INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Guarulhos

Estruturas de Dados 2

02 – Complexidade e análise de algoritmos

Antonio Angelo de Souza Tartaglia
angelot@ifsp.edu.br

Análise de Algoritmos

- Na ciência da computação, é a área de pesquisa cujo foco são os algoritmos;
- Busca responder a seguinte questão:

  Podemos fazer um algoritmo mais eficiente?

- Para responder a esta pergunta precisamos ser capazes de comparar os algoritmos, é aí que entre a parte de Análise de Algoritmos.

Análise de Algoritmos

- Podemos resolver um problema de várias maneiras diferentes, isto é, podemos utilizar algoritmos diferentes para resolver um mesmo problema.

- A questão é: Algoritmos diferentes, mas capazes de resolver o mesmo problema, **não necessariamente** o fazem com a mesma eficiência;

- Essas diferenças de eficiência podem ser:
  - **Irrelevantes** – para um pequeno número de elementos processados;
  - **Crescer proporcionalmente** – com o número de elementos processados.

É necessário compará-los para definir...

Ex.: com 10 elementos:
Se for O(n) -> 10 passos de execução
Se for O(n^2) -> 100 passos de execução

Análise de Algoritmos

- Para comparar a eficiência dos algoritmos foi criada uma medida chamada de "**Complexidade Computacional**";

- Basicamente, ela indica o custo ao se aplicar um algoritmo, sendo:

CUSTO = MEMÓRIA + TEMPO

- Onde:
  - MEMÓRIA – quanto de espaço o algoritmo vai consumir;
  - TEMPO – a duração de sua execução.

Análise de Algoritmos

- A complexidade de um algoritmo reflete o esforço computacional requerido para executá-lo. As principais medidas de esforço são tempo e espaço, relacionadas à velocidade de processamento e quantidade de memória utilizada, respectivamente.

- O esforço computacional não pode ser descrito simplesmente por um número, porque a quantidade de trabalho requerido em geral depende do volume de entrada.

- O desempenho de um algoritmo sobre o volume de entrada reflete o esforço, ou seja, quantidade de trabalho necessária para executar o algoritmo com esse tamanho de entrada.

Análise de Algoritmos

- A complexidade, então, depende do tamanho da entrada, e os principais critérios são o **pior caso** e **caso médio**. Assim, complexidade de um algoritmo reflete o esforço para executá-lo sobre um conjunto de entradas (de um determinado tamanho).

- A complexidade **pessimista**, que é o **pior caso,** dá o valor máximo que ela pode atingir.

- A complexidade média, que é a que normalmente trabalhamos, dá o valor esperado: a média dos esforços levando em conta a probabilidade de ocorrência de cada entrada.

- A complexidade pode ser determinada com base em **operações fundamentais** e no **tamanho da entrada**. Ambos devem ser apropriados ao algoritmo ou problema.

Análise de Algoritmos

- Para determinar se um algoritmo é o mais eficiente, podemos então utilizar duas abordagens:
  - **ANÁLISE EMPÍRICA** – comparação entre os programas
  - **ANÁLISE MATEMÁTICA** – estudo das propriedades do algoritmo

A análise experimental ou empírica, costuma depender de detalhes de implementação, variando de máquina a máquina

Análise matemática fornece a complexidade intrínseca do algoritmo

Análise de Algoritmos

- Análise Empírica:
  - Avalia o custo (ou complexidade) de um algoritmo a partir da avaliação da execução do mesmo quando implementado, ou seja, um algoritmo é analisado pela execução de seu programa correspondente;

A comparação é feita com a implementação deste e com a do outro algoritmo

- Possui uma série de vantagens. Através dela podemos:
  - Avaliar o desempenho em uma determinada configuração de computador/linguagem;
  - Considerar custos não aparentes, como por exemplo “custos de alocação de memória”;
  - Comparar computadores;
  - Comparar linguagens.

Análise de Algoritmos

- Análise Empírica – Dificuldades:
  - Há a necessidade de implementação do algoritmo, e isso depende da habilidade do programador;

Se o programador for experiente, ele pode tirar vantagens da linguagem...

  - O resultado pode ser **mascarado** pelo hardware (o computador utilizado) ou software (eventos ocorridos no momento da avaliação);
  - E, qual a natureza dos dados:
    - Dados reais;
    - Aleatórios – avaliam o desempenho médio;
    - Perversos – avaliam o comportamento no pior caso.

Análise de Algoritmos

- Análise Matemática:
  - Permite um estudo formal de um algoritmo ao nível da **ideia** por trás do algoritmo;
  - Faz uso de um computador idealizado e **simplificações** que buscam considerar somente os custos dominantes do algoritmo;

Como o algoritmo se comporta realizando determinada tarefa...

  - A medição do tempo gasto, é realizada de maneira independente do **hardware** ou da **linguagem** usada na sua implementação.

Analisa-se apenas a ideia do algoritmo, pequenos detalhes da implementação são desconsiderados.

