INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Campos do Jordão

ESTRUTURA DE DADOS

Análise de Algoritmos e Complexidade

Análise Assintótica, Método Analítico, Análise de Recorrências, Teorema Mestre.

Professor Mestre Igor de Moraes Sampaio
igor.sampaio@ifsp.edu.br

Instituto Federal de Educação, Ciência e Tecnologia de São Paulo - **Campus Campos do Jordão**

Relembrando

Onde estamos?

- **Complexidade de Tempo**
  - Mede a quantidade de operações executadas pelo algoritmo conforme o tamanho da entrada cresce.
- **Complexidade de Espaço**
  - Mede a quantidade de memória extra necessária para armazenar variáveis, pilhas de recursão e estruturas auxiliares.
- **Análise teórica (matemática)**
  - Modela o algoritmo como uma sequência de passos e conta o número de operações em função do tamanho da entrada n.
- **Análise empírica (experimental)**
  - Executa o algoritmo em diferentes tamanhos de entrada e mede o tempo e o uso de memória na prática.

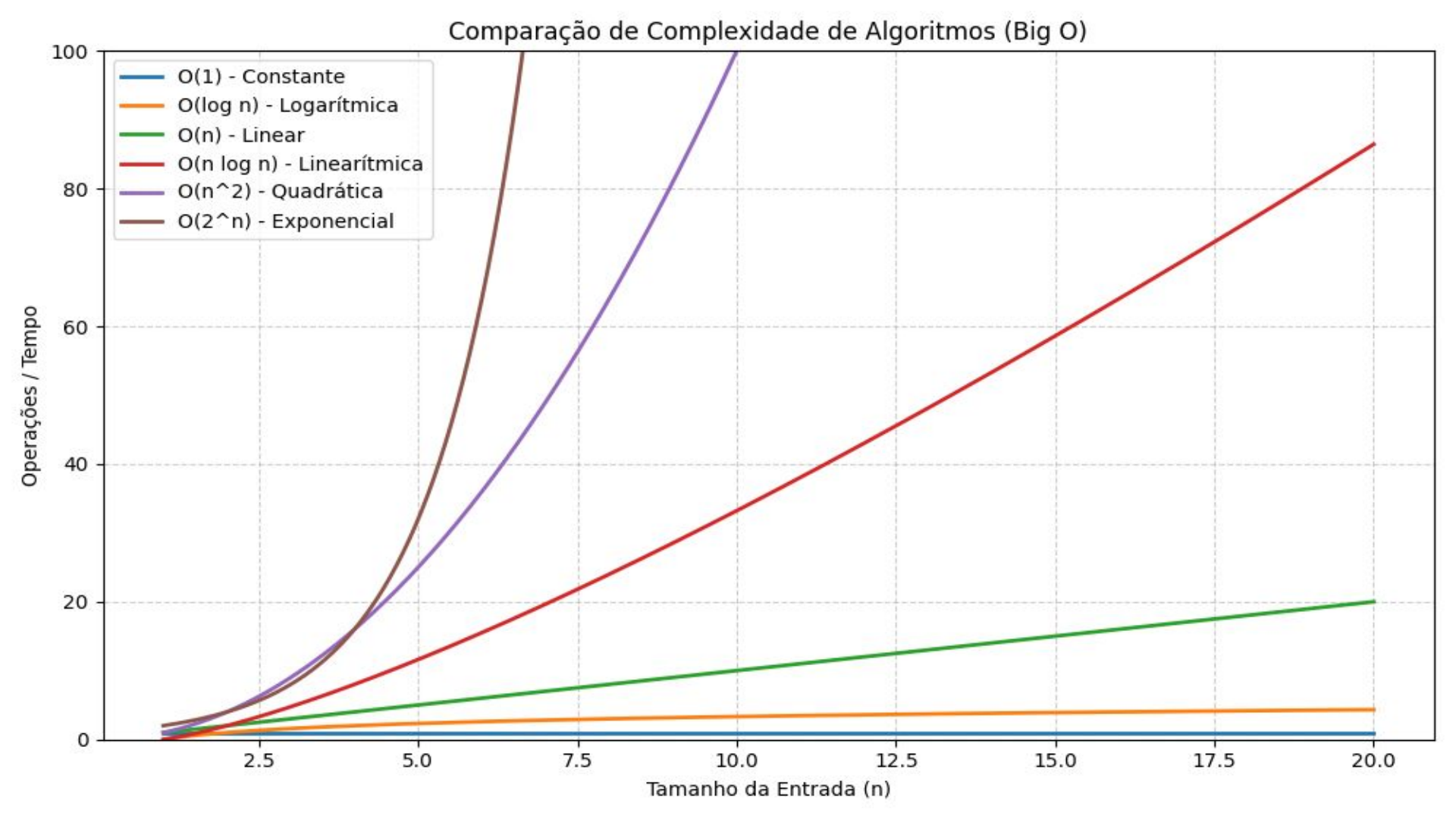
Onde estamos?

