ANÁLISE DE ALGORITMOS

Prof. André Backes | @progdescomplicada

Algoritmos

- Como resolver um problema no computador?
  - Precisamos descrevê-lo de uma forma clara e precisa

- Precisamos escrever o seu **algoritmo**
  - Um **algoritmo** é uma sequência simples e objetiva de **instruções**
  - Cada **instrução** é uma informação que indica ao computador uma ação básica a ser executada

Algoritmo: Bolo de Chocolate

- Aqueça o forno a 180 C
- Unte uma forma redonda
- Numa taça
  - Bata
    - 75g de manteiga
    - 250g de açúcar
  - até ficar cremoso
  - Junte
    - 4 ovos, um a um
    - 100g de chocolate derretido
  - Adicione aos poucos 250g de farinha peneirada
- Deite a massa na forma
- Leve ao forno durante 40 minutos

Algoritmos

- Vários algoritmos para um mesmo problema
  - Os algoritmos se diferenciam uns dos outros pela maneira como eles utilizam os recursos do computador
- Os algoritmos dependem
  - Principalmente do tempo que demora pra ser executado
  - Da quantidade de memória do computador

Análise de algoritmos

- Área de pesquisa cujo foco são os algoritmos
  - Busca responder a seguinte pergunta: **podemos fazer um algoritmo mais eficiente?**
  - Algoritmos diferentes mas capazes de resolver o mesmo problema não necessariamente o fazem como a mesma eficiência
    - Exemplo: ordenação de números

Análise de algoritmos

- As diferenças de eficiência podem ser
  - **Irrelevantes** para um pequeno número de elementos processados
  - **Crescer proporcionalmente** com o número de elementos processados
- Dependendo do tamanho dos dados e da eficiência, um programa poderia executar
  - Instantaneamente
  - De um dia para o outro
  - Por séculos

Análise de algoritmos

- **Complexidade computacional**
  - Medida criada para comparar a eficiência dos algoritmos
  - Indica o **custo** ao se aplicar um algoritmo

    custo = memória + tempo

    - **memória**: quanto de espaço o algoritmo vai consumir
    - **tempo**: a duração de sua execução

Análise de algoritmos

- Importante
  - O custo pode estar associado a outros recursos computacionais, além da memória
    - Exemplo: tráfego de rede
  - No entanto, para a maior parte dos problemas o custo está relacionado ao tempo de execução em função do **tamanho da entrada** a ser processada

Análise de algoritmos

- Para determinar se um algoritmo é o mais eficiente, podemos utilizar duas abordagens
  - **Análise empírica**
    - Comparação entre os programas
  - **Análise matemática**
    - Estudo das propriedades do algoritmo

Análise empírica

- Definição
  - Avalia o custo (ou complexidade) de um algoritmo a partir da avaliação da execução do mesmo quando implementado
  - Análise pela execução de seu programa correspondente

Análise empírica

- Exemplo: calcular o tempo de execução

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
    clock_t inicio, fim;
    unsigned long int tempo;
    inicio = clock();
    //============================
    /* coloque seu código aqui */
    //============================
    fim = clock();
    tempo = (fim - inicio)*1000/CLOCKS_PER_SEC;
    printf("tempo: %lu milissegundo\n",tempo);

    return 0;
}
```

Análise empírica

- Vantagens
  - Permite avaliar o desempenho em uma determinada configuração de computador/linguagem
  - Considera custos não aparentes
    - Por exemplo, o custo da alocação de memória
  - Permite comparar computadores
  - Permite comparar linguagens

Análise empírica

- Desvantagens
  - Necessidade de implementar o algoritmo
    - Depende da habilidade do programador
  - Resultado pode ser mascarado
    - Hardware: computador utilizado
    - Software: eventos ocorridos no momento de avaliação
  - Depende da natureza dos dados
    - **Dados reais**
    - **Dados aleatórios**: desempenho médio
    - **Dados perversos**: desempenho no pior caso

Análise matemática

- Muitas vezes, é preferível que a medição do tempo gasto por um algoritmo seja feita de maneira independente
  - Neste caso, devemos desconsiderar o hardware ou a linguagem de programação usada

- Nesse tipo de situação, convém utilizar a **análise matemática** do algoritmo

Análise matemática

- Permite um estudo formal de um algoritmo ao nível da sua idéia
  - Faz uso de um computador idealizado e simplificações que buscam considerar somente os custos dominantes do algoritmo
  - Detalhes de baixo nível são ignorados
    - Linguagem de programação utilizada
    - Hardware no qual o algoritmo é executado
    - Conjunto de instruções da CPU
    - Etc

Análise matemática

- Permite entender como um algoritmo se comporta à medida que o conjunto de dados de entrada cresce
  - Exemplo: número de elementos em um array, lista, árvore, etc
- Expressa a relação entre o conjunto de dados de entrada e a quantidade de tempo necessária para processar esses dados

Contando instruções

- Considere o pequeno trecho de código
  - Este algoritmo procura o maior valor presente em um array **A** contendo **n** elementos e o armazena na variável **M**

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Quantas **instruções simples** ele executa?
  - **Instruções simples**: instruções que podem ser executadas diretamente pelo CPU (ou algo muito perto disso)
    - atribuição de um valor a uma variável
    - acesso ao valor de um determinado elemento do array
    - comparação de dois valores
    - incremento de um valor
    - operações aritméticas básicas, como adição e multiplicação

Contando instruções

- Instruções simples
  - Todas possuem o mesmo custo
  - Comandos de seleção (como o comando **if**) possuem custo zero
    - Não contam como instruções

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Custo da inicialização de **M**: 1 instrução
  - Apenas uma operação de atribuição

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Custo de inicialização do laço **for**: 2 instruções
  - Uma operação de atribuição e uma comparação

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Custo de execução do laço **for**: 2n instruções
  - Uma operação de incremento e uma comparação executadas **n** vezes

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Ignorando os comandos contidos dentro do laço **for**, temos que o algoritmo irá executar **3+2n** instruções
  - 3 instruções antes de iniciar o laço **for**
  - 2 instruções ao final de cada laço **for**

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Assim, considerando um **laço vazio**, podemos definir uma função matemática que representa o custo do algoritmo em relação ao tamanho do array de entrada
  - **f(n) = 2n + 3**

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- As instruções vistas anteriormente eram sempre executadas.
- Porém, as instruções dentro do **for** podem ou não ser executadas
  - Comando de seleção: 1 instrução
    - Sempre executada
  - Atribuição: 1 instrução
    - Depende do resultado do comando de seleção

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Antes, bastava saber o tamanho do array, **n**, para definir a função de custo **f(n)**
- Agora, temos que considerar também o conteúdo do array
- Tome como exemplo os dois arrays abaixo

