Análise de Complexidade

Pior caso, melhor caso, caso médio

Prof. Edson Alves - UnB/FGA

2020

Sumário

Pior caso, melhor caso, caso médio

Definição

- A análise de algoritmos considera três cenários possíveis, os quais relacionam a entrada com o número de iterações do algoritmo
- O pior caso acontece quando o algoritmo executa o número máximo de iterações possível
- No melhor caso o algoritmo executa o número mínimo de iterações possível
- O caso médio representa o cenário esperado quando as entradas possuem determinada distribuição de probabilidade de ocorrência
- Em termos formais, a complexidade do caso médio C_M é dada por

$$C_M = \sum_i p(\textsf{input}_i)\textsf{steps}(\textsf{input}_i)$$

  com $p(\textsf{input}_i) \geq 0$ e $\sum_i p(\textsf{input}_i) = 1$
- A fórmula para o caso médio coincide com a definição probabilística de valor esperado

Observações

- O melhor caso tem interesse meramente teórico, não sendo levado em consideração na maior parte das análises
- A maioria das análises se concentram no pior caso, pois ele é uma estimava de como o algoritmo efetivamente vai se comportar
- Embora o caso médio seja mais próximo da realidade, sua análise é mais técnica e depende de conceitos elaborados de matemática e probabilidade
- Além disso, o caso médio tende a ser idêntico ao pior caso no contexto da complexidade assintótica
- A notação mais utilizada é a notação Big-O, seguida pela notação Big-Θ

Exemplos de análise de complexidade assintótica

Exemplo 1

Problema: Implementar uma função que torne maíscula a primeira letra da string dada como parâmetro

Solução:

```
void capitalize(string& s)
{
    s[0] = toupper(s[0]);
}
```

Complexidade: A implementação da função faz uma única atribuição, de modo que, se a string tem n caracteres, $f(n) = 1, \forall n$. Assim, o algoritmo tem complexidade $O(1)$.

Exemplo 2

Problema: Implementar uma função que retorne a string dada, com a primeira letra em maiúsculo

Solução:

```
1 string capitalize(const string& s)
2 {
3     auto r = s;
4     r[0] = toupper(r[0]);
5
6     return r;
7 }
```

Complexidade: Observe que os n caracteres da string s devem ser copiados em r, além da atribuição feita na linha 4. Assim, $f(n) = n + 1$, de modo que o algoritmo tem complexidade $O(n)$.

Exemplo 3

Problema: Implementar uma função que retorne o produto cartesiano dos conjuntos de números inteiros A e B.

Solução:

```
vector<pair<int, int>> prod(const vector<int>& A, const vector<int>& B)
{
    vector<pair<int, int>> P;
    int n = A.size(), m = B.size();

    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
                P.push_back(make_pair(A[i], B[j]));

    return P;
}
```

Exemplo 3

Complexidade: Na linha 4 são feitas duas atribuições (variáveis n e m, respectivamente). No início do laço externo (linha 6) é feita mais uma atribuição (`i = 0`).

A cada iteração do laço externo, são feitas 2 atribuições (`j = 0` e o incremento `++i`, respectivamente). Como o laço externo é executado n vezes, são mais $2n$ atribuições.

Em cada iteração do laço interno são feitas mais duas atribuições: o incremento `++j` e a adição do pair (i, j) ao vetor P por meio do método push_back(). Como este método tem complexidade constante, ele equivale a c atribuições. Este laço será iniciado n vezes, e em cada execução são m iterações. Assim, serão mais

$$\sum_{i=0}^{n}\sum_{j=0}^{m}(c+1) = (c+1)nm$$

atribuições, de modo que $f(n, m) = 3 + 2n + (c+1)nm$ é $O(nm)$.

Exemplo 4

Problema: Implementar uma função que retorne todos os pares de naturais (a, b) tais que $a < b \leq N$, para um inteiro positivo N dado.

Solução:

```
vector<pair<int, int>> pairs(int N)
{
    vector<pair<int, int>> ps;

    for (int a = 1; a <= N; ++a)
        for (int b = i + 1; b <= N; ++b)
            ps.push_back(make_pair(a, b));

    return ps;
}
```

Exemplo 4

Complexidade: No início do laço externo (linha 5) é feita uma atribuição. Este laço tem N iterações, e em cada uma delas são feitas mais duas atribuições: o incremento ++a e a inicialização `b = i + 1`.

A cada execução o laço interno itera ($N - i$ vezes), sendo que em cada iteração são duas atribuições: o incremento ++b e a inserção do par no vetor por meio do método `push_back()` (para fins de simplicação, considere que o custo deste método seja o mesmo de uma atribuição). Assim, serão

$$\sum_{i=1}^{N} 2(N-i) = 2\sum_{k=1}^{N-1} k = 2\left[\frac{N(N-1)}{2}\right] = N(N-1)$$

atribuições. Portanto,

$$f(N) = 1 + 2N + N(N-1),$$

de modo que o algoritmo tem complexidade $O(N^2)$.

Exemplo 5

Problema: Implemente uma função que compute o produto da matriz $A_{n\times m}$ pela matriz $B_{m\times p}$.

