aumenta muito**

- Complexidade **espacial**: quanta memória vai precisar?

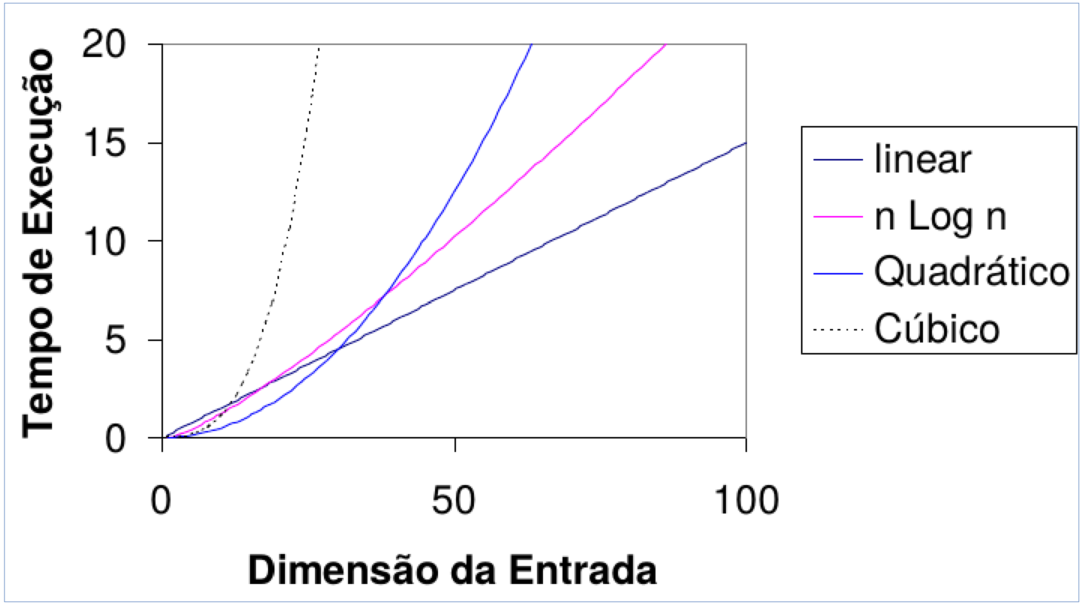
Sao inversamente proporcionais, se podemos usar mais espaço em memoria, demora menos tempo.

- 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
```

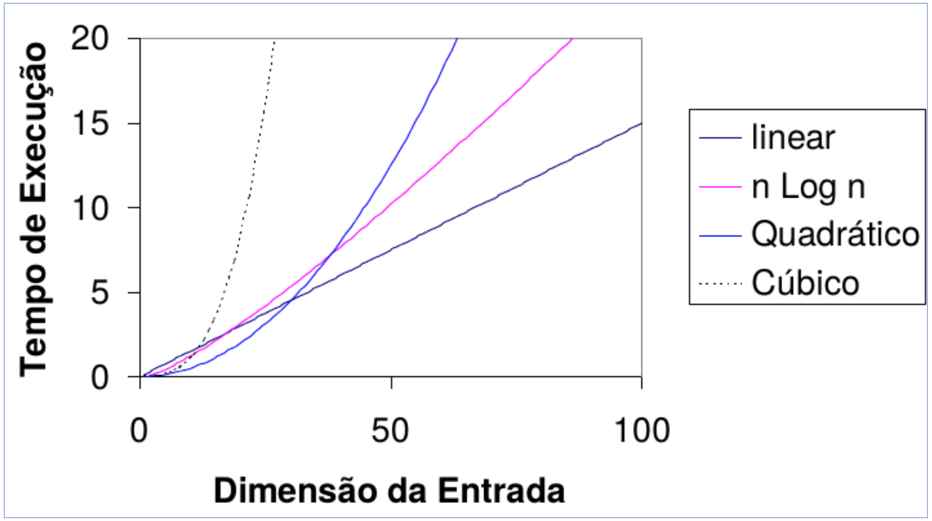
- Normalmente preocupamo-nos apenas com a **ordem de grandeza das operações dominantes**
  - Ex., se houver uma operação de complexidade quadrática, uma operação de complexidade linear pode ser ignorada. Para **n** suficientemente elevado a quadrática sobrepõe-se e a linear pode quase ser desprezada