Análise de Algoritmos

- Vantagens:
  - Detalhes de baixo nível, como a linguagem de programação utilizada, o hardware no qual o algoritmo é executado, ou o conjunto de instruções da CPU, são ignorados;
  - Permite entender como um algoritmo se comporta à medida que o conjunto de dados de entrada cresce. Assim, podemos expressar a relação entre o conjunto de dados de entrada e a quantidade de tempo necessária para processar esses dados, gerando sua saída.

Análise de Algoritmos

- Análise Matemática – Contando instruções de um algoritmo

  - Como exemplo utilizaremos um algoritmo que procura o maior valor em um vetor “V” contendo “n” elementos, e o armazena na variável “M”:

```c
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/       M = V[i];
/*5*/   }
/*6*/}
```

Análise de Algoritmos

- Análise Matemática – Contando instruções de um algoritmo
  - Iniciaremos contando quantas “**instruções simples**” o algoritmo executa;
  - Uma “**Instrução simples**”, é uma instrução que pode ser executada diretamente pela CPU ou algo muito próximo disso.
- Tipos de instruções que serão consideradas como simples:
  - Atribuição de um valor a uma variável;
  - Acesso ao valor de um determinado elemento do Vetor;
  - Comparação de dois valores;
  - Incremento de um valor;
  - Operações aritméticas básicas, tais como adição e multiplicação.
- Assumiremos que:
  - Todas estas instruções possuem o mesmo custo;
  - Os **comandos de seleção** possuem custo zero.

Por exemplo um comando `if` tem custo zero, o que é contado como “custo” é apenas a comparação que é feita dentro do `if`.

Análise de Algoritmos

- Contando instruções de um algoritmo
  - No exemplo abaixo, o custo da “linha 1” é de 1 instrução;
  - Na “ linha 1”, o valor da primeira posição do Vetor é copiado para a variável “M”:

    Acessar o valor “V[0]” e atribuí-lo a “M”.

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/       M = V[i];
/*5*/   }
/*6*/}
```

Atribuição: 1 instrução

Análise de Algoritmos

- Contando instruções de um algoritmo
- O custo da inicialização do laço `for` (linha 2), é de 2 instruções:
  - O comando `for` precisa ser inicializado: 1 instrução (`i = 0`)
  - Mesmo que o vetor tenha tamanho zero, ao menos uma comparação será executada (`i < n`), o que custa mais 1 instrução.

Atribuição: 1 instrução

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/       M = V[i];
/*5*/   }
/*6*/}
```

Atribuição e comparação para inicializar o `for`: 2 instruções. Mesmo que **n** seja igual à zero, e o vetor não possua elementos, ao menos uma comparação será executada.

Análise de Algoritmos

- Contando instruções de um algoritmo
- O custo para executar o comando de laço `for` (linha 2), é de: 2n instruções;
- Ao final de cada iteração do laço `for`, precisamos executar uma instrução de:
    - Incremento (`i++`);
    - A comparação para verificar se vamos continuar a executar o laço `for` (`i < n`).
- O laço for será executado "n" vezes. Assim, essas 2 instruções também serão executadas "n" vezes: 2n instruções.

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/    if(V[i] >= M){
/*4*/       M = V[i];
/*5*/    }
/*6*/}
```

Atribuição: 1 instrução

Atribuição e comparação para inicializar o `for`: 2 instruções

Comparação e incremento: mais 2 instruções que serão executadas "**n**" vezes.

Análise de Algoritmos

- Custo Dominante ou pior caso do algoritmo
  - Ignorando os comandos contidos no corpo do laço `for`, teremos que o algoritmo precisa executar $\mathbf{3} + \mathbf{2}\boldsymbol{n}$ instruções:
    - 3 instruções antes de iniciar o laço `for`
    - 2 instruções ao final de cada laço `for`, o qual é executado "`n`" vezes.
  - Assim, considerando que tivéssemos apenas um **laço vazio**, podemos definir uma função matemática que representa o custo do algoritmo em relação ao tamanho do vetor de entrada:

$$\boldsymbol{f}(\boldsymbol{n}) = \mathbf{2}\boldsymbol{n} + \mathbf{3}$$

Análise de Algoritmos

- Contando as instruções restantes do “for”:
  - O comando `if`: 1 instrução - acesso ao valor do vetor e a sua comparação (`V[i] >= M`);
  - Dentro do comando `if`: 1 instrução - acessa o valor do vetor e o atribui a outra variável (`M = V[i]`). Porém, sua execução depende do resultado da comparação feita anteriormente pelo comando `if`.

- Problema
  - As instruções vistas anteriormente eram sempre executadas;
  - As instruções dentro do “for” podem ou não ser executadas

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/       M = V[i];
/*5*/   }
/*6*/}
```

Essa instrução nem sempre será executada...