Solução:

```cpp
Matrix operator*(const Matrix& A, const Matrix& B)
{
    int n = A.rows(), m = A.cols(), p = B.cols();
    Matrix C(n, p);

    for (int i = 0; i < n; ++i)
        for (int j = 0; i < p; ++j)
            for (int k = 0; k < m; ++k)
                C[i][j] += (A[i][k] * B[k][j]);

    return C;
}
```

Exemplo 5

No cálculo da complexidade assintótica de um algoritmo não é necessário determinar a função $f(n)$ exatamente, e sim determinar o termo dominante.

A cada execução, o laço externo, o laço intermediário e o laço interno iteram n, p e m vezes, respectivamente. Observe que, em cada execução de um destes laços, o número de atribuições associadas a ele é constante.

Assim, o algoritmo terá complexidade $O(nmp)$. A título de curiosidade, a função $f(n, m, p)$ é dada por

$$f(n, m, p) = 4 + 2n + 3np + 2nmp$$

Exemplo 6

Problema: Computar um vetor de inteiros v tal que $v_i = 1$, se i é primo, e $v_i = 0$, caso contrário, para $i \leq N$.

Algoritmo:

```cpp
vector<int> sieve(int N)
{
    vector<int> v(N + 1, 1);
    v[0] = v[1] = 0;

    for (int i = 2; i <= N; ++i)
        if (v[i])
            for (int j = 2*i; j <= N; j += i)
                v[j] = 0;

    return v;
}
```

Exemplo 6

Complexidade: Na linha 3 é alocado um vetor com $N + 1$ posições, e cada uma destas posições é inicializada com o valor 1, de modo que são $N + 1$ atribuições. A linha seguinte faz mais duas atribuições, uma vez que 0 e 1 não são primos.

O laço externo inicia com uma atribuição (`i = 2`) e itera $N - 1$ vezes, incrementando a variável `i` (`++i`) em cada uma destas iterações. Já a inicialização da variável `j` (`j = 2*i`) depende do valor de `v[i]`: só acontecerá quando `v[i] = 1`, ou seja, quando `i` for primo.

Para terminar $f(N)$ precisamente seria necessário determinar o número exato de primos no intervalo $[1, N]$ (função $\pi(N)$). Existem boas aproximações para esta quantidade (por exemplo, $\pi(N) \approx N/\log N$), porém quanto mais precisa a aproximação, mas sofisticada é a função.

Exemplo 6

Importante lembrar que a notação Big-O fornece uma cota superior. Deste modo, o número de execuções do laço interno pode ser majorado, pois $\pi(N) < N$ (pois nem todos os números do intervalo são primos).

A cada execução do laço interno itera $\lfloor N/i \rfloor$ vezes, fazendo duas atribuições (o incremento `j += i` e a atribuição `v[j] = 0`) em cada iteração. Assim, o número total de atribuições associadas ao laço interno é

$$
\begin{aligned}
\sum_{p \text{ primo}}^{N} 2\left\lfloor \frac{N}{p} \right\rfloor \leq \sum_{i=1}^{N} 2\left\lfloor \frac{N}{i} \right\rfloor \leq 2\sum_{i=1}^{N} \frac{N}{i} \\
\leq 2N\sum_{i=1}^{N} \frac{1}{i} \leq 2N\sum_{i=1}^{\infty} \frac{1}{i} \\
< 2N\int_{i=1}^{\infty} \frac{1}{i} dN = 2N \log N
\end{aligned}
$$

Exemplo 6

A sequência de aproximações utilizada forneceu uma cota superior para o número de atribuições desejado. Embora efetivamente ele seja menor do que a aproximação, ao menos há a garantia de que o número total de atribuições feitas pelo laço interno seja sempre menor do que a cota estabelecida.

Assim, a função $f(n)$ pode ser aproximada por

$$f(n) = 3 + (N+1) + 2(N-1) + 2N\log N,$$

de modo que o algoritmo é $O(N\log N)$.

Exemplo 7

Problema: Localizar o índice de uma ocorrência do valor x no vetor ordenado v, ou -1, caso x não esteja presente em v.

Algoritmo:

```
int binary_search(int x, const vector<int>& v) {
    int N = v.size(), a = 0, b = N - 1;

    while (a <= b) {
        auto m = a + (b - a)/2;

        if (x < v[m])
            b = m - 1;
        else if (x > v[m])
            a = m + 1;
        else
            return m;
    }

    return -1;
}
```

Exemplo 7

Complexidade: Observe que, neste algoritmo, o número total de atribuições depende dos valores da entrada.

No melhor caso, o elemento x se encontra no índice $\lfloor (N-1)/2 \rfloor$, de modo que o algoritmo realiza apenas 4 atribuições:

```
N = v.size(), a = 0, b = N - 1, m = a + (b - a)/2
```

Assim, $f(N) = 3$ e o algoritmo tem complexidade $O(1)$.

No pior caso, x não está no vetor. Além das 3 atribuições iniciais, o laço itera k vezes, onde $N/2^k \leq 1$, realizando duas atribuições (m e uma dentre as variáveis a e b) a cada iteração. Logo $f(N) = 3 + 2\log N$ e o algoritmo tem complexidade $O(\log n)$.

Referências

1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. **KNUTH**, Donald. *The Art of Computer Programming - Volume 1: Fundamental Algorithms*, Addison-Wesley, 1968.