Universidade Federal de Ouro Preto
Lecture Notes
Introduction to the Analysis of Algorithms

Prof. Rodrigo Silva

March 28, 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 ohter 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):
    while b != 0:
        t = b
        b = a mod b
        a = t
    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 scpecial 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 innacurate

   (d) Select the algorithm design techinique and data structures

   (e) Prove corectness

   (f) Analyze the algorithm
      - Time efficiency
      - Space effiency
      - 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 $O(nlogn)$ time complexity on average.

```python
def sequential_search(lst, element):
    """
    Search for the given element in the list using sequential search.
    """
    for i in range(len(lst)):
        if lst[i] == element:
            return i
    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 $n0$ such that for all $n \geq n0$, 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 $n0$ such that for all $n \geq n0$, 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 $c1$, $c2$, and $n0$ such that for all $n \geq n0$, 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. ————-

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

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

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

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

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

16. ————

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

17. ————

18. See equation below:

a. $\lim_{n\to\infty}\frac{p(n)}{n^k} = \lim_{n\to\infty}\frac{a_k n^k + a_{k-1}n^{k-1} + \ldots + a_0}{n^k} = \lim_{n\to\infty}(a_k + \frac{a_{k-1}}{n} + \ldots + \frac{a_0}{n^k})$
$= a_k > 0.$

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

19. See equation below:

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

20. ——

4. a. Computes $S(n) = \sum_{i=1}^{n} i^2$.

b. Multiplication (or, if multiplication and addition are assumed to take the same amount of time, either of the two).

c. $C(n) = \sum_{i=1}^{n} 1 = n$.

d. $C(n) = n \in \Theta(n)$. Since the number of bits $b = \lfloor \log_2 n \rfloor + 1 \approx \log_2 n$ and hence $n \approx 2^b$, $C(n) \approx 2^b \in \Theta(2^b)$.

e. Use the formula $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ to compute the sum in $\Theta(1)$ time (which assumes that the time of arithmetic operations stay constant irrespective of the size of the operations' operands).

6. a. The algorithm returns "true" if its input matrix is symmetric and "false" if it is not.

b. Comparison of two matrix elements.

c. $C_{worst}(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1)$
$= \sum_{i=0}^{n-2}(n-1-i) = (n-1)+(n-2)+\ldots+1 = \frac{(n-1)n}{2}$.

d. Quadratic: $C_{worst}(n) \in \Theta(n^2)$ (or $C(n) \in O(n^2)$).

e. The algorithm is optimal because any algorithm that solves this problem must, in the worst case, compare $(n-1)n/2$ elements in the upper-triangular part of the matrix with their symmetric counterparts in the lower-triangular part, which is all this algorithm does.

21. ——

a. $x(n) = x(n-1) + 5 \quad \text{for } n > 1, \quad x(1) = 0$

$$\begin{aligned} x(n) &= x(n-1) + 5 \\ &= [x(n-2) + 5] + 5 = x(n-2) + 5 \cdot 2 \\ &= [x(n-3) + 5] + 5 \cdot 2 = x(n-3) + 5 \cdot 3 \\ &= \ldots \\ &= x(n-i) + 5 \cdot i \\ &= \ldots \\ &= x(1) + 5 \cdot (n-1) = 5(n-1). \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the arithmetical progression:

$$x(n) = x(1) + d(n-1) = 0 + 5(n-1) = 5(n-1).$$

b. $x(n) = 3x(n-1) \quad \text{for } n > 1, \quad x(1) = 4$

$$\begin{aligned} x(n) &= 3x(n-1) \\ &= 3[3x(n-2)] = 3^2 x(n-2) \\ &= 3^2[3x(n-3)] = 3^3 x(n-3) \\ &= \ldots \\ &= 3^i x(n-i) \\ &= \ldots \\ &= 3^{n-1} x(1) = 4 \cdot 3^{n-1}. \end{aligned}$$

Note: The solution can also be obtained by using the formula for the n term of the geometric progression:

$$x(n) = x(1) q^{n-1} = 4 \cdot 3^{n-1}.$$

c. $x(n) = x(n-1) + n \quad \text{for } n > 0, \quad x(0) = 0$

$$\begin{aligned} x(n) &= x(n-1) + n \\ &= [x(n-2) + (n-1)] + n = x(n-2) + (n-1) + n \\ &= [x(n-3) + (n-2)] + (n-1) + n = x(n-3) + (n-2) + (n-1) + n \\ &= \ldots \\ &= x(n-i) + (n-i+1) + (n-i+2) + \cdots + n \\ &= \ldots \\ &= x(0) + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}. \end{aligned}$$

d. $x(n) = x(n/2) + n \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 2^k)$