Notação	Indica	Representa
O(g(n))	Limite superior	Pior caso
Ω(g(n))	Limite inferior	Melhor caso
Θ(g(n))	Limite exato	Crescimento igual nos casos
o(g(n))	Crescimento menor	Estritamente mais rápido
ω(g(n))	Crescimento maior	Estritamente mais lento

Complexidade de Tempo

- Quantifica o número de operações básicas realizadas pelo algoritmo.

Notação	Nome	Exemplo típico
O(1)	Constante	Acesso direto em array
O(log n)	Logarítmica	Busca binária
O(n)	Linear	Pesquisa sequencial
O(n log n)	Linearítmica	Merge Sort, Quick Sort (médio caso)
$O(n^2)$	Quadrática	Bubble Sort, Insertion Sort
$O(2^n)$	Exponencial	Algoritmos de força bruta em conjuntos
O(n!)	Fatorial	Permutações completas


Comparação de Complexidade de Algoritmos (Big O)
O(1) - Constante
O(log n) - Logarítmica
O(n) - Linear
O(n log n) - Linearítmica
O(n^2) - Quadrática
O(2^n) - Exponencial
Operações / Tempo
Tamanho da Entrada (n)
0
20
40
60
80
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

Análise Assintótica

A notação Θ

Vamos definir formalmente o que significa a notação Theta . Se encontrarmos c1, c2 e n0 que satisfaçam a equação, temos que f(n) é Θ(g(n)). c1, c2 e n0 maiores que 0.

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n), \forall n \geq n_0$$

Em um resumo bem simplista essa equação está dizendo que se a gente “imprensar” f(n) com g(n) multiplicada por duas constantes diferentes, dizemos que f(n) é Θ(g(n)).

.

A notação Θ

$g(n) = n$

$f(n) = 3n + 1$

$c1 = 1, c2 = 6$

$$0 <= 1 * n <= 3n + 1 <= 6n$$

$n = 1$

$$0 <= 1 * 1 <= 3 * 1 + 1 <= 6 * 1$$

$$0 <= 1 <= 4 <= 6$$

A notação Θ

$$3 * n + 1 = \Theta(n)$$

- Formalmente, dizemos que g(n) é um limite assintótico restrito para f(n).
- f(n) é limitada inferiormente e superiormente por g(n) = n.
    - Na verdade, todas as funções lineares são.

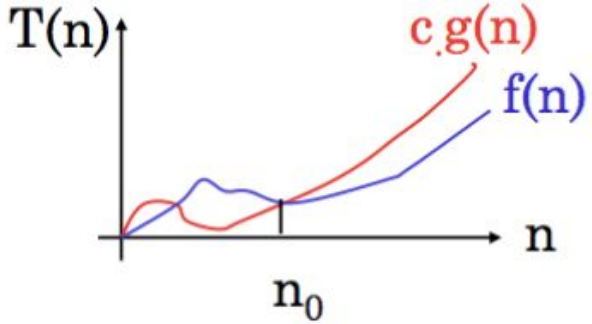
Análise Assintótica

- **Queremos também ser capazes de dizer:**
  - Big O: g(n) é um limite superior para f(n).
  - Ω: g(n) é um limite inferior para g(n).
- **E ainda tem mais...**
  - o: g(n) é um limite superior (não incluso) para f(n).
  - ω: g(n) é um limite inferior (não incluso) para g(n).