Análise de Algoritmos

- Antes, bastava saber o tamanho do vetor “n”, para definir a função de custo $f(n)$. Agora temos que considerar também o conteúdo do vetor.
  - Exemplo - dois vetores de mesmo tamanho:

```
V1 = {1, 2, 3, 4};
V2 = {4, 3, 2, 1};
```

  - Vetor V1: mais instruções são executadas, o comando `if` sempre será executado, porque a comparação sempre será verdadeira;
  - Vetor V2: a atribuição dentro do bloco do `if` nunca será executada, pois a avaliação do comando `if`, sempre resultará em `false`. Como resolver?

Análise de Algoritmos

- Custo Dominante ou pior caso do algoritmo

  - Ao analisarmos um algoritmo, é muito comum considerarmos o **pior caso** possível;
  - **Pior** caso é igual a: maior número de instruções executadas.

  - No nosso algoritmo, o pior caso ocorre quando ao vetor possui valores em ordem crescente.
    - O valor de "M" é sempre será substituído a cada avaliação: maior número de instruções;
    - O laço "for" sempre executa as 2 instruções "n" vezes ($\boldsymbol{2n}$);

  - Assim, a função custo do algoritmo será $\boldsymbol{f(n) = 3 + 2n + 2n}$ ou $\boldsymbol{f(n) = 4n + 3}$.
  - Essa função representa o custo do algoritmo em relação ao tamanho do vetor ("n") de entrada no pior caso.

Análise de Algoritmos

- A **Complexidade assintótica** é definida pelo crescimento da complexidade para entradas suficientemente grandes.

- O comportamento assintótico de um algoritmo é o mais procurado, já que, para um volume grande de dados, a complexidade torna-se mais importante. O algoritmo assintóticamente mais eficiente é melhor para todas as entradas, exceto talvez para entradas relativamente pequenas.

- Além disso, a complexidade exata costuma conter muita informação, resultando em expressões como $\boldsymbol{f(n) = 3 + 2n + 2n}$. Tais expressões podem ser de difícil análise e manipulação. Por essas razões, a análise de complexidade de algoritmos costuma concentrar-se em **comportamento assintótico**.

Análise de Algoritmos

- **Comportamento assintótico**

- No algoritmo abaixo vimos que seu custo é dado pela função $\boldsymbol{f(n) = 4n + 3}$

```
/*1*/ int M = V[0];
/*2*/ for(i = 0; i < n; i++){
/*3*/   if(V[i] >= M){
/*4*/       M = V[i];
/*5*/   }
/*6*/}
```

O comportamento assintótico analisa a função de custo quando "n" tende para o infinito.

- Essa é a função de **complexidade de tempo**. Ela nos dá uma ideia do custo de execução do algoritmo para um problema de tamanho "n".

Análise de Algoritmos

- Comportamento assintótico – Dúvida:
  - Será que todos os termos da função f são necessários para termos uma noção do custo?
  - De fato, nem todos os termos são necessários;
  - Podemos descartar certos termos na função e manter apenas os que nos informam o que acontece com a função quando o tamanho dos dados de entrada ("n"), cresce muito;
  - Se um algoritmo é mais rápido do que outro para um grande conjunto de dados de entrada, é muito provável que ele continue sendo também mais rápido em um conjunto de dados menor.

Análise de Algoritmos

- Comportamento assintótico
  - Podemos descartar todos os termos que crescem lentamente, e manter apenas os que crescem mais rápido a medida que o valor de "n" se torna maior;
  - A função $\boldsymbol{f(n) = 4n + 3}$, possui dois termos:
    - $\boldsymbol{4n}$
    - $\boldsymbol{3}$
  - O termo 3 é uma **constante de inicialização**, e não se altera a medida que "n" aumenta;
  - Assim, nossa função pode ser reduzida para $\boldsymbol{f(n) = 4n}$.

Análise de Algoritmos

- Comportamento assintótico
  - Constantes que multiplicam o termo “n” da função também devem ser descartadas;
  - Isso faz sentido se pensarmos em diferentes linguagens de programação. Por exemplo, a seguinte linha de código em Pascal:

```
M := V[i];
```

  - Equivale ao seguinte código em linguagem C:

```
if(i >= 0 && i < n){
    M = V[i];
}
```

  - O acesso a um elemento do vetor:
    - Pascal: 1 instrução (a linguagem possui etapas internas de verificação);
    - C: para atingir o mesmo efeito C precisa de 3 instruções (C não tem etapa de verificação).

