INTRODUÇÃO A ANÁLISE

DE ALGORITMOS

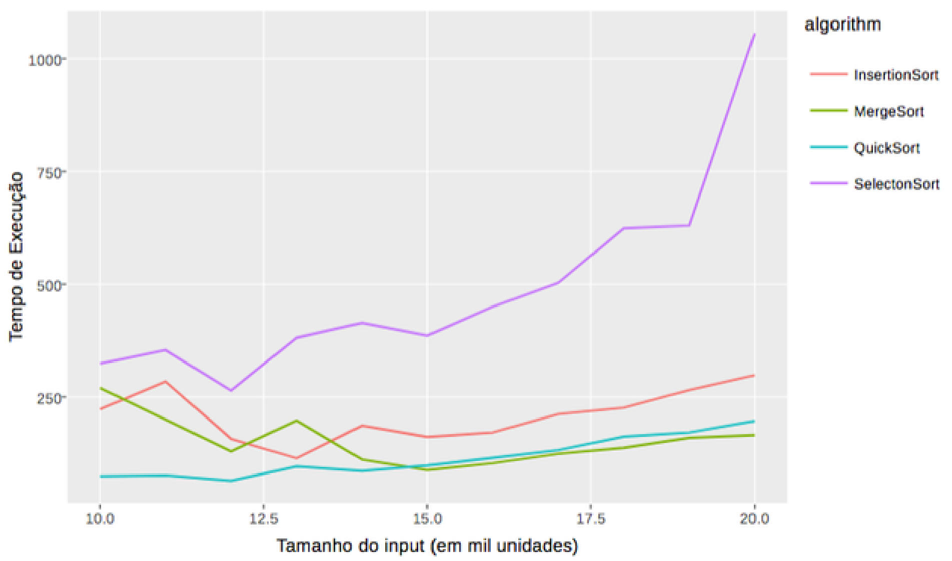
O que é análise de algoritmos?

É prever a quantidade de recursos que tal algoritmo consome ao ser executado
Para medir o tempo de execução, podemos usar as seguintes abordagens:

Abordagem empírica

Também conhecido como método experimental, é uma abordagem direta para analisar o desempenho de um algoritmo, nesse caso, o ambiente é configurado com:

1. Variáveis controladas
2. Executa-se o algoritmo para medir o tempo de execução
3. O eixo y contém o tempo de execução
4. O eixo x contém o tamanho da entrada



Entretanto, um dos problemas desse método é que as conclusões são limitadas às entradas fornecidas ao experimento. Por isso, para uma análise que seja independente de hardware, com uma análise de entradas maiores e que seja simples, surge então a **análise assintótica**

Análise assintótica

Mas antes de irmos para a Análise assintótica(também conhecida como método analítico), vamos aderir o custo primitivo, em que **" O custo de operações primitivas é constante"**. Esse tempo constante é 1, e pode ser representado como O(1) o O(c). Segue abaixo algumas operações primitivas:

- Avaliação de expressões booleanas (i >= 2; i == 2, etc)
- Operações matemáticas (*, - ,+ , % , etc)
- Retorno de métodos (return x)
- Atribuição (i = 2)
- Acesso a variáveis e posição arbitrárias de um array (v[i])

```
int multiplicaRestoPorPartesInteiras(int i, int j){
    int resto = i % j;
    int pInteira = i / j;
    int resultado = resto * pInteira;
    return resultado;
}
```

1º Passo: Identifique as operações primitivas

1. Atribuição (resto =) --> c1
2. Operação aritmética (i % J) --> c2
3. Atribuição (pInteira =) --> c3
4. Operação aritmética (i / j) --> c4
5. Atribuição (resultado =) --> c5
6. Operação aritmética (resultado * pInteira) --> c6
7. Retorno de método (return resultado) --> c7

2º Passo: Identificar a quantidade de vezes que cada uma das primitivas são executadas, neste caso, são executadas apenas uma vez

3ºPasso: Somar o custo total:

$$f(n) = c1 + c2 + c3 + c4 + c5 + c6 + c7$$

$$f(n) = 7c$$

Existem algumas funções, que independente do tamanho de sua entrada, o custo será sempre o mesmo, como é o caso do exemplo anterior, já que não é levado em consideração o número da entrada em si, mas sim a quantidade de passos de execução e seu custo primitivo, então, mesmo que eu passasse como parâmetro da função os números 1 e 2, ou 400 e 300, o custo de execução seria o mesmo. Logo, funções com esse comportamento possuem **custo constante**.