A notação Big O

Se encontrarmos c e n0 que satisfaçam a equação, temos que f(n) é O(g(n)).

$$0 \leq f(n) \leq c * g(n), \forall n \geq n_0$$

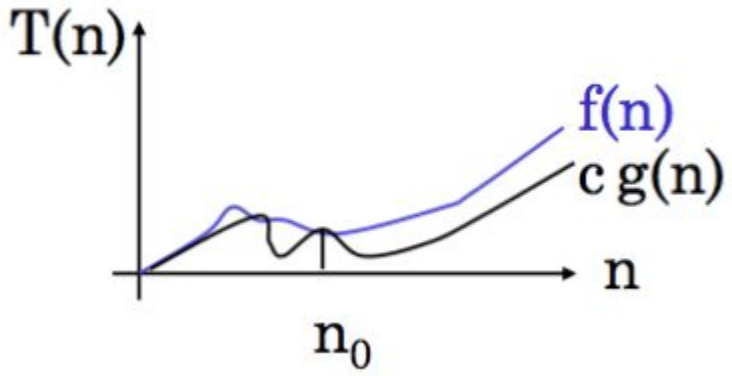


Note que g(n) é apenas o limite superior.

A notação Omega (Ω)

A notação Omega (Ω) define apenas o limite inferior. Para duas funções f(n) e g(n), dizemos que f(n) é Ω(g(n)) se:

$$0 \leq c * g(n) \leq f(n), \forall n \geq n_0$$



Desafio

Desafios

1 - Prove formalmente que a função $f(n) = 3n + 1$ pertence à classe $\Theta(n)$.

2 - Dada a função $f(n) = 2n^2 + 10$, prove que ela é $O(n^2)$.

3 - Compare as funções abaixo e determine qual domina assintoticamente a outra.

$f(n) = 4n + 10$
$g(n) = 5n^2 - 10000$

Método Analítico

Método Analítico

O **Método Analítico** na análise de algoritmos é uma abordagem baseada em **fórmulas matemáticas e técnicas formais** para determinar o comportamento do tempo de execução e o uso de recursos de um algoritmo.

Ele é utilizado para derivar a **complexidade assintótica** do algoritmo, fornecendo uma estimativa do seu desempenho conforme o tamanho da entrada cresce.

É uma abordagem formal para derivar uma função matemática **f(n)** que descreve o custo do algoritmo somando o custo de suas operações primitivas.

Operações Primitivas

- No método analítico não contamos segundos, contamos primitivas.

- **Operações primitivas:** São operações básicas e indivisíveis que um algoritmo executa (como somar, comparar, acessar posição de vetor). Elas são o "átomo" do custo computacional.

- Cada operação primitiva tem custo constante, ou seja, O(1).

Operações Primitivas

Operação Primitiva	Exemplo em C++	Custo
Atribuição	x = 5;	O(1)
Acesso a vetor	v[i]	O(1)
Comparação	a < b	O(1)
Soma	a + b	O(1)
Subtração	a - b	O(1)
Multiplicação	a * b	O(1)
Divisão	a / b	O(1)
Incremento	i++	O(1)
Decremento	i--	O(1)
Leitura de variável	x	O(1)
Escrita em vetor	v[i] = x;	O(1)

Método Analítico

- Nesse contexto, o tempo de execução de um algoritmo é a soma do custo das operações primitivas.

- Por exemplo, considere o algoritmo a seguir:

```
int calc(int n) {
    int a = n + 5;   // Linha 1
    int b = a * 10;  // Linha 2
    return b;        // Linha 3
}
```

Método Analítico

- **Linha 1:**
  - atribuição: a = -> c1
  - soma: n + 5 -> c2
- **Linha 2:**
  - atribuição: b = -> c3
  - multiplicação: a * 10 -> c4
- **Linha 3:**
  - retorno: return b -> c5
- **Equação Final:**
  - f(n) = c1 + c2 + c3 + c4 + c5
  - f(n) = 5c

```
int calc(int n) {
    int a = n + 5;    // Linha 1
    int b = a * 10;   // Linha 2
    return b;         // Linha 3
}
```

Método Analítico

- Lembrando estamos interessados em uma função que nos diga o tempo de execução em relação ao tamanho da entrada. Nesse caso, escolhemos n para representar o tamanho da entrada.

- Como pode ser visto na função detalhada, o custo não depende de de maneira alguma. Independente dos números passados como parâmetro, o custo será sempre o mesmo.

- Por isso dizemos que essa função, e portanto o algoritmo que é descrito por ela, tem custo constante, ou seja, independe do tamanho da entrada.

- Não importa se n é 1 ou 1 milhão, o número de operações nunca muda.

Método Analítico

- Dizer que um algoritmo tem custo constante significa dizer que o seu tempo de execução independe do tamanho da entrada.

- Outro fator de destaque é que podemos considerar que todas as constantes possuem o mesmo valor c. Assim, podemos simplificar a função para:

- f(n) = 5c -> f(n) = 5 * 1 -> f(n) = 5

- Ou seja, a notação assintótica do algoritmo é O(1).