Análise de Algoritmos

- Comportamento assintótico

  - Ignorar essas constantes de multiplicação equivale a ignorar as particularidades de cada linguagem e compilador, e analisar apenas a ideia do algoritmo;

Particularidades da linguagem

$$\cancel{4}n + \cancel{3}$$

Constantes de inicialização

  - Assim, nossa função pode ser reduzida ainda mais para:

$$f(n) = n$$

Análise de Algoritmos

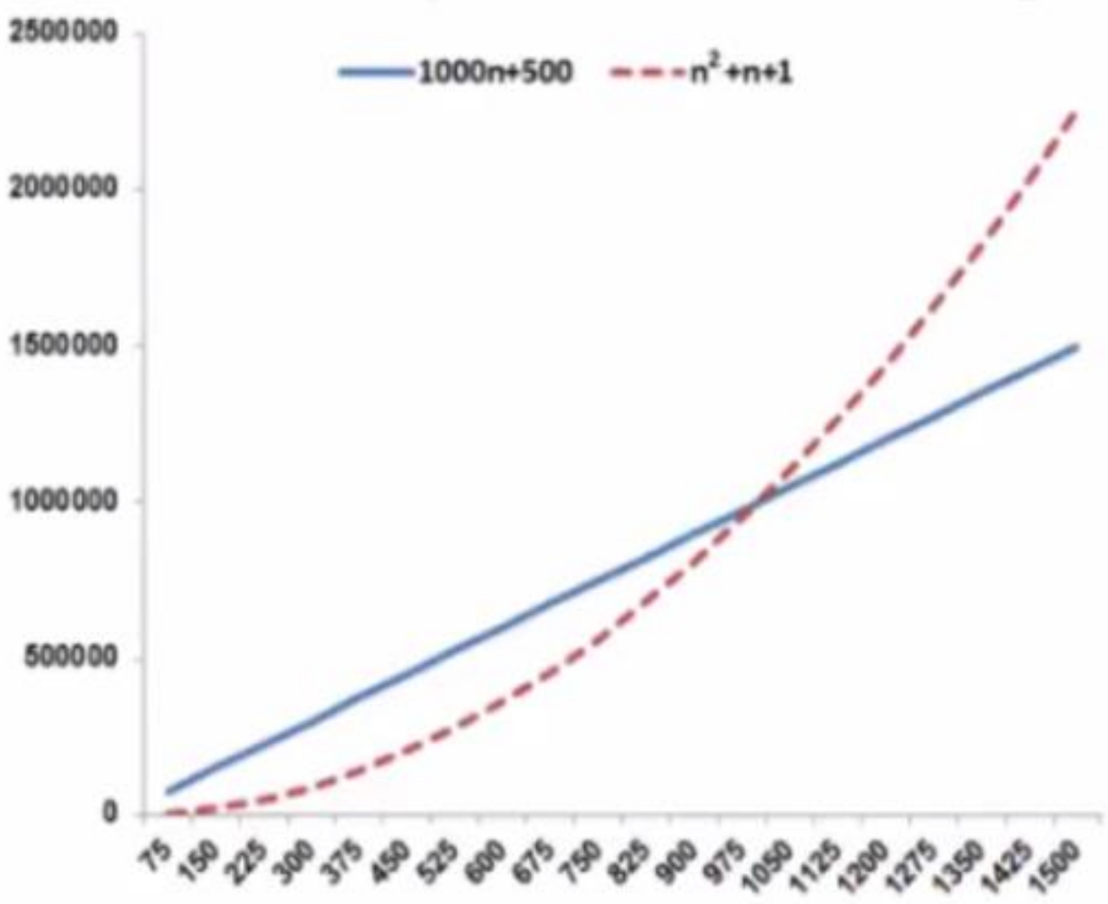
- Comportamento assintótico

  - Descartando todos os termos constantes e mantendo apenas o de maior crescimento, obtemos o **comportamento assintótico**;
  - Trata-se do comportamento de uma função $\boldsymbol{f(n)}$ quando $\boldsymbol{n}$ tende ao infinito;

  - Isso acontece porque o termo que possui o maior expoente domina o comportamento da função $\boldsymbol{f(n)}$ quando “n” tende ao infinito.