```
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
    }
```

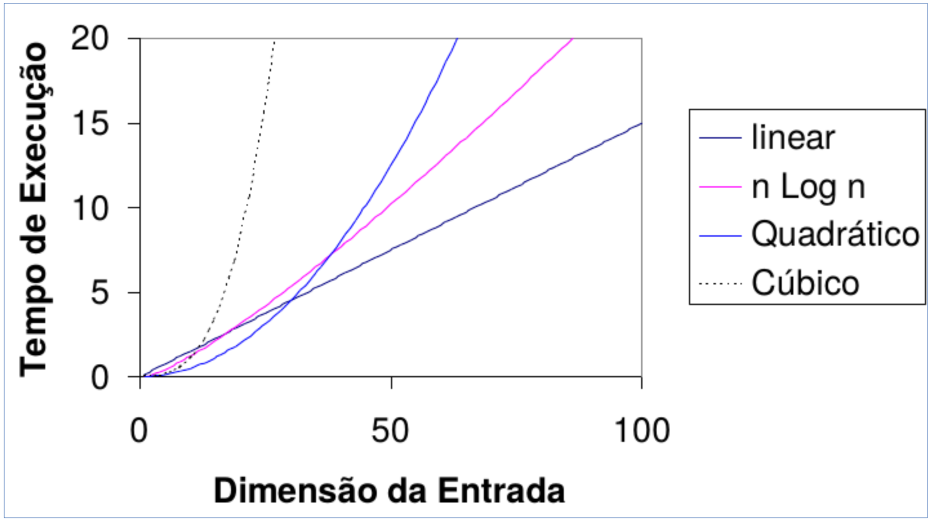
Aqui so nos preocupamos com n*n pq é muito maior que n vezes e muito mais que 1 vez (nas atribuiçoes)

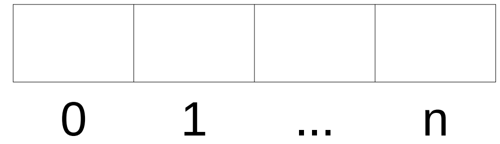
- Notação **O-grande (Big-O)**, Θ-grande, Ω grande
  - **O: ordem de grandeza melhor ou igual** (o O é como que o "limite superior")
  - Θ: ordem de grandeza media
  - Ω: limite inferior



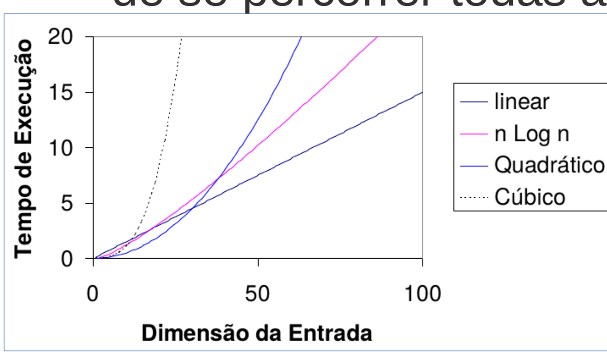