Método Analítico (Passo a Passo)

- Como no exemplo o método analítico pode ser realizado seguindo os seguintes passo:
    - **Passo 1:** Identificar todas as linhas que contêm operações primitivas e atribuir uma constante de custo c1, c2, ... para cada linha.
    - **Passo 2:** Determinar quantas vezes cada linha é executada em função do tamanho da entrada n.
    - **Passo 3:** Somar tudo para encontrar f(n) e simplificar para a notação Big O.

Método Analítico - Condicionais

O uso de comandos condicionais é muito comum em nossos algoritmos e nos impõe uma dificuldade na análise do tempo de execução.

Essa dificuldade está relacionada ao fato de que, dependendo do caso, apenas uma parte do código é executada.

Nesse caso, escolhemos o **pior caso**. Neste contexto estamos interessados em saber como os algoritmos se comportam no seu pior caso. Essa análise nos dá uma visão muito clara sobre o que posso esperar da execução de um algoritmo.

Exemplo - Condicionais

Para demonstrar a análise de pior caso, vamos analisar o código a seguir:

```
double precisaNaFinal(double nota1, double nota2, double nota3) {
    double mediaAtual = (nota1 + nota2 + nota3) / 3.0;

    if (mediaAtual >= 7 || mediaAtual < 4)
        return 0.0;
    else {
        double mediaFinal = 5.0;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double notaFinalNecessaria = (mediaFinal - (mediaAtual * pesoMedia)) / pesoFinal;
        return notaFinalNecessaria;
    }
}
```

Exemplo - Condicionais

```
double precisaNaFinal(double nota1, double nota2, double nota3) {
    double mediaAtual = (nota1 + nota2 + nota3) / 3.0;

    if (mediaAtual >= 7 || mediaAtual < 4)
        return 0.0;
    else {
        double mediaFinal = 5.0;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double notaFinalNecessaria = (mediaFinal - (mediaAtual * pesoMedia)) / pesoFinal;
        return notaFinalNecessaria;
    }
}
```

Passo 1:
Identificar primitivas.

Passo 2:
Identificar a quantidade de vezes que cada uma das primitivas é executada

Passo 3:
Somar o custo total.

Passo 1: Identificar primitivas.

```
double precisaNaFinal(double nota1, double nota2, double nota3) {
    // atribuição: mediaAtual = -> c1
    // soma: nota1 + nota2 + nota3 -> c2
    // soma: c2 + nota3 -> c3
    // divisão: c3 / 3.0 -> c4
    double mediaAtual = (nota1 + nota2 + nota3) / 3.0;

    // comparação: mediaAtual >= 7 -> c5
    // comparação: mediaAtual < 4 -> c6
    if (mediaAtual >= 7 || mediaAtual < 4)
        // retorno: 0.0 -> c7
        return 0.0;
    else {
        // atribuição: mediaFinal = 5.0 -> c8
        double mediaFinal = 5.0;
        // atribuição: pesoFinal = 0.4 -> c9
        double pesoFinal = 0.4;
        // atribuição: pesoMedia = 0.6 -> c10
        double pesoMedia = 0.6;
        // multiplicação: mediaAtual * pesoMedia -> c11
        // subtração: mediaFinal - c11 -> c12
        // divisão: c12 / pesoFinal -> c13
        double notaFinalNecessaria = (mediaFinal - (mediaAtual * pesoMedia)) / pesoFinal;
        // retorno: notaFinalNecessaria -> c14
        return notaFinalNecessaria;
    }
}
```

Passo 2: Quantas de vezes cada primitiva é executada.

- Aqui vem a grande diferença. Como estamos interessados no pior caso, nós vamos descartar a constante c7, pois, no pior caso, o bloco do else será executado, uma vez que é mais custoso que o bloco do if.

- As outras primitivas são executadas apenas uma vez.

Passo 3: Somar o custo total.

```
double precisaNaFinal(double nota1, double nota2, double nota3) {
    double mediaAtual = (nota1 + nota2 + nota3) / 3.0;

    if (mediaAtual >= 7 || mediaAtual < 4)
        return 0.0;
    else {
        double mediaFinal = 5.0;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double notaFinalNecessaria = (mediaFinal - (mediaAtual * pesoMedia)) / pesoFinal;
        return notaFinalNecessaria;
    }
}
```

- Note que c7 é desconsiderada.

f(n) = c1 + c2 + c3 + c4 + c5 + c6 + c8 + c9 + c10 + c11 + c12 + c13 + c14 -> **f(n) = 13c**

Método Analítico - Iteração

Nos dois exemplos que vimos até aqui todas as primitivas são executadas apenas uma vez e, por isso, o tempo de execução do algoritmo é sempre constante.