Análise de Algoritmos

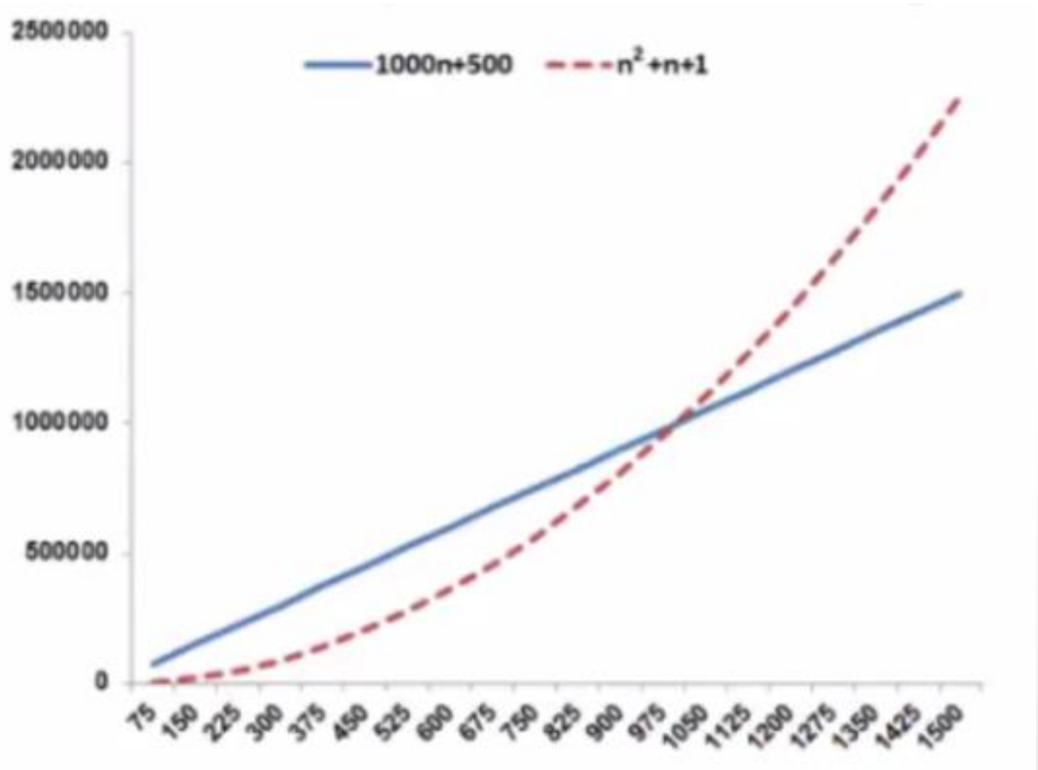
- Comportamento assintótico
  - Para entender melhor, considere duas funções:
    - $g(n) = 1000n + 500$
    - $h(n) = n^2 + n + 1$



Apesar da função $g(n)$ possuir constantes maiores multiplicando seus termos, existe um valor para n, a partir do qual o resultado de $h(n)$ será sempre maior do que $g(n)$, tornando os demais termos e constantes pouco importantes.

Análise de Algoritmos

- Comportamento assintótico
  - Podemos então suprimir os termos menos importantes da função e considerar apenas o termo de maior grau.
  - Assim, podemos descrever a complexidade usando somente o seu custo dominante:
  - $\boldsymbol{n}$ para a função $\boldsymbol{g(n)}$;
  - $\boldsymbol{n}^2$ para $\boldsymbol{h(n)}$



Análise de Algoritmos

- Comportamento assintótico
  - Abaixo podemos ver alguns exemplos de função de custo juntamente com seu comportamento assintótico;
  - Se a função não possui nenhum termo multiplicado por $\boldsymbol{n}$, seu comportamento assintótico é constante (1).

Função de Custo	**Comportamento assintótico**
$f(n) = 227$	$f(n) = 1$
$f(n) = 15n + 4$	$f(n) = n$
$f(n) = n^2 + 3n + 8$	$f(n) = n^2$
$f(n) = 8n^3 + 700n^2 + 467$	$f(n) = n^3$

Análise de Algoritmos

- Comportamento assintótico
  - De modo geral, podemos obter a função de custo de um programa simples apenas contando os comandos dos laços aninhados;
  - Exemplos:
  - Algoritmos sem laço: número constante de instruções, exceto se houver recursão, ou seja $\boldsymbol{f(n) = 1}$;
  - Com um laço indo de 1 a $\boldsymbol{n}$ será $\boldsymbol{f(n) = n}$, ou seja, um conjunto de instruções constantes antes e/ou depois do laço e um conjunto de instruções constantes dentro do laço;
  - Dois comandos de laço aninhados será $\boldsymbol{f(n) = n^2}$, e assim por diante.

Análise de Algoritmos

- Notação **Grande-O** ou “**O**” (big O)

  - Existem várias formas de análise assintótica;

  - A mais conhecida e utilizada é a notação **Grande-O**, “**O**”;
  - Representa o custo do algoritmo (seja no tempo ou no espaço), no pior caso possível, para todas as entradas de tamanho “n”;

  - Ou seja, ela analisa o limite superior de entrada;
  - Desse modo, podemos dizer que o comportamento do nosso algoritmo não pode nunca ultrapassar um determinado limite.

Análise de Algoritmos

- Como exemplo, utilizaremos o algoritmo SelectioSort:

```
void selectionSort(int *V, int n){
    int i, j, me, troca;
    for(i = 0; i < n - 1; i++){
        me = i;
        for(j = i + 1; j < n; j++){
            if(V[j] < V[me]){
                me = j;
            }
        }
        if(i != me){
            troca = V[i];
            V[i]  = V[me];
            V[me] = troca;
        }
    }
}
```

- Dado um vetor “V” de tamanho “n”, procurar o menor valor (posição “me”) e colocar na primeira posição;
- Repetir o processo para a segunda posição, depois para a terceira, e assim por diante;
- Parar quando o vetor estiver ordenado.