```
int A1[4] = {1, 2, 3, 4};
int A2[4] = {4, 3, 2, 1};
```

Contando instruções

- O array **A1** irá executar mais instruções do que o array **A2**
  - Array **A1**: o comando **if** é sempre **verdadeiro**
  - Array **A2**: o comando **if** é sempre **falso**

- Devemos considerar o pior caso possível
  - Maior número de instruções é executado

```
int A1[4] = {1, 2, 3, 4};
int A2[4] = {4, 3, 2, 1};
```

Contando instruções

- Neste exemplo, o **pior caso** ocorre quando o array possui valores em ordem crescente
  - Valor de **M** é sempre substituído: 2n instruções
    - Maior número de instruções
  - A função custo será, no **pior caso**,
    - **f(n) = 3 + 2n + 2n** ou
    - **f(n) = 4n + 3**

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Contando instruções

- Exemplo: maior valor presente em uma matriz **A** contendo **n x n** elementos

```
M = A[0][0];

for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        if(M < A[i][j])
            M = A[i][j];
```

$$f(n) = 1 + 2 + 2n + n(2 + 2n + 2n)$$

$$f(n) = 3 + 2n + 2n + 2n^2 + 2n^2$$

$$f(n) = 3 + 4n + 4n^2$$

COMPORTAMENTO ASSINTÓTICO

Comportamento assintótico

- Vimos que o custo para o algoritmo abaixo é dado pela função
- **f(n) = 4n + 3**

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Comportamento assintótico

- Essa é a função de complexidade de tempo
  - Nos dá uma ideia do custo de execução do algoritmo para um problema de tamanho **n**
    - Exemplo: array de **n** elementos
  - É possível criar o mesmo tipo de função para a análise do espaço gasto

```
int M = A[0];

for(i = 0; i < n; i++){
    if(A[i] >= M) {
        M = A[i];
    }
}
```

Comportamento assintótico

- Dúvida
  - Será que todos os termos da função **f(n)** são necessários para termos uma noção do custo?

- De fato, nem todos os termos são necessários
  - Podemos descartar certos termos na função
  - Devemos manter apenas os que nos dizem o que acontece quando o tamanho dos dados de entrada (**n**) cresce muito

Comportamento assintótico

- Idéia geral
  - 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
  - Assim, podemos
    - Descartar todos os termos que crescem lentamente
    - Manter apenas os que crescem mais rápido à medida que o valor de **n** se torna maior

Comportamento assintótico

- A função **f(n) = 4n + 3** possui dois termos
  - **4n** e **3**

- O termo **3** é uma **constante de inicialização**
  - Não se altera à medida que o valor de **n** cresce
    - Exemplo: atribuições antes de um laço
  - Pode, portanto, ser descartado

- Assim, a função é reduzida para **f(n) = 4n**

Comportamento assintótico

- Constantes que multiplicam o termo **n** da função também devem ser descartadas
  - Representam particularidades de cada linguagem e compilador
  - Queremos analisar apenas a ideia por trás do algoritmo, sem influências da linguagem

Comportamento assintótico

- Exemplo: atribuição em array
  - Na linguagem Pascal, essa operação equivale a um teste lógico e uma atribuição na linguagem C
    - Pascal: 3 instruções (etapa de verificação)
    - C: 1 instrução (sem etapa de verificação)