$$\begin{aligned} x(2^k) &= x(2^{k-1}) + 2^k \\ &= [x(2^{k-2}) + 2^{k-1}] + 2^k = x(2^{k-2}) + 2^{k-1} + 2^k \\ &= [x(2^{k-3}) + 2^{k-2}] + 2^{k-1} + 2^k = x(2^{k-3}) + 2^{k-2} + 2^{k-1} + 2^k \\ &= \ldots \\ &= x(2^{k-i}) + 2^{k-i+1} + 2^{k-i+2} + \cdots + 2^k \\ &= \ldots \\ &= x(2^{k-k}) + 2^1 + 2^2 + \cdots + 2^k = 1 + 2^1 + 2^2 + \cdots + 2^k \\ &= 2^{k+1} - 1 = 2 \cdot 2^k - 1 = 2n - 1. \end{aligned}$$

e. $x(n) = x(n/3) + 1 \quad \text{for } n > 1, \quad x(1) = 1 \quad (\text{solve for } n = 3^k)$

$$\begin{aligned}
x(3^k) &= x(3^{k-1}) + 1 \\
&= [x(3^{k-2}) + 1] + 1 = x(3^{k-2}) + 2 \\
&= [x(3^{k-3}) + 1] + 2 = x(3^{k-3}) + 3 \\
&= \ldots \\
&= x(3^{k-i}) + i \\
&= \ldots \\
&= x(3^{k-k}) + k = x(1) + k = 1 + \log_3 n.
\end{aligned}$$

Preliminaries

- Arithmetic progression

  $$a_n = a_1 + (n-1)d$$

  where

  - an is the nth term in the arithmetic progression
  - a1 is the first term in the arithmetic progression
  - n is the number of terms in the arithmetic progression
  - d is the common difference between consecutive terms in the arithmetic progression.

- Sum of terms of an AP

  $$S_n = \frac{n}{2}(a_1 + a_n)$$

  where

  - S_n is the sum of the first n terms in the arithmetic progression
  - a_1 is the first term in the arithmetic progression
  - a_n is the nth term in the arithmetic progression.

- Geometric progression

  $$a, ar, ar^2, ar^3, \ldots$$

  The $n-th$ term

  $$a_n = a_1 r^{n-1}$$

  - a_n is the nth term in the geometric progression
  - a_1 is the first term in the geometric progression
  - n is the number of terms in the geometric progression
  - r is the common ratio between consecutive terms in the geometric progression.

- Sum of a GP

  $$S_n = \sum_{i=1}^{n} a_1 r^{i-1} = a_1 + a_1 r + a_1 r^2 + \ldots + a_1 r^{n-1} = \frac{a_1(1-r^n)}{1-r}$$

  where:

  - S_n is the sum of the first n terms of the geometric progression
  - a_1 is the first term of the geometric progression

  - r is the common ratio between consecutive terms in the geometric progression.

- **Master Theorem:** Suppose we have a recurrence relation of the form

$$T(n) = aT(\frac{n}{b}) + f(n),$$

where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is a non-negative function. Then, the time complexity $T(n)$ can be asymptotically bounded as follows:

  (a) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
  (b) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
  (c) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

- **Subtract and Conquer Master Theorem:** Suppose we have a subtract and conquer algorithm that solves a problem of size n by subtracting a smaller subproblem of size m from n and then solves the remainder using a recursive call to itself. If the base case takes time $f(1)$, and the recurrence relation for the algorithm is given by

$$T(n) = T(n - m) + f(m) + g(n),$$

where $m < n$, then:

  (a) If $g(n) = O(n^k)$ for some constant k and $f(m) = O(m^c)$ for some constant c, then $T(n) = O(n^{k+1})$.
  (b) If $g(n) = O(n^k)$ for some constant k and $f(m) = \Theta(m^c)$ for some constant c, then $T(n) = \Theta(n^{k+1})$.
  (c) If $g(n) = \Omega(n^k)$ for some constant k and if $T(n - m) \leq cT(n)$ for some constant $c < 1$ and sufficiently large n, then $T(n) = \Theta(g(n))$.

22. Tracing the algorithm for n = 1 and n = 2 should help.

    The recurrence for the number of key comparisons is

$$C(n) = C(n-1) + 1 \text{ for } n > 1, C(1) = 0 \tag{1}$$

    Solving it by backward substitutions yields $C(n) = n - 1$

23. ———

24. In fact, even some seemingly simple algorithms have proved to be very difficult to analyze with mathematical precision and certainty. As we pointed out in Section 2.1, this is especially true for the average-case analysis.

    (a) Understand the experiment's purpose.
        - Compare different algorithms for the same problem
        - Check the accuraccy of theoretical assertion about the algorithm's efficiency
        - Develop an hypothesis about the algorithm efficiency class
    (b) Decide on the efficiency metric M to be measured and the measurement unit (an operation count vs. a time unit).
        - if time, get repeated measures
        - if stochastic algorithm, get repeated measures
    (c) Decide on characteristics of the input sample (its range, size (random or uniform?), and so on).

(d) Prepare a program implementing the algorithm (or algorithms) for the experimentation.

(e) Generate a sample of inputs.

(f) Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

(g) Analyze the data obtained.