Análise de Algoritmos

- Como exemplo, utilizaremos o algoritmo SelectionSort:

```c
void selectionSort(int *V, int n){
    int i, j, me, troca;
    for(i = 0; i < n - 1; i++){
        me = i;
        for(j = i + 1; j < n; j++){
            if(V[j] < V[me]){
                me = j;
            }
        }
        if(i != me){
            troca = V[i];
            V[i]  = V[me];
            V[me] = troca;
        }
    }
}
```

- Temos dois comandos de laço;
- Laço externo: executado “n” vezes;
- Laço interno: número de execuções depende do valor do índice do laço externo
  (n -1, n - 2, n - 3, ..., 2, 1)

para cada iteração do laço externo, o laço interno itera n-1, n-2, ..., 1 vezes

Análise de Algoritmos

- SelectionSort como exemplo:
- Para calcularmos o custo do algoritmo SelectionSort, temos que calcular o resultado da soma:

$$1 + 2 + ... + (n - 1) + n$$

- Essa soma representa o número de execuções do laço interno, algo que não é tão simples de se calcular;
- Dependendo do algoritmo, calcular o seu custo exato pode ser uma tarefa muito complicada;
- Em nosso caso, a soma $1 + 2 + ... + (n - 1) + n$ equivale a soma dos “n” termos de uma progressão aritmética de razão 1:

$$1 + 2 + ... + (n - 1) + n \Rightarrow \frac{n(1+n)}{2}$$

Análise de Algoritmos

- SelectionSort como exemplo:
  - Sabemos agora que o número de execuções do laço interno é $\frac{n(1+n)}{2}$;
  - Uma alternativa mais simples seria estimar um limite superior;
  - A ideia é alterar o algoritmo para algo “menos eficiente” do que temos;
  - Assim, saberemos que o algoritmo original é no máximo tão ruim, ou talvez melhor, que o novo algoritmo.

Análise de Algoritmos

- SelectionSort como exemplo:
  - Podemos diminuir a eficiência do SelectionSort trocando o laço interno, que muda de tamanho a cada execução do laço externo, por um laço que seja executado sempre todas as "n" vezes;
  - Isso simplifica a análise do custo do algoritmo, mas também piora seu desempenho, já que algumas execuções do laço interno serão inúteis;
  - Agora temos dois comandos de laço aninhados sendo executados todas as "n" vezes cada;
  - A função de custo passa a ser $\boldsymbol{f(n) = n^2}$;
  - Utilizando a notação **grande-O**, podemos dizer que o custo do algoritmo no pior caso é $O(n^2)$.

Análise de Algoritmos

- SelectionSort como exemplo:

  - A notação $\boldsymbol{O}(\boldsymbol{n}^2)$ nos diz que o custo do algoritmo não é **assintóticamente**, pior do que $\boldsymbol{n}^2$;

...com o "n" crescendo para o infinito...

  - Em outras palavras, o custo do algoritmo original é no máximo, tão ruim quanto $\boldsymbol{n}^2$;
  - Pode ser melhor, mas nunca pior;
  - Assim, com a notação **grande-O** podemos estabelecer um **limite superior** para a complexidade real de um algoritmo;
  - Isso significa que o nosso programa nunca será mais lento do que um determinado limite.

Análise de Algoritmos

- Tipos de análise assintótica
  - Existem várias formas de análise assintótica;
  - A mais conhecida e utilizada é a notação Grande-O, que representa a complexidade do nosso algoritmo no pior caso;
  - A notação Grande-O é a mais utilizada pois é o caso mais fácil de se identificar (limite superior sobre o tempo de execução do algoritmo). Para diversos algoritmos o pior caso acontece com frequência, por isso é a mais usada.

Análise de Algoritmos

- Tipos de análise assintótica – Tipos mais utilizados:
  - Notação **Grande-Omega**;
    - Descreve o limite assintótico inferior – analisa o melhor caso do algoritmo;
  - Notação **Grande-O**;
    - Descreve o limite assintótico superior – analisa o pior caso de um algoritmo;
  - Notação **Grande-Theta**;
    - Descreve o limite assintótico firme, ou estreito – analisa o limite inferior e superior de um algoritmo, em outras palavras por exemplo, o custo do algoritmo original é x dentro de um fator constante **acima** e **abaixo**
  - Notação **Pequeno-o** e Pequeno-ômega.
    - Parecidas com Grande-O e Grande Ômega, possuem relação de maior ou igual e menor ou igual. Não representam limites próximos da função, mas estritamente com superiores sempre maiores e inferiores sempre menores.

Análise de Algoritmos

- Notação Grande-O – Regra da Soma;
  - Importante na análise da complexidade de diferentes algoritmos em sequência;
  - Basicamente, se dois algoritmos são executados em sequência, a complexidade da execução será dada pela complexidade do maior deles:

$$\boldsymbol{O}\big(\boldsymbol{f}(\boldsymbol{n})\big) + \boldsymbol{O}\big(\boldsymbol{g}(\boldsymbol{n})\big) = \boldsymbol{O}(\boldsymbol{max}(\boldsymbol{f}(\boldsymbol{n}), \boldsymbol{g}(\boldsymbol{n}))$$

`max()` é uma função que recebe dois parâmetros e devolve o maior deles.