```
Θ(n) – complexidade exatamente linear
O(n) – complexidade linear ou melhor

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

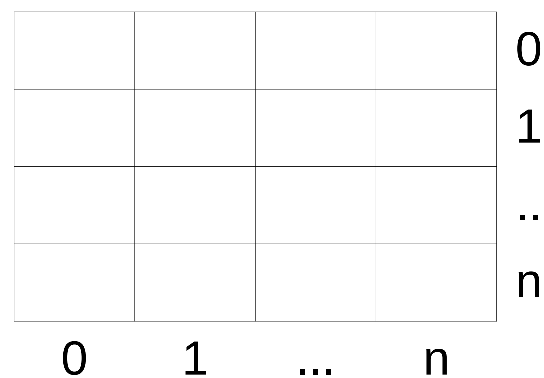
- **Ordens de complexidade típicas**
  - O(1) – complexidade constante, não aumenta com n
  - O(log(n)) – logarítmica
  - O(n) – linear
  - O(n.log(n)) – log linear
  - O(n^2), O(n^3), ... – quadrática, cúbica ...
  - O(e^n) – exponencial





- 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(n²)
  - 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

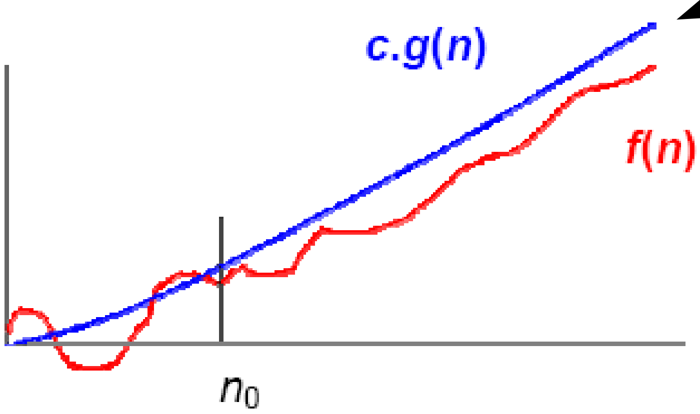




Comportamento assintótico

O comportamento inicial nao interessa. Se analisarmos ao inicio e for cúbico e depois for linear, então é complexidade linear, pois so interessa quando o n é infinito, ou seja quando o numero de dados é muito grande, ou o número de operaçoes e muito grande

- 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

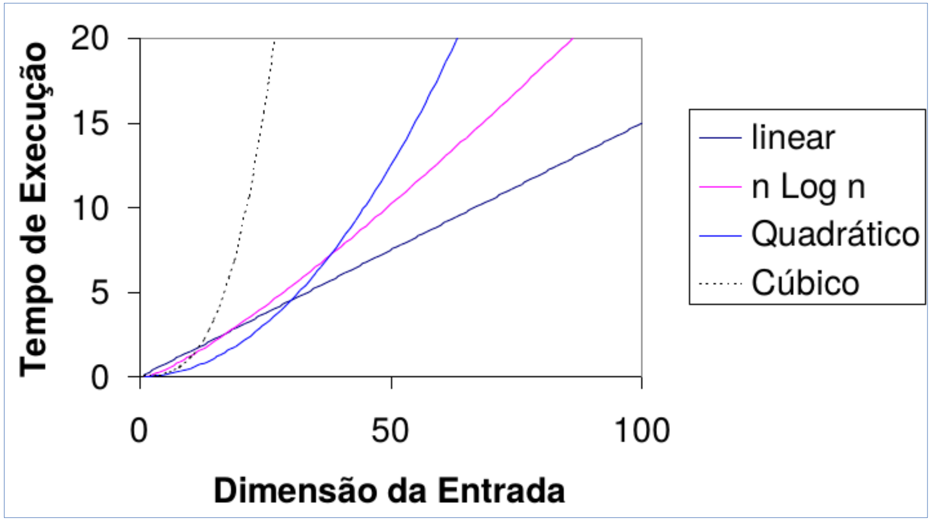



```
Exemplo de limite:
   f(n) tende para c.g(n), para
valores muito grandes de n.
   Para valores pequenos tem um
comportamento errático.
```

- 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 quadruplica...