Vejamos o que acontece quando há iteração.

Exemplo - Loop

```
bool contem(int vetor[], int tamanho, int valor) {
    for (int i = 0; i < tamanho; i++) {
        if (vetor[i] == valor)
            return true;
    }
    return false;
}
```

Passo 1:
Identificar primitivas.

Passo 2:
Identificar a quantidade de vezes que cada uma das primitivas é executada

Passo 3:
Somar o custo total.

Passo 1: Identificar primitivas.

```
bool contem(int vetor[], int tamanho, int valor) {
    // atribuição: i = 0 -> c1
    // comparação: i < tamanho -> c2
    // incremento: i++ -> c3
    for (int i = 0; i < tamanho; i++) {
        // comparação: vetor[i] == valor -> c4
        if (vetor[i] == valor)
            // retorno: true -> c5
            return true;
    }
    // retorno: false -> c6
    return false;
}
```

Observação Importante

Se lançarmos um olhar mais detalhista em algumas expressões, na verdade, vamos perceber que estamos passando por cima de algumas primitivas.

Por exemplo, nesse exemplo nós consideramos que a expressão booleana v[i] == n é uma primitiva (c4), mas ela envolve também o acesso à v[i] e o acesso a n que, como sabemos, são também primitivas. Então, sendo bem detalhistas, teríamos que identificar 3 primitivas na expressão v[i] == n. Da mesma forma, a expressão i + j pode ser considerada como sendo 3 primitivas, isto é, o acesso à variável i, o acesso à variável j e a expressão aritmética.

Eu escolhi não fazer isso por fins didáticos. Iríamos poluir muito nossa análise. Por isso, quando houver uma expressão booleana, mesmo que ela envolva outras primitivas, vamos considerar como apenas uma. O mesmo será feito para expressões aritméticas.

Passo 2: Quantas de vezes cada primitiva é executada.

- Aqui mora a grande diferença da análise deste exemplo em relação aos demais. Em primeiro lugar, nem todas as primitivas são executadas apenas uma vez.

- Depois, temos que voltar a lembrar que estamos tratando do pior caso. Esse cenário é representado por um array que não contém o número procurado, pois o algoritmo irá realizar todas as iterações e retornar false no final.

- Veja que se o número procurado estiver presente, a execução pode terminar bem antes do fim da iteração no array. Isso significa que na nossa análise vamos descartar a primitiva c5, pois no pior caso ela nunca é executada.

Passo 2: Quantas de vezes cada primitiva é executada.

```
bool contem(int vetor[], int tamanho, int valor) {
    // atribuição: i = 0 -> c1 é executada apenas uma vez
    // comparação: i < tamanho -> c2 é executada (n+1) vezes
    // Exemplo: para tamanho = 5, c2 é executada 6 vezes
    // 0 < 5 (true), 1 < 5 (true), 2 < 5 (true), 3 < 5 (true), 4 < 5 (true), 5 < 5 (false)
    // incremento: i++ -> c3 é executada n vezes
    // Exemplo: para tamanho = 5, c3 é executada 5 vezes
    // incremento de i: 0 -> 1, 1 -> 2, 2 -> 3, 3 -> 4, 4 -> 5
    for (int i = 0; i < tamanho; i++) {
        // comparação: vetor[i] == valor -> c4 é executada n vezes
        if (vetor[i] == valor)
            // retorno: true -> c5 no pior caso não é executado
            return true;
    }
    // retorno: false -> c6 é executado apenas uma vez no pior caso
    return false;
}
```

Passo 3: Somar o custo total.

- O tempo de execução do algoritmo é a soma das execuções das operações primitivas. Nesse caso temos que a função que descreve o tempo de execução é:

$$f(n) = c1 + c2 * (n + 1) + c3 * n + c4 * n + c6$$

- Considerando todas as primitivas com custo c e simplificando a função, temos:

$$f(n) = 3cn + 3c$$

Método Analítico - Loop

Veja que essa função é diretamente relacionada ao tamanho do array (n). À medida que cresce o tamanho de n, cresce também o tempo de execução do pior caso.