- Exemplos:
  - A execução de dois algoritmos, $\boldsymbol{O}(\boldsymbol{n})$ e $\boldsymbol{O}(\boldsymbol{n}^2)$, em sequência é $\boldsymbol{O}(\boldsymbol{n}^2)$;
  - A execução de dois algoritmos, $\boldsymbol{O}(\boldsymbol{n})$ e $\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$, em sequência é $\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$;
  - A execução de três algoritmos, $\boldsymbol{O}(\boldsymbol{n})$, $\boldsymbol{O}(\boldsymbol{n}\log\boldsymbol{n})$ e $\boldsymbol{O}(\boldsymbol{n}^2)$, em sequência é $\boldsymbol{O}(\boldsymbol{n}^2)$.

Análise de Algoritmos

- Classes de Problemas

  - A seguir temos algumas classes de complexidade de problemas comumente usadas:
  - $\boldsymbol{O}(\mathbf{1})$: Ordem constante
    - As instruções são executadas um número fixo de vezes. Não depende do tamanho dos dados de entrada.
  - $\boldsymbol{O}(\mathbf{log}(\boldsymbol{n}))$: Ordem logarítmica
    - Típica de algoritmos que resolvem um problema transformando-o em problemas menores;
  - $\boldsymbol{O}(\boldsymbol{n})$: Ordem linear
    - Em geral, uma certa quantidade de operações é realizada sobre cada um dos elementos de entrada.

Análise de Algoritmos

- Classes de Problemas
  - $\boldsymbol{O}(\boldsymbol{n}\log(\boldsymbol{n}))$: Ordem log linear
    - Típica de algoritmos que trabalham com particionamento de dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos;
  - $\boldsymbol{O}(\mathbf{n}^2)$: Ordem quadrática
    - Normalmente ocorre quando os dados são processados aos pares. Uma característica deste tipo de algoritmos é a presença de um aninhamento de dois comandos de repetição.

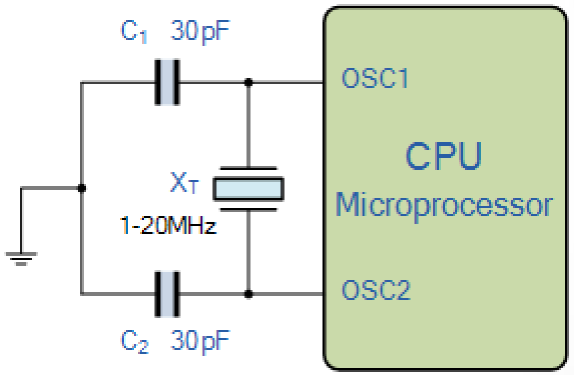
Análise de Algoritmos

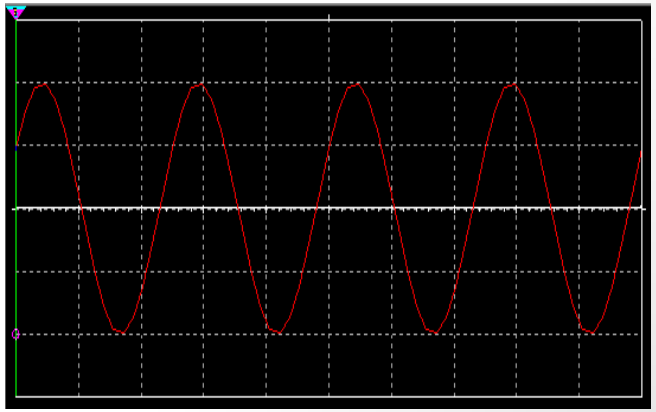
- Classes de Problemas
  - $\boldsymbol{O(n^3)}$: Ordem cúbica
    - É caracterizado pela presença de três estruturas de repetição aninhadas
  - $\boldsymbol{O(2^n)}$: Ordem exponencial
    - Geralmente ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático;
  - $\boldsymbol{O(n!)}$: Ordem fatorial
    - Geralmente ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático. Possui um comportamento muito pior do que o exponencial.

Análise de Algoritmos

- Classes de Problemas
- Comparação do tempo de execução: Considerando que um ipotético computador é capaz de executar um milhão de operações por segundo – 1 MHz para o *clock* do processador.

f(n)	n = 10	n = 20	n = 30	n = 50	n = 100
n	1,0E-05 segundos	2,0E-05 segundos	4,0E-05 segundos	5,0E-05 segundos	6,0E-05 segundos
$n \log n$	3,3E-05 segundos	8,6E-05 segundos	2,1E-04 segundos	2,8E-04 segundos	3,5E-04 segundos
n^2	1,0E-04 segundos	4,0E-04 segundos	1,6E-03 segundos	2,5E-03 segundos	3,6E-03 segundos
n^3	1,0E-03 segundos	8,0E-03 segundos	6,4E-02 segundos	0,13 segundos	0,22 segundos
2^n	1,0E-03 segundos	1,0 segundo	2,8 dias	35,7 anos	365,6 séculos
3^n	5,9E02 segundos	58,1 minutos	3855,2 séculos	2,3E+08 séculos	1,3E+13 séculos





Análise de Algoritmos

- Os algoritmos de ordenação básicos que vimos, têm um tempo de execução que cresce na ordem de $\boldsymbol{n^2}$. Os algoritmos mais sofisticados (mais rápidos), têm o crescimento do tempo de execução na ordem de $\boldsymbol{n \log n}$.