Ficha 1 - Exercícios

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?
```

$4n$

$4n$

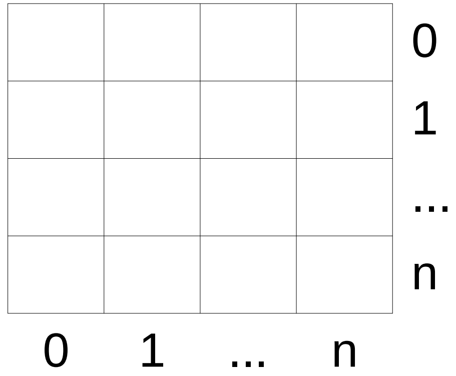
$(4n \times 4n) = 16n^2$

Complexidade: O(n^2). -> para n dados

Para dimensao de N ser 4x:

o primeiro for fica até 4n e o segundo for tambem 4n entao fica 4n * 4n que dá 16n^2 entao —> O(16N^2)

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 $O(n^2)$.

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

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

executado ————— n vezes

tempo —————— O(n)

4n

Ordem - O(4n)

Na linear quadriplico os dados o tempo tambem quadruplica

```
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).

- Quando 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?
```

Como da saltos de 2 demora metade do tempo para fazer tudo entao:

tempo -------------- n/2

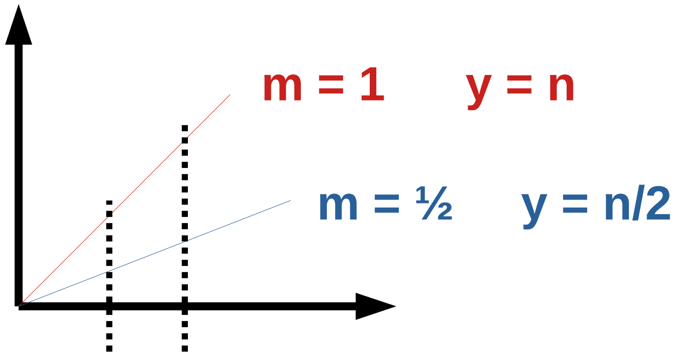
Ordem O(n/2) —> O(n)

A constante ali nao muda nada portanto é linear —> em termos de grafico O(n) tem declive 1 e O(n/2) tem declive 1/2 portanto ambas tendem para +infinito na mesma ordem de grandeza

Para 4n execuçoes o tempo de execuçao aumenta 4 vezes

```
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?
```

— 1000

— n

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

tempo ———— 1000 * n

O(1000 * n) —> O(n)

4x mais dados —> 4 vezes mais tempo para executar

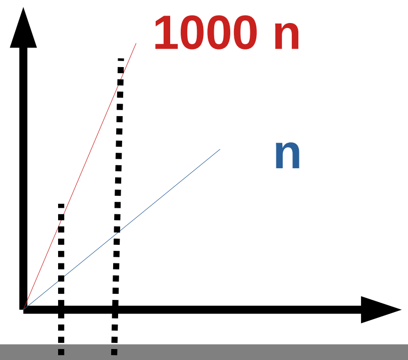
```
d) for(long i=0;i<1000;i++)
       for(long j=0;j<n;j++)
         soma++; // Quantas vezes é executada esta linha?
```

- 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.*



Ficha 1 - Exercícios

e)

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

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

tempo: n (primeiro ciclo) + n (segundo ciclo) = 2n

O(2n) = O(n)

Vários “algoritmos” ou trechos de codigos lineares consecutivos resultam em codigo linear.

Para 4n execuçoes fica 4x mais tempo

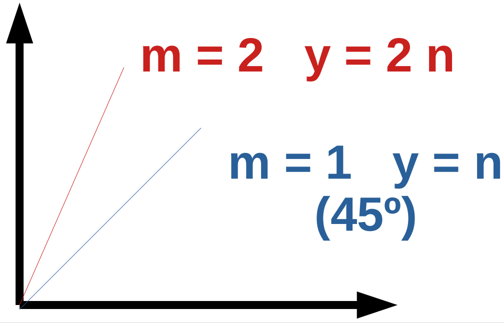
e)
```
for(long i=0;i<n;i++)
    soma++;
for(long j=0;j<n;j++)
    soma++;
```

- 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.*



```
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?