```
//Pascal
M:= A[i];
//C
if(i >=0 && i < n)
    M = A[i];
```

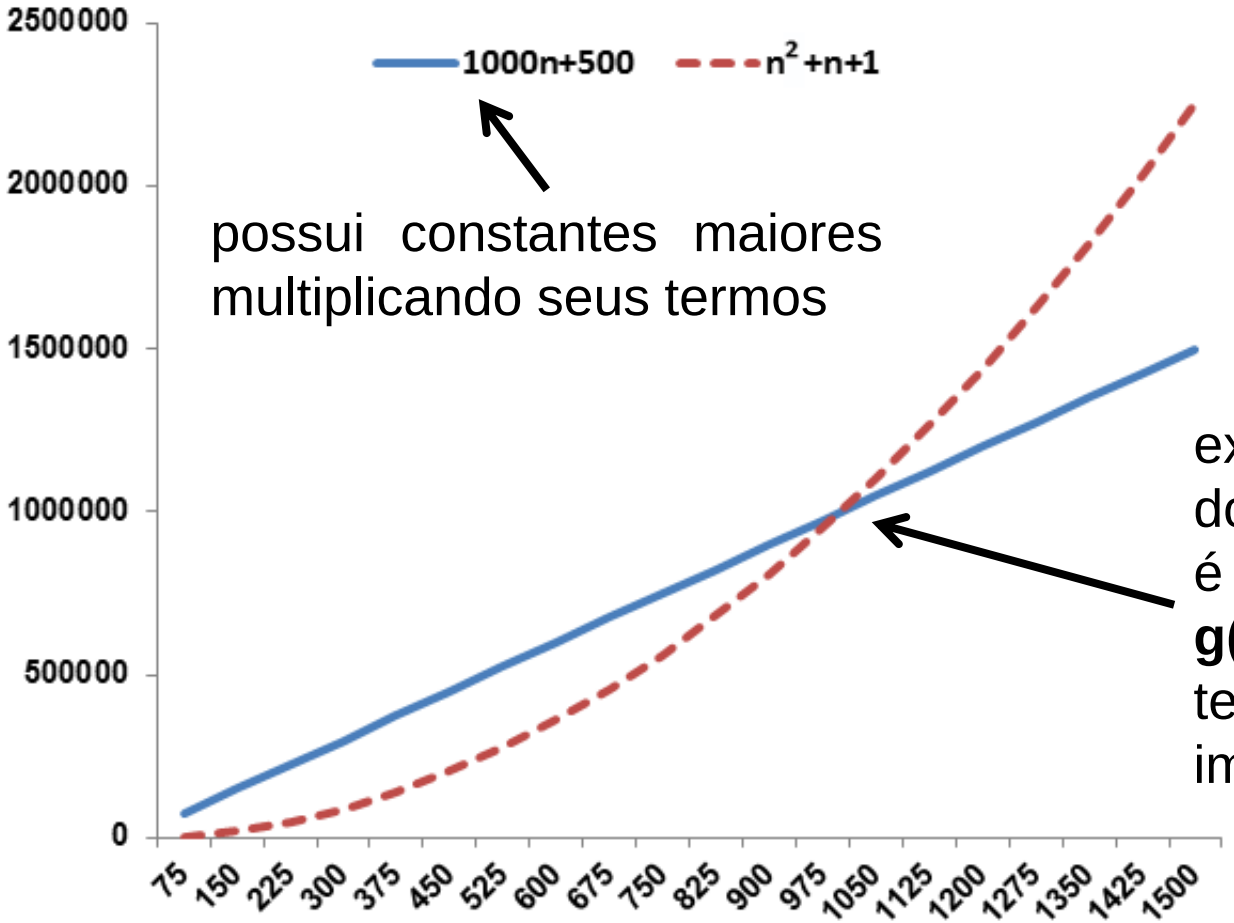
- Assim, a função **f(n) = 4n + 3** é reduzida para **f(n) = n**

Comportamento assintótico

- Descartando todos os termos constantes e mantendo apenas o de maior crescimento obtemos o **comportamento assintótico**
  - Comportamento de uma função **f(n)** quando **n** tende ao infinito
  - O termo de maior expoente domina o comportamento da função

Comportamento assintótico

- Exemplo: **g(n) = 1000n+500** e **h(n) = n^2+n+1**



possui constantes maiores multiplicando seus termos

existe um valor de **n** a partir do qual o resultado de **h(n)** é sempre maior do que **g(n)**, tornando os demais termos e constantes pouco importantes

Comportamento assintótico

- Assim, podemos
  - Suprimir os termos menos importantes da função e considerar apenas o termo de **maior grau**
  - Descrever a complexidade usando somente o seu custo dominante
    - **n** para a função **g(n)**
    - $\mathbf{n^2}$ para **h(n)**

Comportamento assintótico

- Exemplos de função de custo juntamente com o seu comportamento assintótico
  - **Obs**: Se a função não possui nenhum termo multiplicado por **n**, seu comportamento assintótico é constante

Função custo	Comportamento assintótico
$f(n) = 105$	**$f(n) = 1$**
$f(n) = 15n + 2$	**$f(n) = n$**
$f(n) = n^2 + 5n + 2$	**$f(n) = n^2$**
$f(n) = 5n^3 + 200n^2 + 112$	**$f(n) = n^3$**

Comportamento assintótico

- De modo geral, podemos obter a função de custo de um programa simples apenas contando os comandos de laços aninhados
  - Não possui laço (exceto se houver recursão)
    - **f(n) = 1**
  - Um comando de laço indo de 1 a **n**
    - **f(n) = n**
  - Dois comandos de laço aninhados
    - **f(n) = n^2**
  - e assim por diante

Notação Grande-O

- Existem várias formas de análise assintótica
- A mais conhecida e utilizada é a notação **grande-O** (***O***)
  - Custo do algoritmo no **pior caso** possível para todas as entradas de tamanho **n**
  - Analisa o limite superior de entrada
  - Permite dizer que o comportamento do nosso algoritmo não pode nunca ultrapassar um certo limite

Notação Grande-O

- Para entender essa notação, considere o algoritmo de ordenação **selection sort**
  - Dado um array **V** de tamanho **n**, procure o menor valor (posição **me**) e coloque na primeira posição
  - Repetir processo para a segunda posição, depois para a terceira etc.
  - Pare quando o array estiver ordenado

Notação Grande-O

- Implementação do **selection sort**

```
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;
        }
    }
}
```

Notação Grande-O

- Dois comandos de laço no **selection sort**
  - Laço externo: executado **n-1 vezes**
  - Laço interno: número de execuções depende do valor do índice do laço externo (n-1,n-2,n-3,..., 2,1)

Procura o menor valor.
Número de execuções depende do laço externo
n-1 vezes na 1º iteração
n-2 vezes na 2º iteração
etc

```
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;
        }
    }
}
```

Repete o processo para cada posição do array
Executado **n-1** vezes

Notação Grande-O

- Como calcular o custo do **selection sort**?
  - Temos que calcular o resultado da soma
    - **1 + 2 + ... + (n - 1) + n**
  - Essa soma representa o número de execuções do laço interno
  - Dependendo do algoritmo, isso pode ser uma tarefa muito complicada

Notação Grande-O

- Neste caso, temos uma ajuda da matemática
  - A soma
    - **1 + 2 + ... + (n - 1) + n**
  - equivale a soma dos **n** termos de uma progressão aritmética, $\mathbf{S_n}$, de razão 1
  - Assim
    - $\mathbf{S_n = 1 + 2 + ... + (n - 1) + n}$
    - $\mathbf{S_n = n + (n - 1) + ... + 1 + 1}$
    - $\mathbf{2S_n = (1 + n) + (2+(n-1))+ ... + ((n - 1)+2) + (n+1)}$

Notação Grande-O

- Vamos analisar os termos equidistantes dos extremos
  - Como **1** e **n**, **2** e **(n-1)**,...,
  - Suas somas são sempre iguais a **(1 + n)**
  - Logo
    - $\mathbf{2S_n = (1 + n) + (2+(n-1)) + ... + ((n - 1)+2) + (n+1)}$
    - $\mathbf{2S_n = n(1 + n)}$
    - $\mathbf{S_n = n(1 + n)/2}$

Notação Grande-O

- Como resultado, o número de execuções do laço interno é

$$S_n = n(1 + n)/2$$

  - Sabemos agora o número de execuções do laço interno
  - Porém, essa é uma tarefa trabalhosa!

Notação Grande-O

- Uma alternativa mais simples: **estimar um limite superior**
  - Podemos alterar mentalmente o algoritmo e, em seguida, calcular o custo desse novo algoritmo
  - Desse modo, vamos torná-lo **menos eficiente**
    - Assim, saberemos que o algoritmo original é no máximo tão ruim, ou talvez melhor, que o novo algoritmo

Notação Grande-O

- Como diminuir a eficiência do **selection sort**?
  - Podemos trocar o laço interno por um laço que seja executado sempre **n** vezes
    - Fica mais fácil descobrir o custo do algoritmo
    - Piora o desempenho: algumas execuções do laço interno serão inúteis

