Estruturas de Dados

Análise de Algoritmos

Prof. Igor de Moraes Sampaio
igor.sampaio@ifsp.edu.br

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

Análise de eficiência

Em diversas situações, o interesse está em comparar algoritmos, ao invés de determinar o seu tempo exato de execução. Em particular, estamos interessados nas funções no comportamento dos algoritmos para grandes tamanhos de entrada – análise assintótica.

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.

Operações Primitivas

- O custo de execução das operações primitivas é constante $\mathcal{O}(1)$
  - Avaliação de expressões booleanas (i >= 2; i == 2, etc);
  - Operações matemáticas (*, -, +, %, etc);
  - Retorno de métodos (return x;);
  - Atribuição (i = 2);
  - Acesso à variáveis e posições arbitrárias de um array (v[i]).

Método Analítico

- Nesse contexto, o tempo de execução de um algoritmo é a soma do custo das operações primitivas. Por exemplo, considere o algoritmo que multiplica o resto da divisão de dois inteiros pela parte inteira da mesma divisão:

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

Passo 1: Identificar primitivas.

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

- atribuição (resto =) -> c1
- operação aritmética (i % j) -> c2
- atribuição (pInteira =) -> c3
- operação aritmética (i / j) -> c4
- atribuição (resultado =) -> c5
- operação aritmética (resto * pInteira) -> c6
- retorno de método (return resultado) -> c7

Passo 2: Quantas de vezes cada primitiva é executada.

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

- Identificar a quantidade de vezes que cada uma das primitivas é executada. Para o algoritmo acima, todas as primitivas são executadas apenas uma vez.

Passo 3: Somar o custo total.

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

- 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:

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7$$

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 n 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.

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:

$$f(n) = 7c \rightarrow f(n) = 7 * 1 \rightarrow f(n) = 7$$

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

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 um método que recebe as três notas de um aluno e calcula a nota que ele precisa obter na prova final, se esse for o caso. Se o aluno for aprovado (media >= 7.0) ou reprovado sem direito a final (media < 4), o método deve retornar 0.

```
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;
    }
}
```

Exemplo - 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;
    }
}
```

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.

- atribuição (media =) -> c1
- operação aritmética (nota1 + nota2 + nota3) -> c2
- operação aritmética (... / 3) -> c3
- avaliação de expressão booleana (media >=7 || media < 4) -> c4
- retorno de método (return 0) -> c5
- atribuição (mediaFinal =) -> c6
- atribuição (pesoFinal =) -> c7
- atribuição (pesoMedia =) -> c8
- atribuição (precisa =) -> c9
- operação aritmética (pesoMedia * media) -> c10
- operação aritmética (mediaFinal - ...) -> c11
- operação aritmética (... / pesoFinal) -> c12
- retorno de método (return precisa) -> c13

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 c5, 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 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;
    }
}
```

- Note que c_5 é desconsiderada.

$$f(n) = c_1 + c_2 + c_3 + c_4 + c_6 + c_7 + c_8 + c_9 + c_{10} + c_{11} + c_{12} + c_{13}$$

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 contains(int v[], int size, int n) {
    for (int i = 0; i < size; i++) {
        if (v[i] == n) {
            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.

- Atribuição (int i = 0) -> c1
- Avaliação de expressão booleana (i < v.length) -> c2
- Operação aritmética (i++) -> c3
- Avaliação de expressão booleana (v[i] == n) -> c4
- Retorno de método (return true) -> c5
- Retorno de método (return false) -> c6

```
bool contains(int v[], int size, int n) {
    for (int i = 0; i < size; i++) {
        if (v[i] == n) {
            return true;
        }
    }
    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, ok? 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.

- Dado que o tamanho do vetor (v.length) é n, temos:
    - c1 é executada apenas uma vez.
    - c2 é executada (n+1) vezes. Exemplo: se n = 5, temos as seguintes verificações: 0 < 5, 1 < 5; 2 < 5, 3 < 5, 4 < 5 e 5 < 5, quando encerra-se o loop. Ou seja, 6 verificações.
    - c3 é executada n vezes. Exemplo: se n=5, temos os seguintes incrementos em i: 1, 2, 3, 4 e 5, quando encerra-se o loop.
    - c4 é executada n vezes.
    - No pior caso, c5 não é executada.
    - c6 é executada apenas uma vez.

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:

$$f(n) = c_1 + c_2 * (n + 1) + c_3 * n + c_4 * n + c_6$$

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

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

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. O termo refere-se a ambos:

- a estratégia de procurar o elemento de modo sequencial em uma coleção e
- o tempo de execução do algoritmo.

Desafio

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

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

Conclusão

- Uma forma de calcular o tempo de execução é seguir os passos:
  - Identificar primitivas
  - Identificar o número de vezes que cada uma das primitivas é executada
  - Somar o custo total
- É preciso estar atento para o fato de que estamos falando de análise do pior caso e, por isso, descartamos os fluxos alternativos de menor custo.