Esse crescimento é linear, pois a função é linear. Faz sentido, certo? Iterar em um array com 100 posições é 10 vezes mais lento que iterar em um array de 10 posições. Não é por acaso que o nome desse algoritmo é busca linear.

Desafio

Vamos juntos usar o método analitico para gerar a função que descreve o tempo de execução do seguinte algoritmo:

```
void contarLog(int n){
    for (int i = 1; i < n; i = i * 2) {
        cout << i << " ";
    }
}
```

Passo 1: Identificar primitivas.

```
void contarLog(int n){
    for (int i = 1; i < n; i = i * 2) {
        cout << i << " ";
    }
}
```

Passo 1: Identificar primitivas.

```
void contarLog(int n){
    // atribuição: i = 1 -> c1
    // comparação: i < n -> c2
    // incremento: i = i * 2 -> c3
    for (int i = 1; i < n; i = i * 2) {
        // operação de saída: cout -> c4
        cout << i << " ";
    }
}
```

Passo 2: Quantas de vezes cada primitiva é executada.

```
void contarLog(int n){
    // atribuição: i = 1 -> c1 é executada apenas uma vez
    // comparação: i < n -> c2 é executada log2(n) + 1 vezes
    // incremento: i = i * 2 -> c3 é executada log2(n) vezes
    for (int i = 1; i < n; i = i * 2) {
        // operação de saída: cout -> c4 é executada log2(n) vezes
        cout << i << " ";
    }
}
```

Passo 3: Somar o custo total.

```
void contarLog(int n){
    // atribuição: i = 1 -> c1 é executada apenas uma vez
    // comparação: i < n -> c2 é executada log2(n) + 1 vezes
    // incremento: i = i * 2 -> c3 é executada log2(n) vezes
    for (int i = 1; i < n; i = i * 2) {
        // operação de saída: cout -> c4 é executada log2(n) vezes
        cout << i << " ";
    }
}
```

$$f(n) = c1 + c2 * (\log2(n) + 1) + c3 * \log2(n) + c4 * \log2(n)$$

$$f(n) = 3*c*\log2(n) + 2c \quad \rightarrow \quad f(n) = 3*1*\log2(n) + 2*1$$

$$f(n) = 3\log2(n) + 2 \quad \rightarrow \quad O(\log n)$$

Desafio

Desafio

Use o método analitico para gerar a função que descreve o tempo de execução dos algoritmos a seguir:

Desafio

```
bool mostraMatrizDinamica(int** matriz, int linhas, int colunas) {
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            cout << matriz[i][j] << " ";
        }
        cout << endl;
    }
    return true;
}
```

Desafio

```
bool contemDuplicacao(int vetor[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        for (int j = i + 1; j < tamanho; j++) {
            if (vetor[i] == vetor[j])
                return true;
        }
    }
    return false;
}
```

Desafio

```
bool buscaBinaria(int vetor[], int tamanho, int valor) {
    int esquerda = 0;
    int direita = tamanho - 1;

    while (esquerda <= direita) {
        int meio = esquerda + (direita - esquerda) / 2;

        if (vetor[meio] == valor)
            return true;

        if (vetor[meio] < valor)
            esquerda = meio + 1;
        else
            direita = meio - 1;
    }
    return false;
}
```

Análise de Recorrências

Recorrência

- **Quando um algoritmo contém uma chamadas recursivas**, seu tempo de execução pode **frequentemente ser descrito por uma recorrência.**

- Para os algoritmos recursivos, **a ferramenta principal desta análise não é uma somatória**, mas um tipo especial de equação chamada **relação de recorrência.**

- Uma recorrência é uma equação ou desigualdade que **descreve uma função em termos de seu valor em entradas menores.**