```
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;
        }
    }
}
```

Notação Grande-O

- Temos agora dois comandos de laço aninhados sendo executados **n** vezes cada
  - Função de custo passa a ser **$f(n) = n^2$**
  - Utilizando a notação **grande-O**, ***O***, podemos dizer que o custo do algoritmo no **pior caso** é ***$O(n^2)$***

Notação Grande-O

- O que ***O(n^2)*** significa para um algoritmo?
  - A notação ***O(n^2)*** nos diz que o custo do algoritmo não é, assintoticamente, pior do que **n^2**
    - Nosso algoritmo nunca vai ser mais lento do que um determinado limite
  - Ou seja, o custo do algoritmo original é no máximo tão ruim quanto **n^2**
    - Pode ser melhor, mas nunca pior
    - **Limite superior** para a complexidade real do algoritmo

TIPOS DE ANÁLISE ASSINTÓTICA

Diferentes tipos de análise assintótica

- A notação grande-O é a forma mais conhecida e utilizada de análise assintótica
  - Complexidade do nosso algoritmo no pior caso
    - Seja de tempo ou de espaço
  - É o caso mais fácil de se identificar
    - Limite superior sobre o tempo de execução do algoritmo
    - Para diversos algoritmos o pior caso ocorre com frequência

Diferentes tipos de análise assintótica

- No entanto, existem várias formas de análise assintótica
  - Notação grande-Omega, Ω
  - Notação grande-O, O
  - Notação grande-Theta, Θ
  - Notação pequeno-o, o
  - Notação pequeno-omega, ω
- A seguir, são matematicamente descritas outras formas de análise assintótica.

Diferentes tipos de análise assintótica

- Notação grande-Omega, Ω
  - Descreve o **limite assintótico inferior**
  - É utilizada para analisar o **melhor caso** do algoritmo
  - A notação $\Omega(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, maior ou igual a n^2
    - Ou seja, o custo do algoritmo original é no mínimo tão ruim quanto n^2

Diferentes tipos de análise assintótica

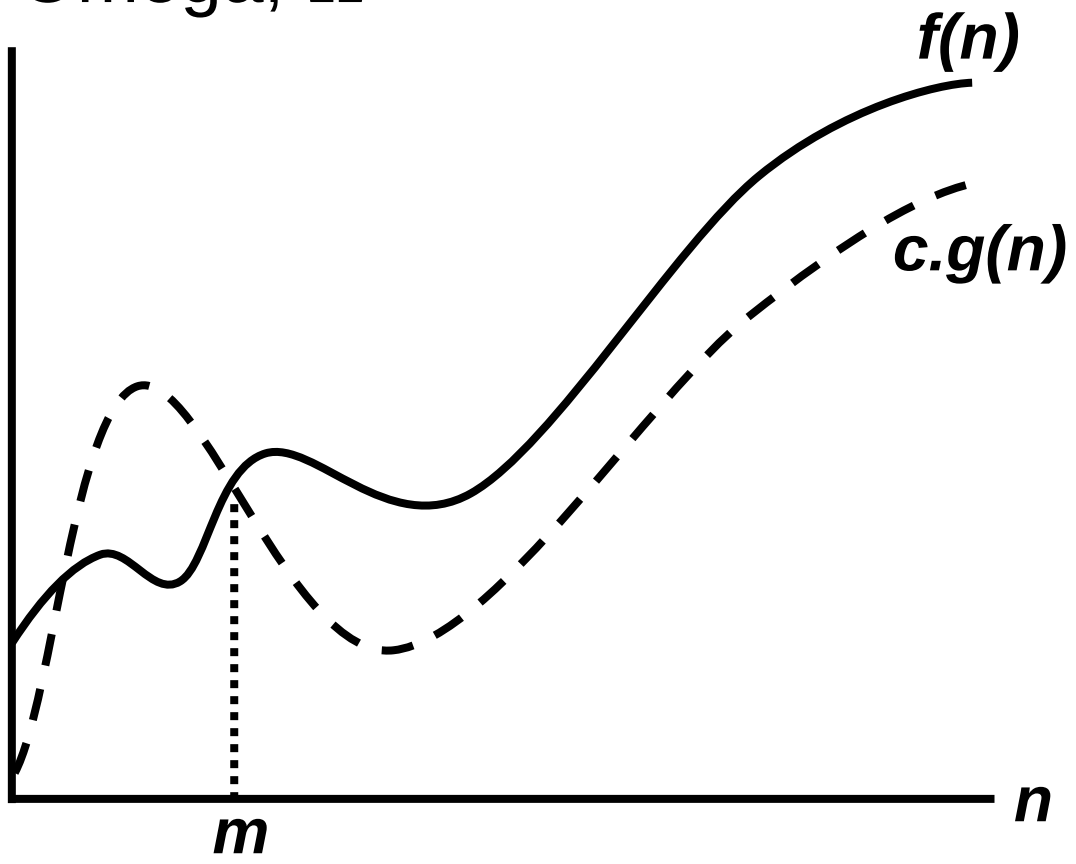
- Notação grande-Omega, **Ω**
  - Matematicamente, a notação **Ω** é assim definida
    - Uma função custo **f(n)** é ***Ω(g(n))*** se existem duas constantes positivas **c** e **m** tais que
    - Para **n ≥ m**, temos **f(n) ≥ c.g(n)**
    - Confuso?

Diferentes tipos de análise assintótica

- Notação grande-Omega, **Ω**
  - Em outras palavras, para todos os valores de **n** à direita de **m**, o resultado da função custo **f(n)** é sempre **maior ou igual** ao valor da função usada na notação **Ω**, **g(n)**, multiplicada por uma constante **c**

Diferentes tipos de análise assintótica

- Notação grande-Omega, Ω



$f(n) = \Omega(g(n))$

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo é **f(n)=3n² + n** é $\Omega(n)$
  - Temos que encontrar constantes **c** e **m** tais que
  - $3n^2 + n \geq cn$
  - Dividindo por **n²**, temos
  - $3 + 1/n \geq c/n$
  - Considerando **c = 4** e **n > 0**, temos que **f(n)=3n²+n** é $\Omega(n)$

Diferentes tipos de análise assintótica

- Notação grande-O, ***O***
  - Descreve o **limite assintótico superior**
  - É utilizada para analisar o **pior caso** do algoritmo
  - A notação $O(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, menor ou igual a n^2
    - Ou seja, o custo do algoritmo original é no máximo tão ruim quanto n^2

Diferentes tipos de análise assintótica

- Notação grande-O, ***O***
  - Matematicamente, a notação ***O*** é assim definida
    - Uma função custo **f(n)** é ***O(g(n))*** se existem duas constantes positivas **c** e **m** tais que
    - Para **n ≥ m**, temos **f(n) ≤ c.g(n)**

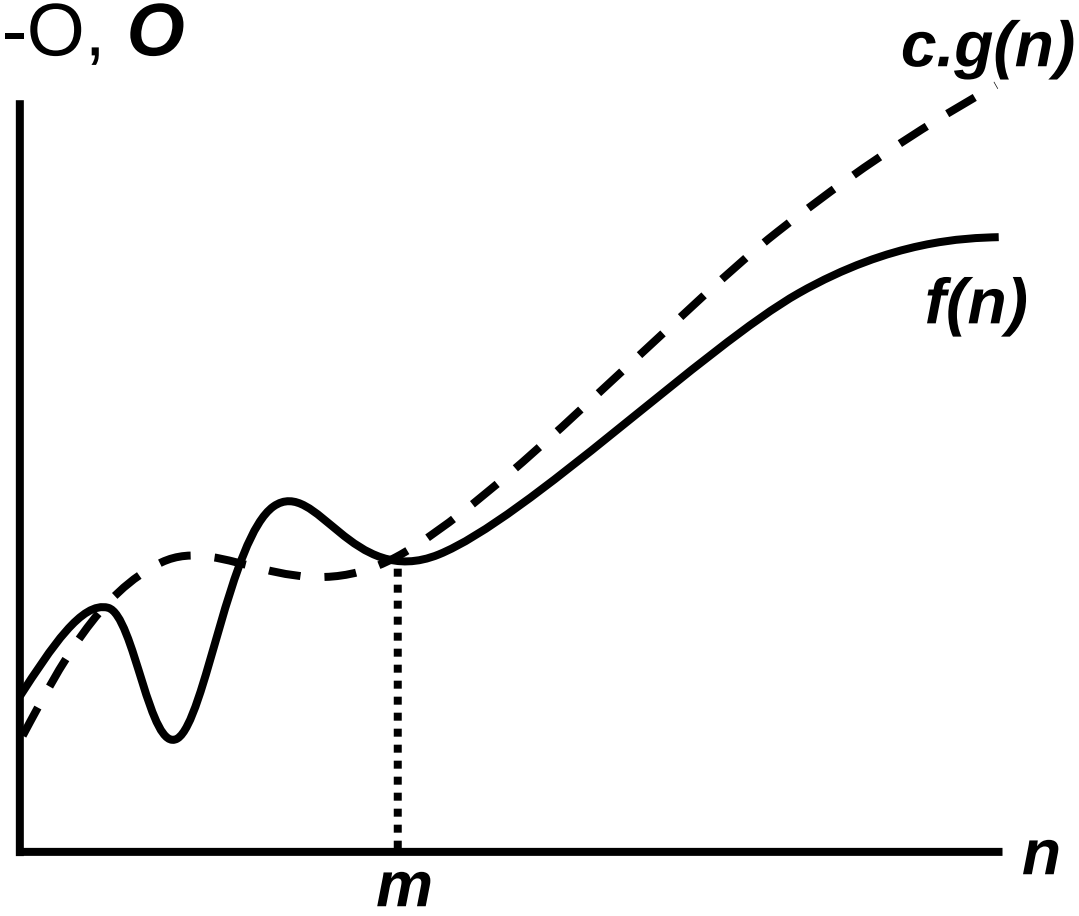
    - Confuso?

Diferentes tipos de análise assintótica

- Notação grande-O, ***O***
  - Em outras palavras, para todos os valores de **n** à direita de **m**, o resultado da função custo **f(n)** é sempre **menor ou igual** ao valor da função usada na notação ***O***, **g(n)**, multiplicada por uma constante **c**.

Diferentes tipos de análise assintótica