- Mas, o que significa $\boldsymbol{\log n}$?
  - É o logaritmo na base 2 de $\boldsymbol{n}$, ou $\boldsymbol{\log_2 n}$. Cientistas da computação trabalham com logaritmos na base 2 com tanta frequência, que utilizam sua própria abreviatura para logaritmos na base 2: $\boldsymbol{\log n}$;
  - Como a função $\boldsymbol{\log n}$, é o inverso de uma função exponencial, ela cresce muito lentamente com $\boldsymbol{n}$. Se $\boldsymbol{n = 2^x}$, então $\boldsymbol{x = \log n}$, por exemplo:
    - $\boldsymbol{2^{10} = 1024}$, logo $\boldsymbol{\log 1024} = 10$
    - $\boldsymbol{2^{20} = 1.048.576}$, logo $\boldsymbol{\log 1.048.576 = 20}$

Análise de Algoritmos

- Como exemplo mais concreto, confrontando dois hipotéticos computadores. O computador **A,** que executa um algoritmo de ordenação cujo tempo de execução para $\boldsymbol{n}$ valores cresce segundo $\boldsymbol{n}^2$**,** com um computador mais lento (**B**), que executa um algoritmo de ordenação cujo tempo de execução cresce segundo $\boldsymbol{n \log n}$**.** Cada um deles deve ordenar um vetor de 10 milhões de elementos.
- Vamos supor que o computador **A** execute 10 bilhões de instruções por segundo, e que o computador **B** execute somente 10 milhões de instruções por segundo, de modo que o computador **A** é mil vezes mais rápido do que o computador B, em poder de computação bruto.
- Ainda no campo da suposição, o algoritmo de ordenação do computador **A**, foi desenvolvido por um programador experiente que utilizou a linguagem *assembly* e seu custo computacional ficou em $\boldsymbol{2n}^2$ instruções para ordenar $\boldsymbol{n}$ números, e para o computador **B**, um programador mediano utilizou uma linguagem de alto nível e o código resultante tenha $\boldsymbol{50\, n \log n}$ instruções.

Análise de Algoritmos

- Para ordenar 10 milhões de elementos, o computador **A** leva:
  - $$\frac{2 * (10^7)^2 instruções}{10^{10}\ instruções/segundo} = 20.000\ segundos$$
- O que é mais de 5,5 horas, enquanto o computador **B** leva:
  - $$\frac{50 * 10^7 \log 10^7\ instruções}{10^7\ instruções/segundo} \simeq 350\ segundos$$
- O que é um pouco menos de 6 minutos. Utilizando um algoritmo cujo tempo de execução cresce mais lentamente, o computador **B** executa a ordenação 57 vezes mais rapidamente que o computador **A**!
- A vantagem do algoritmo $\boldsymbol{n \log n}$, é ainda mais pronunciada quando a ordenação é elevada para 100 milhões de elementos: enquanto o computador **A** com o algoritmo $\boldsymbol{n^2}$ leva mais de 23 dias, o algoritmo $\boldsymbol{n \log n}$ do computador **B** leva pouco mais de 1 hora.

Estrutura de Dados 2

Atividade 1

- Calcule a complexidade, no pior caso, do seguinte fragmento de código:

```
int i, j, k,
for(i = 0; i < n; i++){
    for(j = 0; j < n; j++){
        r[i][j] = 0;
        for(k = 0; k < n; k++){
            r[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

- Transcreva o fragmento para um arquivo texto (pode ser no próprio CodeBlocks, ou qualquer editor que desejar), juntamente com a resposta e entregue no Moodle como atividade 1.

Atividade 2

- Calcule a complexidade, no pior caso, do seguinte fragmento de código:

```
int i, j, k, s;
for(i = 0; i < n - 1, i++){
    for(j = i + 1; j < n; j++){
        for(k = 1; k < j; k++){
            s = 1;
        }
    }
}
```

- Transcreva o fragmento para um arquivo texto (pode ser no próprio CodeBlocks, ou qualquer editor que desejar), juntamente com a resposta e entregue no Moodle como atividade 2.

Atividade 3

- Calcule a complexidade, no pior caso, do seguinte fragmento de código:

```c
int i, j, s;
s = 0;
for(i = 1; i < n - 1; i++){
    for(j = 1; j < 2 * n; j++){
        s = s + 1;
    }
}
```

- Transcreva o fragmento para um arquivo texto (pode ser no próprio CodeBlocks, ou qualquer editor que desejar), juntamente com a resposta e entregue no Moodle como atividade 3.