Até ao n < 20000 tem um comportamente quadratico nos for, mas quando on tende para mais infinito fica linear porque o n vai ser sempre 20000

ou seja,

inicialmente é quadratico mas passa a ser linear

```
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****.*

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

n

$n \times n = n^2$

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

tempo —> n (n * n) = n^3

O(n^3)

Para 4x mais dados:

tempo —> 4n (4n * 4n) = 4^3 * n^3 = 64 n^3

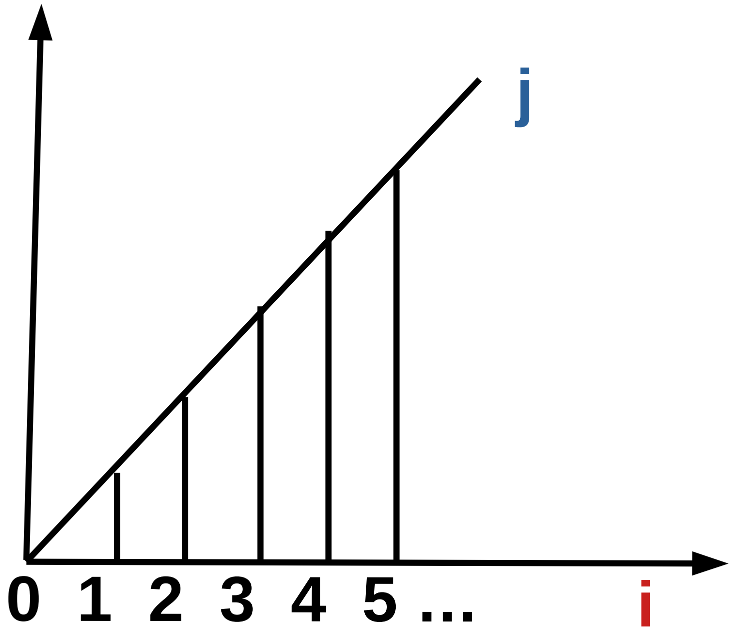
g)
```
for(long i=0;i<n;i++)
    for(long j=0;j<n*n;j++)
      soma++;  //Quantas vezes é executada esta linha?
```

*- 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

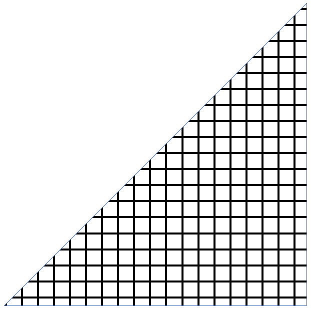
Para i=1, o ciclo interior é executado 1 iterações

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.

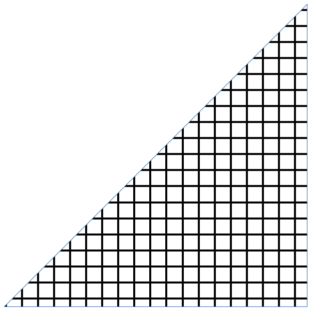
\- Quando n aumenta 4 vezes, o tempo de execução aumenta na mesma 16 vezes.

Ficha 1 - Exercícios

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

Ficha 1 - Exercícios

```
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.