- Notação grande-O, ***O***



f(n) = *O(g(n))*

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo é **f(n)=2n² + 10** é ***O(n³)***
  - Temos que encontrar constantes **c** e **m** tais que
  - $2n^2 + 10 \leq cn^3$
  - Dividindo por **n²**, temos
  - $2 + 10/n^2 \leq cn$
  - Considerando **c = 1** e **n > 3**, temos que **f(n)=2n²+10** é ***O(n³)***
    - Dá para melhorar essa análise!

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo é **f(n)=2n² + 10** é ***O(n²)***
  - Temos que encontrar constantes **c** e **m** tais que
  - $2n^2 + 10 \leq cn^2$
  - Dividindo por **n²**, temos
  - $2 + 10/n^2 \leq c$
  - Considerando **c = 12** e **n > 0**, temos que **f(n)=2n²+10** é ***O(n²)***

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo é **f(n)=4n + 7** é ***O(n)***
  - Temos que encontrar constantes **c** e **m** tais que
  - $4n + 7 \leq cn$
  - Dividindo por **n**, temos
  - $4 + 7/n \leq c$
  - Considerando **c = 8** e **n > 1**, temos que **f(n)=4n+7** é ***O(n)***

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo é **f(n)=n² não** é ***O(n)***
  - Temos que encontrar constantes **c** e **m** tais que
  - $n^2 \leq cn$
  - Dividindo por **n**, temos
  - $n \leq c$
- A desigualdade é inválida!
  - O valor de **n** está limitado pela constante **c**
  - A análise assintótica não é possível (entrada tendendo ao infinito)

Diferentes tipos de análise assintótica

- Notação grande-O, ***O***
  - Essa notação possui algumas operações
  - A mais importante é a **regra da soma**
    - Permite a análise da complexidade de diferentes algoritmos em sequência
  - Definição
    - Se dois algoritmos são executados em sequência, a complexidade será dada pela complexidade do maior deles
    - ***O(f(n)) + O(g(n)) = O(max(f(n),g(n)))***

Diferentes tipos de análise assintótica

- Notação grande-O, ***O***
  - Exemplo da ***regra da soma***. Se temos
    - Dois algoritmos cujos tempos de execução são ***O(n)*** e ***O(n^2)***, a execução deles em sequência será ***O(max(n,n^2))*** que é ***O(n^2)***
    - Dois algoritmos cujos tempos de execução são ***O(n)*** e ***O(n log n)***, a execução deles em sequência será ***O(max(n,n log n))*** que é ***O(n log n)***

Diferentes tipos de análise assintótica

- Notação grande-Theta, Θ
  - Descreve o **limite assintótico firme**
  - É utilizada para analisar o **limite inferior** e **superior** do algoritmo
  - A notação $\Theta(n^2)$ nos diz que o custo do algoritmo é, assintoticamente, igual a n^2
    - Ou seja, o custo do algoritmo original é n^2 dentro de um fator constante **acima** e **abaixo**

Diferentes tipos de análise assintótica

- Notação grande-Theta, ***Θ***
  - Matematicamente, a notação ***Θ*** é assim definida
    - Uma função custo **f(n)** é ***Θ(g(n))*** se existem três constantes positivas $\mathbf{c_1}$, $\mathbf{c_2}$ e **m** tais que
    - Para $\mathbf{n \geq m}$, temos $\mathbf{c_1.g(n) \leq f(n) \leq c_2.g(n)}$

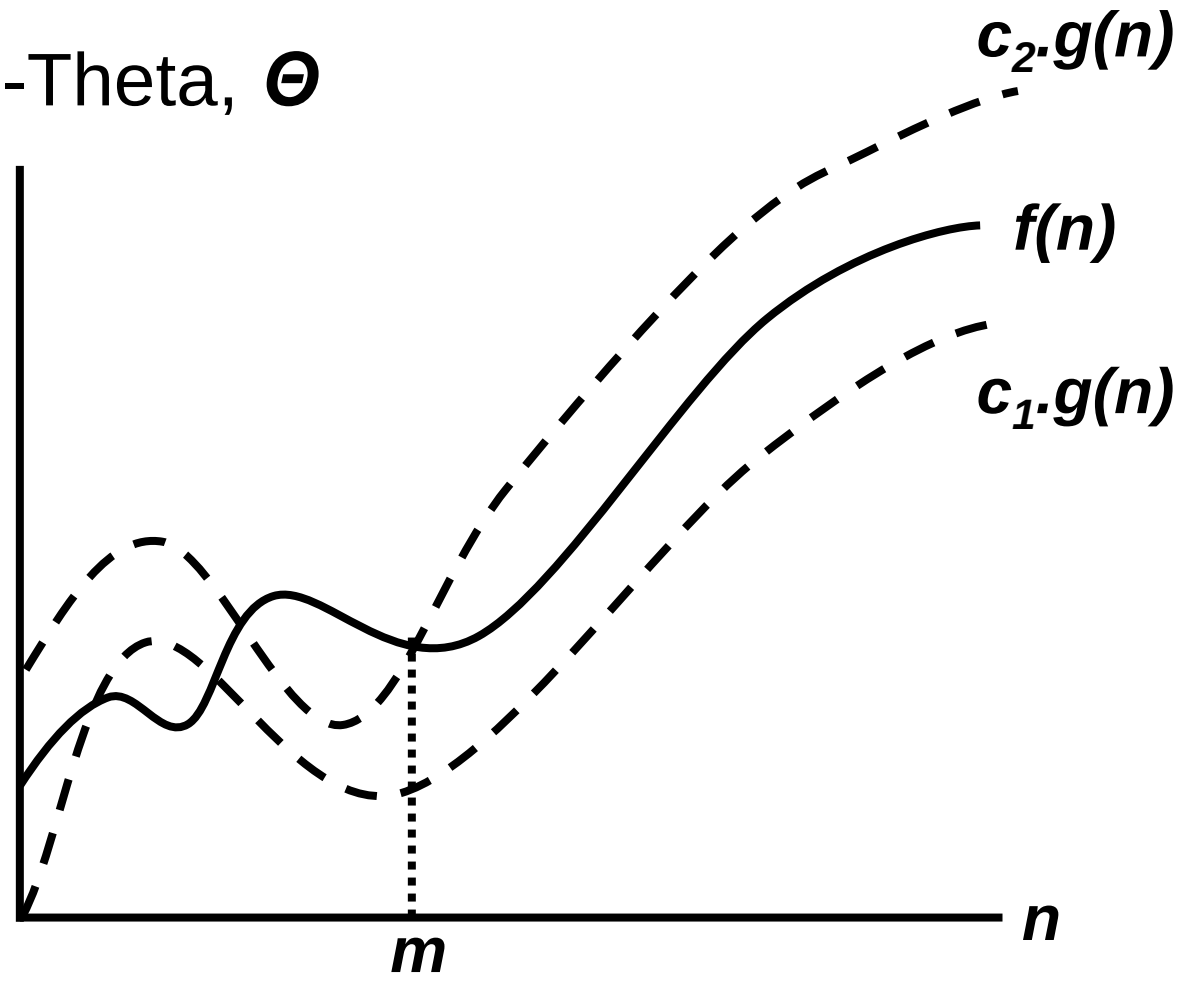
    - Confuso?

Diferentes tipos de análise assintótica

- Notação grande-Theta, ***Θ***
  - Em outras palavras, para todos os valores de **n** à direita de **m**, o resultado da função custo **f(n)** é sempre **igual** ao valor da função usada na notação ***Θ***, **g(n)**, quando está é multiplicada por constantes $\mathbf{c_1}$ e $\mathbf{c_2}$

Diferentes tipos de análise assintótica

- Notação grande-Theta, **Θ**



f(n) = Θ(g(n))

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo $f(n) = \frac{1}{2}n^2 - 3n$ é ***Θ(n²)***
  - Temos que encontrar constantes **c_1** e **c_2** e **m** tais que
  - $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$
  - Dividindo por **n^2**, temos
  - $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo $f(n) = \frac{1}{2}n^2 - 3n$ é ***Θ(n²)***
  - $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$
  - A desigualdade do lado direito é válida para **n ≥ 1** escolhendo **c_2 ≥ 1/2**
