

Universidade Federal de Ouro Preto
PCC104 - Projeto e Análise de Algoritmos
Lista 1

Prof. Rodrigo Silva

March 22, 2023

1. Why do we study algorithms?

- You have to know a standard set of important algorithms from different areas of computing;
- You should be able to design new algorithms and analyze their efficiency.

There are several other reasons why we study algorithms:

- Efficiency: We study algorithms to find efficient solutions to problems. An efficient algorithm is one that can solve a problem with minimal time and space complexity.
- Correctness: We study algorithms to ensure that they are correct and produce the desired output for all possible inputs. A correct algorithm should always produce the correct output for any given input.
- Abstraction: We study algorithms to understand how to abstract a problem into a set of instructions that can be implemented on a computer.
- Reusability: We study algorithms to develop reusable solutions that can be used across different applications and domains.
- Optimization: We study algorithms to optimize existing solutions and develop new solutions that can solve problems in a more efficient and effective manner.

In summary, studying algorithms helps us:

- Understand how to solve problems efficiently and correctly,
- Abstract a problems into a set of instructions,
- Develop reusable solutions,
- Optimize existing solutions.

2. An algorithm is a step-by-step procedure for solving a problem or accomplishing a task. It is a finite set of instructions that take some input, perform a series of computations, and produce an output.

More strictly, an algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

```
3. def gcd(a, b):  
  2     while b != 0:  
  3         t = b  
  4         b = a mod b  
  5         a = t  
  6     return a
```

We can prove that the algorithm above stops by showing that the value of b decreases with each iteration of the while loop, and that eventually b will become zero, causing the loop to terminate.

The algorithm works by finding the greatest common divisor (GCD) of two numbers, a and b . In each iteration of the loop, the algorithm calculates the remainder of a divided by b , and assigns it to the variable b . Since the remainder is always less than b , the value of b decreases with each iteration of the loop.

Eventually, the algorithm will reach a point where the remainder of a divided by b is zero, meaning that b is a divisor of a . At this point, the loop will terminate, and the value of a , which has been progressively replaced with b during the iterations of the loop, will be the GCD of the original values of a and b .

Therefore, we can conclude that the algorithm always terminates and returns a result.

4. _____

5. (a) Understand the problem

- Identify what kind problem it is
- Solve small instances by hand
- Think about special cases

(b) Define the expected capabilities of the computational device

- Is the amount of memory a concern?
- Is the processing capability a concern?

(c) Define what kind of solution you need.

- Exact: May be too slow
- Approximate: Fast but may be inaccurate

(d) Select the algorithm design technique and data structures

(e) Prove correctness

(f) Analyze the algorithm

- Time efficiency
- Space efficiency
- Simplicity - Easier to understand, easier to debug, less likely to lead to bugs.
- Generality

(g) Code

- Programming Language
- Efficient implementation

6. _____

7. _____

8. _____

9. Time complexity refers to the amount of time it takes for an algorithm to run as a function of the input size. Space complexity refers to the amount of memory or storage required by an algorithm as a function of the input size.

10. The best case refers to the scenario in which an algorithm performs in the most efficient manner possible, usually when the input is already sorted or some other advantageous condition is present. For example, the best case for a sorting algorithm might occur when the input is already sorted, in which case the algorithm could complete in $O(n)$ time complexity.

The worst case refers to the scenario in which an algorithm performs in the least efficient manner possible, usually when the input is in a state that is most challenging for the algorithm to handle.

For example, the worst case for a sorting algorithm might occur when the input is in reverse order, in which case the algorithm could take $O(n^2)$ time complexity.

The average case refers to the scenario in which an algorithm performs in a typical manner, taking into account a distribution of possible inputs. This is often more difficult to define than the best and worst cases, as it depends on the specific distribution of inputs that the algorithm will encounter in practice. For example, the average case for a sorting algorithm might be when the input is randomly ordered, in which case the algorithm might take $O(n \log n)$ time complexity on average.

```

1 def sequential_search(lst, element):
2     """
3     Search for the given element in the list using sequential search.
4     """
5     for i in range(len(lst)):
6         if lst[i] == element:
7             return i
8     return -1

```

11. _____

12. _____

13. _____

14. (a) Big O

Formally, let $f(n)$ and $g(n)$ be two functions defined on the positive integers. We say that $f(n)$ is $O(g(n))$ if there exist positive constants c and n_0 such that for all $n \geq n_0$, we have:

$$f(n) \leq cg(n)$$

In other words, $f(n)$ is asymptotically bounded above by $g(n)$ up to a constant factor. This is usually written as:

$$f(n) = O(g(n))$$

Intuitively, this means that as the input size n grows larger, the rate of growth of $f(n)$ is no faster than the rate of growth of $g(n)$, up to a constant factor.