E quando houver condicionais?

Nesse caso, escolhemos o **pior caso,** a análise de pior caso é útil para eliminar soluções ruins, vamos analisar o código abaixo que possui condicionais:

```
double precisaNaFinal(double nota1, double nota2, double nota3) {

    double media = (nota1 + nota2 + nota3) / 3;

    if (media >= 7 || media < 4) {
        return 0;

    } else {
        double mediaFinal = 5;
        double pesoFinal = 0.4;
        double pesoMedia = 0.6;
        double precisa = (mediaFinal - pesoMedia * media) / pesoFinal;

        return precisa;
    }

}
```

Para retirar o melhor caso, simplesmente desconsideramos a constante que retorna o melhor caso **geralmente** o return true

1º Passo: Identifique as operações primitivas

1. Atribuição (media =) --> c1
2. Operação aritmética (nota1 + nota2 + nota3) --> c2
3. Operação aritmética (/2) --> c3
4. Avalia uma expressão booleana (media >= 7 | | media < 4) --> c4
5. Retorno de método (return 0) --> c5
6. Atribuição (mediaFinal =) --> c6
7. Atribuição (pesoFinal =) --> c7
8. Atribuição (pesoMedia =) --> c8
9. Atribuição (precisa =) --> c9
10. Operação aritmética (mediaFinal - pesoMedio) --> c10
11. Operação aritmética (... * media) --> c11
12. Operação aritmética (... / pesoFinal) --> c12
13. Retorno de método (precisa) --> c13

2º Passo: Identificar a quantidade de vezes que cada uma das primitivas são executadas, observe que quando estamos fazendo a análise de um código que tem condicionais queremos sempre fazer essa análise do pior caso, logo, observe que o melhor caso é o if, onde avaliamos a expressão booleana (c4), se acontecer de cair no melhor caso o método irá retornar 0, mas não queremos que caia no melhor caso, por isso, vamos descartar a c5 (onde ocorre o retorno e é menos custoso), e as outras primitivas serão executadas apenas uma vez.

3ºPasso: Somar o custo total:

f(n) = c1 + c2 + c3 + c4 + c5 + c6 + c7 + c8 + c9 + c10 + c11 + c12 + c13

f(n) = 12c

E quando houver iterações?

Lembrando de algumas operações primitivas temos que v[i] == n é uma primitiva.
de acesso a variáveis e posição arbitrárias de um array (v[i]), dessa forma, deveriam ser considerados 2 operações primitivas, porém, por fins didáticos, vamos assumir apenas uma primitiva, vamos analisar o código abaixo:

```java
public static boolean contains(int[] v, int n) {
    for (int i = 0; i < v.length; i++)
        if (v[i] == n)
            return true;
    return false;
}
```

1º Passo: Identifique as operações primitivas

1. Atribuição (int i = 0) --> c1
2. Avaliação de expressão booleana (i < v.length) --> c2
3. Operação arimética (i++) --> c3
4. Avaliação de uma expressão booleana (v[i] == n) --> c4
5. Retorno de método (return true) --> c5
6. Retorno de método (return false) --> c6

2º Passo: Identificar a quantidade de vezes que cada uma das primitivas são executadas, primeiro, vamos levar em consideração que nem todas as primitivas são executadas apenas uma vez. Em seguida, estamos sempre procurando analisar o pior caso, nessa situação o pior caso é quando o número não está no array, então ele percorre todo o array para retornar o false, assim, como visto anteriormente vamos tirar o melhor caso (quando o número está no array.) Por isso vamos descartar o melhor caso, quando ele retorna true, nesse caso, a primitiva c5. Agora vamos analisar a quantidade de vezes que a primitiva vai aparecer, aderindo que o tamanho do array, v.length é = n