  - A desigualdade do lado esquerdo é válida para **n ≥ 7** escolhendo **c_1 ≥ 1/14**
  - Assim, para **c_1 ≥ 1/14, c_2 ≥ 1/2** e **n ≥ 7,** $f(n) = \frac{1}{2}n^2 - 3n$ é ***Θ(n²)***

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo $f(n) = 6n^3$ **não** é ***Θ(n²)***
  - Temos que encontrar constantes **c_1** e **c_2** e **m** tais que
  - $c_1 n^2 \leq 6n^3 \leq c_2 n^2$
  - Dividindo por **n^2**, temos
  - $c_1 \leq 6n \leq c_2$

Diferentes tipos de análise assintótica

- Exemplo: mostrar que a função custo $f(n) = 6n^3$ **não** é ***Θ(n²)***
  - $c_1 \leq 6n \leq c_2$
- A desigualdade do lado direito é inválida!
  - $n \leq \frac{c_2}{6}$
  - O valor de **n** está limitado pela constante $\mathbf{c_2}$
  - A análise assintótica não é possível (entrada tendendo ao infinito)

Diferentes tipos de análise assintótica

- Notação pequeno-o, ***o***, *e* pequeno-omega, **ω**
  - Parecidas com as notações **Grande-O** e **Grande-Omega**
  - As notações **Grande-O** e **Grande-Omega** possuem uma relação de **menor ou igual** e **maior ou igual**
  - As notações **Pequeno-o** e **Pequeno-omega** possuem uma relação de **menor** e **maior**

Diferentes tipos de análise assintótica

- Notação pequeno-o, ***o****, e* pequeno-omega, **ω**
  - Ou seja, essas notações não representam limites próximos da função
  - Elas representam limites estritamente
    - **superiores**: sempre maior
    - **inferiores**: sempre menor

Classes de problemas

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

$$\mathbf{O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)}$$

Classes de problemas

- Mais classes de problemas
  - O(n log n): ordem log linear
    - Típica de algoritmos que trabalham com particionamento dos dados. Esses algoritmos resolvem um problema transformando-o em problemas menores, que são resolvidos de forma independente e depois unidos
  - $O(n^2)$: 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

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Classes de problemas

- Mais classes de problemas
  - $O(n^3)$: ordem cúbica
    - É caracterizado pela presença de três estruturas de repetição aninhadas
  - $O(2^n)$: ordem exponencial
    - Geralmente ocorre quando se usa uma solução de **força bruta**. Não são úteis do ponto de vista prático
  - 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 que o exponencial

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$

Classes de problemas

- Comparação no tempo de execução
  - Computador executa 1 milhão de operações por segundo

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 50$	$n = 100$
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	5,9E-02 segundos	58,1 minutos	3855,2 séculos	2,3E+08 séculos	1,3E+13 séculos

Classes de problemas

- Cuidado
  - Na análise assintótica as constantes de multiplicação são consideradas irrelevantes e descartadas
    - Porém, elas podem ser relevantes na prática, principalmente se o tamanho da entrada é pequeno
  - Exemplo: qual função tem menor custo?
    - **$f(n) = 10^{100} * n$**
    - **$g(n) = 10n \log n$**

Classes de problemas

- Cuidado
  - Análise assintótica: o primeiro é mais eficiente
    - **f(n) = 10^{100} * n** tem complexidade **O(n)**
    - **g(n) = 10n log n** tem complexidade **O(n log n)**
  - No entanto, 10^{100} é um número muito grande
    - Neste caso, **10n log n > 10^{100} * n** apenas para
    - Para qualquer valor menor de **n** o $n > 2^{10^{99}}$ complexidade **O(n log n)** será melhor

RELAÇÕES DE RECORRÊNCIAS

Relações de Recorrências

- Função recursiva
  - Função que chama a si mesma durante a sua execução
- Exemplo: fatorial de um número **N**.
  - Para **N = 4** temos
    - 4! = 4 * 3!
    - 3! = 3 * 2!
    - 2! = 2 * 1!
    - 1! = 1 * 0!
    - 0! = 1

Relações de Recorrências

- Função recursiva
  - Matematicamente, o fatorial é definido como
    - N! = N * (N-1)!
    - 0! = 1
  - Implementação