It does not mean that $f(n)$ is equal to $g(n)$, or that $f(n)$ is the worst-case running time of an algorithm that solves a problem of size n , but rather that $f(n)$ is "no worse than" $g(n)$ in terms of its growth rate.

(b) Big Ω

Big Omega notation (also called lower bound notation) is used to describe the asymptotic lower bound of a function as its input size approaches infinity. It is another way to analyze the performance of algorithms and to compare them in terms of their efficiency.

Formally, let $f(n)$ and $g(n)$ be two functions defined on the positive integers. We say that $f(n)$ is $\Omega(g(n))$ if there exist positive constants c and n_0 such that for all $n \geq n_0$, we have:

$$f(n) \geq cg(n)$$

In other words, $f(n)$ is asymptotically bounded below by $g(n)$ up to a constant factor. This is usually written as:

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

Intuitively, this means that as the input size n grows larger, the rate of growth of $f(n)$ is no slower than the rate of growth of $g(n)$, up to a constant factor.

It does not mean that $f(n)$ is equal to $g(n)$, or that $f(n)$ is the best-case running time of an algorithm that solves a problem of size n , but rather that $f(n)$ is "no better than" $g(n)$ in terms of its growth rate.

(c) Big Θ

Big Theta notation (also called tight bound notation) is used to describe the asymptotic behavior of a function when both its upper and lower bounds are the same. It is a way to precisely characterize the running time or space complexity of an algorithm.

Formally, let $f(n)$ and $g(n)$ be two functions defined on the positive integers. We say that $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1 , c_2 , and n_0 such that for all $n \geq n_0$, we have:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

In other words, $f(n)$ is asymptotically bounded both above and below by $g(n)$ up to constant factors. This is usually written as:

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

Intuitively, this means that as the input size n grows larger, the rate of growth of $f(n)$ is the same as the rate of growth of $g(n)$, up to constant factors. It implies that $f(n)$ is both an upper and lower bound for the growth rate of $g(n)$. This is a precise way to describe the running time or space complexity of an algorithm, as it captures both the best and worst-case behavior.

15. —————

$$\lim_{n \rightarrow \infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \frac{n^2 - n}{n^2} = \frac{1}{2} \lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}.$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n \rightarrow \infty} \frac{(\log_2 e) \frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2 \log_2 e \lim_{n \rightarrow \infty} \frac{1}{\sqrt{n}} = 0.$$

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

$$\lim_{n \rightarrow \infty} \frac{\log_2 n}{n} = \lim_{n \rightarrow \infty} \frac{(\log_2 n)'}{(n)'} = \lim_{n \rightarrow \infty} \frac{\frac{1}{n} \log_2 e}{1} = \log_2 e \lim_{n \rightarrow \infty} \frac{1}{n} = 0.$$

$$\lim_{n \rightarrow \infty} \frac{n}{n \log_2 n} = \lim_{n \rightarrow \infty} \frac{1}{\log_2 n} = 0.$$

16. —————

$$\lim_{n \rightarrow \infty} \frac{n \log_2 n}{n^2} = \lim_{n \rightarrow \infty} \frac{\log_2 n}{n} = (\text{see the first limit of this exercise}) = 0.$$

17. —————

18. See equation below:

$$\begin{aligned} \text{a. } \lim_{n \rightarrow \infty} \frac{p(n)}{n^k} &= \lim_{n \rightarrow \infty} \frac{a_k n^k + a_{k-1} n^{k-1} + \dots + a_0}{n^k} = \lim_{n \rightarrow \infty} \left(a_k + \frac{a_{k-1}}{n} + \dots + \frac{a_0}{n^k} \right) \\ &= a_k > 0. \end{aligned}$$

Hence $p(n) \in \Theta(n^k)$.

19. See equation below:

$$\lim_{n \rightarrow \infty} \frac{a_1^n}{a_2^n} = \lim_{n \rightarrow \infty} \left(\frac{a_1}{a_2} \right)^n = \begin{cases} 0 & \text{if } a_1 < a_2 & \Leftrightarrow & a_1^n \in o(a_2^n) \\ 1 & \text{if } a_1 = a_2 & \Leftrightarrow & a_1^n \in \Theta(a_2^n) \\ \infty & \text{if } a_1 > a_2 & \Leftrightarrow & a_2^n \in o(a_1^n) \end{cases}$$