- c1 --> int i = 0;
  Será executado apenas uma vez
- c2 --> i < v.length
  É uma expressão booleana que vai ser verificada n + 1 vez, isso porque ele começa verificando se 0 < v.length, então ele verifica:
  0 < v.length ------- 1 vez
  1 < v.length ------- 2 vezes
  2 < v.length ------- 3 vezes
  3 < v.length ------- 4 vezes
  4 < v.length ------- 5 vezes
  5 < v.length ------- 6 vezes

Assim, o tamanho do array é 5, mas a verificação acontece n + 1 = 5 + 1

- c3 --> i ++
  Como o i já começa com 0, eu não preciso contar o 0, por isso a quantidade de vezes é igual a n, pois os incrementos começam quando o i = 0, logo i vira 1, 2, 3, 4, 5
- c4
  n vezes
- c5 não é executado

- c6

  Uma vez

3º Passo: Somar o custo total, agora a função começou a mudar

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

E então, se lembrarmos bem, cada passo pode ser c que equivale a 1, simplificando, temos então:

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

E de onde vem isso? Vamos destrinchar

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

Temos c3 * n

c4 * n

c2 * n , assim, temos 3 * (c *n)

e sobra c1 + 1 + c6, como 1 pode ser escrito como c

eu tenho c1 + c + c6, que é igual a 3*c, logo:

3 * (c * n) + 3 * c

E quando houver loops alinhados

```java
public boolean contemDuplicacao(int[] v) {
    for (int i = 0; i < v.length; i++)
        for (int j = i + 1; j < v.length; j++)
            if (v[i] == v[j])
                return true;
    return false;
}
```

1º Passo: Identifique as operações primitivas

1. Atribuição (int i = 0) --> c1
2. Avaliação de expressão booleana (i < v.length) --> c2
3. Expressão arimética (i ++) --> c3
4. Atribuição (int j = ...) --> c4
5. Expressão arimética (= i + 1) --> c5
6. Avaliação de expressão booleana (j < v.length) --> c6
7. Expressão arimética (j ++) --> c7
8. Avaliação de expressão booleana (v[i] == v[j]) --> c8
9. Retorno de método (return true) --> c9
10. Retorno de método (return false) --> c10

2º Passo: Identificar a quantidade de vezes que cada uma das primitivas são executadas:

O pior caso do array é quando não há repetições de valores, então os loops são executados até o final, por isso descartamos a c9, porque ela é o melhor caso, onde o algoritmo retorna true, ou seja, existe uma repetição. Considerando que o tamanho do vetor é n.

- c1 é executada 1 vez
- c2 é executada n + 1 vezes
- c3 é executada n vezes
- c4 é executada n vezes
- c5 é executada n vezes
- A quantidade de execuções de c6 depende do laço mais externo, pois j varia de acordo com i (j=i+1). Como o laço externo executa n vezes, a quantidade de vezes que j varia é dada por: n+(n−1)+(n−2)+(n−3)+(n−4)+...1. Essa série representa uma Progressão Aritmética finita decrescente com razão 1. A soma de uma PA com essas características é dada por S=n/2∗(a1+an) onde a1 e an são o primeiro e o último elemento da sequência, respectivamente. Assim, para a1=1 e an=n, temos que c6 é executada (n**2+n)/2 vezes.
- Como c7 é executada uma vez a menos que c6, então temos que o primeiro termo da PA é a1=1 e an=n−1. Assim, temos que c7 é executada n**2/2
- c8 é executada a mesma quantidade de vezes que c7.
- c9 não é executada nenhuma vez porque estamos falando do pior caso.
- c10 é executada apenas uma vez.

3ºPasso: Somar o custo total:

f(n) = c1 + c2 * (n +1) + c3 * n + c4 * n + c5 * n + c6 * (n**2 + n)/2 + c7 * n **2/2 +c8 * n **2 / 2 + c10