```
int fatorial(int n){
    if (n == 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

Relações de Recorrências

- Recorrência ou Relação de Recorrência
  - Expressão que descreve uma função em termos de entradas menores da função
    - Exemplo: definição de um função recursiva
  - Muitos algoritmos se baseiam em recorrência
  - Ferramenta importante para a solução de problemas combinatórios
- Relação de recorrência do fatorial
  - $T(n) = T(n-1) + n$

Relações de Recorrências

- Complexidade da recorrência
  - Uma recursão usualmente não utiliza estruturas de repetição, apenas comandos condicionais, atribuições etc
  - Podemos erroneamente imaginar que essa funções possuem complexidade ***O(1)***

```
int fatorial(int n){
    if (n == 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

Relações de Recorrências

- Complexidade da recorrência
  - Saber a complexidade da recursão envolve resolver a sua relação de recorrência
  - T(n) = T(n-1) + n

```
int fatorial(int n){
    if (n == 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

Relações de Recorrências

- Complexidade da recorrência
  - Temos que encontrar uma **fórmula fechada** que nos dê o valor da função **T(n) = T(n-1) + n** em termos de seu parâmetro **n**
    - Geralmente obtido como uma combinação de polinômios, quocientes de polinômios, logaritmos, exponenciais etc.

```
int fatorial(int n){
    if (n == 0)
        return 1;
    else
        return n * fatorial(n-1);
}
```

Relações de Recorrências

- Considere a seguinte relação de recorrência
  - **T(n) = T(n-1) + 2n + 3**
- Para $\boldsymbol{n} \in \{\mathbf{2}, \mathbf{3}, \mathbf{4}, \ldots\}$, existem inúmeras funções **T** que satisfazem a recorrência
  - Depende do **caso base**, **T(1)**
  - Exemplos
    - **T(1) = 1**

n	1	2	3	4	5
T(n)	1	8	17	28	41

    - **T(1) = 5**

n	1	2	3	4	5
T(n)	5	12	21	32	45

Relações de Recorrências

- Problema
  - Para cada valor **i** e o intervalo $\boldsymbol{n} \in \{\mathbf{2}, \mathbf{3}, \mathbf{4}, \ldots\}$ existe uma (e apenas uma) função **T** que tem caso base **T(1) = i** e satisfaz a recorrência
  - **T(n) = T(n-1) + 2n + 3**

n	1	2	3	4	5
T(n)	1	8	17	28	41

n	1	2	3	4	5
T(n)	5	12	21	32	45

Relações de Recorrências

- Solução
  - Precisamos encontrar uma **fórmula fechada** para a recorrência
  - Podemos expandir a relação de recorrência **T(n)=T(n-1) + 2n + 3** até que se possa detectar um comportamento no seu caso geral

Relações de Recorrências

- Para entender essa técnica de expansão, considere a seguinte recorrência
  - **T(n) = T(n-1) + 3**
  - Essa relação de recorrência representa um algoritmo que possui 3 operações mais uma chamada recursiva