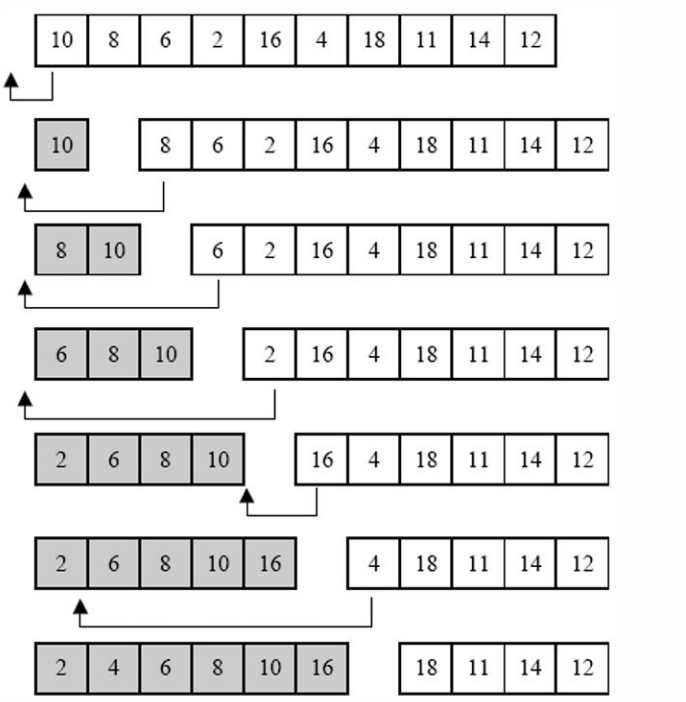
Recorrência

- **Para cada procedimento recursivo é associada uma função de complexidade T(n)** desconhecida, onde n mede o tamanho dos argumentos para o procedimento.

- **Equação de recorrência:** maneira de definir uma função por uma expressão envolvendo a mesma função.

Recorrência

- Vamos considerar o algoritmo de ordenação por Inserção.



Recorrência

- Considerando o pior caso:
  - Na primeira vez, apenas uma operação é necessária ...
  - Da segunda, 2 ...
  - Da terceira, 3 ... E isso é executado ... N vezes ...
  - Ou seja, todos os elementos serão inseridos na última posição verificada.

Recorrência

- Considerando o pior caso:
  - Na primeira vez, apenas uma operação é necessária ...
  - Da segunda, 2 ...
  - Da terceira, 3 ... E isso é executado ... N vezes ...

$$T(n) = T(n-1) + 1$$
$$T(n) = T(n-2) + 2$$
$$T(n) = T(n-3) + 3$$
$$T(n) = T(n-4) + 4$$
$$\dots$$
$$T(2) = T(1) + n-2$$
$$T(1) = n-1$$

Recorrência

- O caso anterior ao caso base é a ordenação de um vetor de duas posições, onde é efetuada apenas uma comparação.

- Então...

$$T(n) = 1 + 2 + 3 + 4 + \ldots + (n - 2) + (n - 1)$$

Recorrência

$$T(n) = 1 + 2 + 3 + 4 + \ldots + (n-2) + (n-1)$$

$$T(n) = \sum_{i=1}^{n-1} i = \left(\sum_{i=1}^{n} i \right) - n = \left(\frac{n(n+1)}{2} \right) - n$$

$$T(n) = \frac{n^2 + n}{2} - n = \frac{n^2 + n - 2n}{2}$$

$$T(n) = \frac{n^2 - n}{2}$$

$$T(n) = \frac{n^2 - n}{2} \in \Theta(n^2)$$

Recorrência

- Outro exemplo de recorrência:

```
int somaVetor(int vetor[], int n) {
    if (n <= 0)
        return 0;
    return vetor[n - 1] + somaVetor(vetor, n - 1);
}
```

Recorrência

- Montando a equação de recorrência:

```
int somaVetor(int vetor[], int n) {
    // caso base da recursão é O(1)
    if (n <= 0)
        return 0;
    // chamada recursiva é O(n), pois é feita n vezes
    // Exemplo: para n = 4
    // somaVetor(vetor, 4) = vetor[3] + somaVetor(vetor, 3)
    // somaVetor(vetor, 3) = vetor[2] + somaVetor(vetor, 2)
    // somaVetor(vetor, 2) = vetor[1] + somaVetor(vetor, 1)
    // somaVetor(vetor, 1) = vetor[0] + somaVetor(vetor, 0)
    return vetor[n - 1] + somaVetor(vetor, n - 1);
}
```

Recorrência

- Montando a equação de recorrência:

```
int somaVetor(int vetor[], int n) {
    // PARTE A: O Caso Base (Critério de Parada)
    if (n <= 0)
        return 0; // Custo constante (c1)

    // PARTE B: Trabalho Local + Chamada Recursiva
    return vetor[n - 1] + somaVetor(vetor, n - 1);
    //     |___________|  |_____________________|
    //   Trabalho Local      Chamada Recursiva
    //   (Soma, Acesso)      Tamanho reduzido
    //     Custo (c2)           T(n-1)
}
```

Recorrência

- Montando a equação de recorrência:
- Passo a Passo:
  - Identificando o Caso Base ($n=0$):
    - O algoritmo faz apenas uma comparação e um retorno.
    - Equação: T(0) = c1 (onde c1 é uma constante de tempo).
  - Identificando o Passo Recursivo (n > 0):
    - O algoritmo gasta um tempo para somar e acessar o array (c2).
    - E gasta um tempo T(n-1) para calcular o restante.
    - Equação: T(n) = T(n-1) + c_2.