Ficha 1 - Exercícios

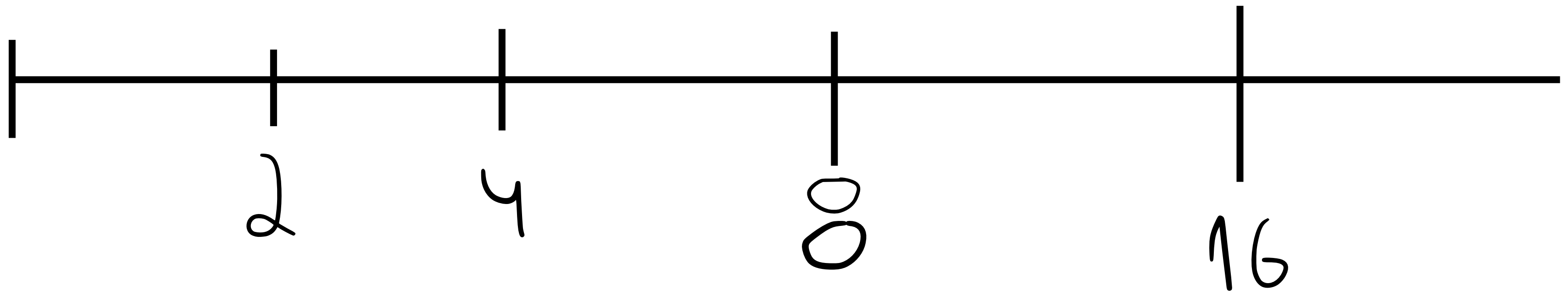
```
j) for(long i=1;i<n;i*=2)
       soma++;
```

Se eu duplicar o numero de dados so tenho que fazer mais uma operaçao porque ela é a duplicar, se 4x numero de dados so tenho que fazer mais duas operaçoes

Se for de logaritmico de base dez quer dizer que quando multiplico por 10 so faço mais uma iteraçao

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

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

```
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
  }
}
```

3 – Considere uma matriz de NxN, na qual os números estão colocados por ordem ascendente, tanto nas linhas como nas colunas. Pretende-se criar e implementar um algoritmo que verifica se um determinado numero está ou não na matriz. Sendo assim,

a) Crie, implemente e teste um algoritmo de complexidade $O(N^2)$

Solução trivial, pesquisar todas as células

b) Crie, implemente e teste um algoritmo de complexidade O(N)

ver descrição em https://www.geeksforgeeks.org/search-in-row-wise-and-column-wise-sorted-matrix/

Começa-se num dos cantos inf.esq ou sup.dir e em cada iteração elimina-se a linha ou coluna correspondente

Exemplo: encontrar o 35

1	10	20	40
2	18	30	80
3	35	70	90
4	70	85	100

Começar num destes dois cantos
Escolhe-se o canto superior direito

1	10	20	40
2	18	30	80
3	35	70	90
4	70	85	100

A sequência de números de linha e coluna que fazem esquina este ponto é crescente:

1, 10, 20, 40, 80, 90, 100

Portanto, ao encontrar o 40 pode ser eliminada a coluna que o contém: o 35 só pode estar para a esquerda
Se fosse encontrado o 34, por exemplo, poderia ser eliminada a linha

1	10	20	40
2	18	30	80
3	35	70	90
4	70	85	100

Eliminada a coluna, repete-se o processo e encontra-se o 20 no canto superior direito. Como 20<35 pode-se eliminar a primeira linha.

1	10	20	40
2	18	30	80
3	35	70	90
4	70	85	100

Eliminadas a primeira coluna e primeira linha, repete-se o processo e encontra-se o 30 no canto superior direito.
Como 30 < 35, elimina-se a segunda linha.

Repete-se o processo e encontra-se o 70 no canto superior direito da tabela restante. Elimina-se a coluna que tem o 70.

Ao repetir o processo, encontra-se o 35 no canto superior direito.

1	10	20	40
2	18	30	80
3	35	70	90
4	70	85	100

1	10	20	40
2	18	30	80
3	35	70	90
4	70	85	100

Complexidade:
- Em cada iteração é descartada uma linha ou uma coluna.
- Para uma matriz NxN, o algoritmo no pior dos casos termina em 2N iterações.
- Portanto, o algoritmo é **O(N)**

- Podem existir algoritmos mais eficientes (logarítmicos), mas de mais difícil implementação e compreensão.