Relações de Recorrências

- Expandindo a recorrência **T(n) = T(n-1) + 3**
  - Se aplicarmos o termo **T(n-1)** sobre a relação **T(n)**. Com isso, obtemos
    - **T(n-1) = T(n-2) + 3**
  - Se aplicarmos o termo **T(n-2)** sobre a relação **T(n)**, teremos
    - **T(n-2) = T(n-3) + 3**

Relações de Recorrências

- Expandindo a recorrência **T(n) = T(n-1) + 3**
  - Se continuarmos esse processo, teremos a seguinte expansão
    - **T(n) = T(n-1) + 3**
    - **T(n) = (T(n-2) + 3) + 3**
    - **T(n) = ((T(n-3) + 3) + 3) + 3**
  - Perceba que a cada passo um valor 3 é somado a expansão e o valor de **n** é diminuído em uma unidade

Relações de Recorrências

- Expandindo a recorrência **T(n) = T(n-1) + 3**
  - Podemos resumir essa expansão para usando a seguinte equação
    - **T(n) = T(n-k) + 3k**
  - Resta saber quando esse processo de expansão termina
    - Isso ocorre no **caso base**

Relações de Recorrências

- Expandindo a recorrência **T(n) = T(n-1) + 3**
  - O **caso base** ocorre quando **n-k = 1** ou seja, **k=n-1**
  - Substituindo, temos
    - **T(n) = T(n-k) + 3k**
    - **T(n) = T(1) + 3(n-1)**
    - **T(n) = T(1) + 3n - 3**

Relações de Recorrências

- Expandindo a recorrência **T(n) = T(n-1) + 3**
- Obtemos **T(n) = T(1) + 3n - 3**
  - **T(1)** é o **caso base**: recursão termina
  - Logo, seu custo é constante: ***O(1)***

- Complexidade da recorrência
  - **T(n) = 3n - 3 + *O(1)***
  - Ou seja, **linear**: ***O(n)***

Relações de Recorrências

- Outro exemplo: considere a seguinte recorrência
  - **T(n) = T(n/2) + 5**
  - Essa relação de recorrência representa um algoritmo que possui 5 operações mais uma chamada recursiva que divide os dados sempre pela metade **(n/2)**

Relações de Recorrências

- Neste caso, a recorrência existe apenas para valores de **n** que representem uma potência de 2
  - $n \in \{2^1, 2^2, 2^3, \ldots\}$
- Considerando $\mathbf{n = 2^k}$, podemos reescrever a recorrência como
  - $\mathbf{T(2^k) = T(2^{k-1}) + 5}$

Relações de Recorrências

- Expandindo a recorrência $\mathbf{T(2^k) = T(2^{k-1}) + 5}$
  - Se aplicarmos o termo $\mathbf{T(2^{k-1})}$ sobre a relação $\mathbf{T(2^k)}$. Com isso, obtemos
    - $\mathbf{T(2^{k-1}) = T(2^{k-2}) + 5}$
  - Se aplicarmos o termo $\mathbf{T(2^{k-2})}$ sobre a relação $\mathbf{T(2^k)}$, teremos
    - $\mathbf{T(2^{k-2}) = T(2^{k-3}) + 5}$

Relações de Recorrências

- Expandindo a recorrência **$T(2^k) = T(2^{k-1}) + 5$**
  - Se continuarmos esse processo, teremos a seguinte expansão
    - **$T(2^k) = T(2^{k-1}) + 5$**
    - **$T(2^k) = (T(2^{k-2}) + 5) + 5$**
    - **$T(2^k) = ((T(2^{k-3}) + 5) + 5) + 5$**
  - Perceba que a cada passo um valor 5 é somado a expansão e o valor de **k** é diminuído em uma unidade

Relações de Recorrências

- Expandindo a recorrência **$T(2^k) = T(2^{k-1}) + 5$**
  - Ao final da expansão, teremos
    - **$T(2^k) = T(2^{k-k}) + 5k$**
    - **$T(2^k) = T(2^0) + 5k$**
    - **$T(2^k) = T(1) + 5k$**
  - Podemos resumir essa expansão usando a seguinte equação, a qual já considera o seu caso base
    - **$T(2^k) = T(1) + 5k$**

Relações de Recorrências

- Expandindo a recorrência $\mathbf{T(2^k) = T(2^{k-1}) + 5}$
  - Temos que substituir o custo do **caso base**, ***O(1)***
  - Complexidade da recorrência
    - $\mathbf{T(2^k) = }$ ***O(1)*** $\mathbf{+ 5k}$

  - Devemos lembrar que substituímos **n** por $\mathbf{2^k}$ no início da expansão, de modo que $\mathbf{n = 2^k}$

Relações de Recorrências

- Expandindo a recorrência $\mathbf{T(2^k) = T(2^{k-1}) + 5}$
  - Aplicando o logaritmo em $\mathbf{n = 2^k}$, temos que $\mathbf{k=\log_2 n}$
  - Substituindo, temos
    - $\mathbf{T(2^k) = }$ ***O(1)*** $\mathbf{+ 5k}$
    - $\mathbf{T(n) = }$ ***O(1)*** $\mathbf{+ 5 \log_2 n}$
- Complexidade da recorrência
  - $\mathbf{T(n) = }$ ***O(1)*** $\mathbf{+ 5 \log_2 n}$
  - Ou seja, **logarítmica**: $\boldsymbol{O(\log_2 n)}$

Material Complementar | Vídeo Aulas

- Aula 99: Análise de Algoritmos:
  - youtu.be/iZK5WwJFIPE
- Aula 100: Análise de Algoritmos – Contando Instruções:
  - youtu.be/wflNJurvTTQ
- Aula 101: Análise de Algoritmos – Comportamento Assintótico:
  - youtu.be/SClFMUpBiaw
- Aula 102: Análise de Algoritmos – Notação Grande-O:
  - youtu.be/Q7nwypDgTS8
- Aula 103: Análise de Algoritmos – Tipos de Análise Assintótica:
  - youtu.be/9RgC2dxi4W8
- Aula 104: Análise de Algoritmos – Classes de Problemas:
  - youtu.be/8RYvWMOMnXw
- Aula 122 – Relações de Recorrência:
  - youtu.be/QeLYRyW5T94

Material Complementar | GitHub

- https://github.com/arbackes