Recorrência

- Resultado Final (A Recorrência):

$$T(n) = \begin{cases} c_1 & \text{se } n = 0 \\ T(n-1) + c_2 & \text{se } n > 0 \end{cases}$$

- Ler-se: O tempo para somar n elementos é igual ao tempo para somar n-1 elementos mais o tempo de uma adição.

Desafio

E se o código fosse assim? O que muda na equação?

```
return somaVetor(vetor, n - 1) + somaVetor(vetor, n - 1);
```

Recorrência

- Exemplo 2: Divisão e Conquista (Divide pela metade)
  - Exemplo Clássico: Merge Sort (Ordenação por Intercalação)

```
void mergeSort(int v[], int inicio, int fim) {
    if (inicio < fim) {
        int meio = (inicio + fim) / 2;

        mergeSort(v, inicio, meio);
        mergeSort(v, meio + 1, fim);

        merge(v, inicio, meio, fim);
    }
}
```

Recorrência

- Montando a equação de recorrência:

```
void mergeSort(int v[], int inicio, int fim) {
    if (inicio < fim) {                  // Custo constante (c1)
        int meio = (inicio + fim) / 2; // Custo constante (c2)

        // Chamadas Recursivas
        mergeSort(v, inicio, meio);    // T(n/2)
        mergeSort(v, meio + 1, fim);   // T(n/2)

        // Trabalho Local Pesado
        merge(v, inicio, meio, fim);   // Intercalação: Custo Theta(n)
    }
}
```

Recorrência

- **Caso Base:** Se o vetor tem tamanho 1 (inicio == fim), não faz nada. Custo constante.
  - T(1) = O(1)

- Chamadas Recursivas:
  - Quantas vezes chamamos a função? **2 vezes.**
  - Qual o tamanho do problema? O vetor é dividido ao meio. Logo, n/2.
  - Termo: 2 * T(n/2)

- Trabalho Local (O Segredo):
  - Calcular o meio e o if são baratos (c).
  - Mas a função merge() (intercalar) percorre o vetor todo para ordenar. O custo dela é proporcional a n.
  - Custo local total: O(n) (ou c * n)

Recorrência

- Resultado Final (A Recorrência):

$$T(n) = \begin{cases} c_1 & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

A Anatomia da Recorrência

- A "fórmula geral" que cobre 90% dos casos de Divisão e Conquista:

$$T(n) = a * T(n/b) + f(n)$$

- **a**: O número de chamadas recursivas dentro do código (quantos filhos a árvore tem).
- **n/b**: O quanto o problema diminui (se divide ao meio, b = 2; se divide em 3 partes, b = 3).
- **f(n)**: O custo de dividir o problema + o custo de combinar os resultados (o código que sobra fora da recursão).

Desafio

Desafio

Encontre a equação de recorrência de:

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

Desafio

Encontre a equação de recorrência de:

```
int buscaBinaria(int v[], int meio, int chave) {
    if (v[meio] == chave) return meio;

    if (chave < v[meio])
        return buscaBinaria(v, meio / 2, chave);
    else
        return buscaBinaria(v + meio + 1, meio / 2, chave);
}
```

Resolvendo Equações de Recorrência

Formas de Resolver Recorrências

- **Método da Substituição (ou Iterativo):**
  - O que é: "Abrir" a equação repetidamente até encontrar um padrão matemático (série aritmética ou geométrica).
  - Melhor para: Recorrências lineares simples (T(n) = T(n-1) + c).
- **Método da Árvore de Recorrência:**
  - O que é: Desenhar a estrutura das chamadas recursivas e somar o custo de cada nível da árvore.
  - Melhor para: Entender "visualizar" o custo e palpitar a resposta antes de provar.
- **Teorema Mestre (Master Theorem):**
  - O que é: Uma "receita de bolo" (fórmula pronta) para casos de Divisão e Conquista.
  - Melhor para: Algoritmos clássicos como Merge Sort, Busca Binária (T(n) = aT(n/b) + f(n)).