DATA STRUCTURES AND ALGORITHMS

- LECTURE 3-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

3. Analysis of Algorithms

3.1. Introduction

- In the analysis of the algorithm, it is generally focused on CPU (time) usage, Memory usage, Disk usage, and Network usage. All are important, but the most concern is about the *CPU time*. Be careful to differentiate between:
  - **Performance:** How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
  - **Complexity:** How do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved by the code gets larger.

3.1. Introduction

- **Note:** Complexity affects performance, but not vice-versa.
- **Algorithm Analysis:** Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.
- Analysis of algorithms is the determination of the amount of *time and space resources* required to execute it.

3.1. Introduction

- **Why Analysis of Algorithms is important?**
  - To predict the behavior of an algorithm without implementing it on a specific computer.
  - It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
  - It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
  - The analysis is thus only an approximation; it is not perfect.
  - More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

3.1. Introduction

- **Types of Algorithm Analysis:**
  - **Best case:** Define the input for which algorithm takes ***less time*** or ***minimum time***. In the best case calculate the *lower bound of an algorithm*. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
  - **Worst Case:** Define the input for which algorithm takes a ***long time*** or ***maximum time***. In the worst calculate the *upper bound of an algorithm*. Example: In the linear search when search data is not present at all then the worst case occurs.
  - **Average case:** In the average case take all random inputs and calculate the ***computation time for all inputs***. And then we divide it by the total number of inputs (Average case=all time/total #).

3.2. Asymptotic analysis

- **Asymptotic Analysis** is defined as the big idea in analyzing algorithms.
- In Asymptotic Analysis, we ***evaluate the performance of an algorithm in terms of input size*** (we don't measure the actual running time).
- We calculate, how the time (or space) taken by an algorithm increases with the input size.

3.2. Asymptotic analysis

- **Asymptotic notation** is a way to describe the *running time* or *space complexity* of an algorithm based on the input size.
- It is commonly used in *complexity analysis* to describe how an algorithm performs as the size of the input grows.
- The three most commonly used notations are **Big O, Omega, and Theta.**

3.2. Asymptotic analysis

- **Big O notation (O):** This notation provides an *upper bound* on the growth rate of an algorithm's running time or space usage.
- It represents the ***worst-case scenario***, i.e., the *maximum* amount of time or space an algorithm may need to solve a problem.
- For example, if an algorithm's running time is O(n), then it means that the running time of the algorithm increases *linearly* with the *input size n* or *less*.

3.2. Asymptotic analysis

- **Omega notation (Ω):** This notation provides a *lower bound* on the growth rate of an algorithm's running time or space usage.
- It represents the ***best-case scenario***, i.e., the *minimum* amount of time or space an algorithm may need to solve a problem.
- For example, if an algorithm's running time is Ω(n), then it means that the running time of the algorithm increases *linearly* with the input size n or *more*.

3.2. Asymptotic analysis

- **Theta notation (Θ):** This notation provides both an *upper* and *lower bound* on the growth rate of an algorithm's running time or space usage.
- It represents the ***average-case scenario***, i.e., the amount of time or space an algorithm typically *needs* to solve a problem.
- For example, if an algorithm's running time is Θ(n), then it means that the running time of the algorithm increases *linearly* with the input size n.

3.2. Asymptotic analysis

- In general, the choice of *asymptotic notation* depends on the problem and the specific algorithm used to solve it.
- It is important to note that asymptotic notation does not provide an *exact* running time or space usage for an algorithm, but rather a *description* of how the algorithm *scales* with respect to input size.
- It is a useful tool for comparing the efficiency of different algorithms and for predicting how they will perform on large input sizes.

3.2. Asymptotic analysis

- The way to study the efficiency of an algorithm is to implement it and experiment by running the program on various test inputs while *recording the time spent* during each execution.
- Measuring elapsed time provides a reasonable reflection of an algorithm's efficiency.
- Given two algorithms for a task, how do we find out which one is better?

3.2. Asymptotic analysis

- One naive way of doing this is – to implement both the algorithms and run the two programs on your computer for different inputs and see which one takes *less time*.
- There are many problems with this approach for the analysis of algorithms:

3.2. Asymptotic analysis

- It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs second performs better.
- It might also be possible that for some inputs, the first algorithm performs better on one machine, and the second works better on another machine for some other inputs.

3.2. Asymptotic analysis

- **Asymptotic Analysis** is the big idea that handles the above *issues* in analyzing algorithms.
- In Asymptotic Analysis, *the performance of an algorithm in terms of input size* is evaluated (the actual running time is not measured).
- For example, let us consider the ***search problem*** (searching a given item) in a sorted array. The solution to above search problem includes:
  - **Linear Search** (order of growth is *linear*)
  - **Binary Search** (order of growth is *logarithmic*).

3.2. Asymptotic analysis

- To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms, let us say:
  - We run the Linear Search on a fast computer A, and
  - Binary Search on a slow computer B, and
  - Pick the constant values for the two computers so that it tells us *exactly how long* it takes for the given machine to perform the search in seconds.

3.2. Asymptotic analysis

- Let's say the constant for A is 0.2, and the constant for B is 1000, which means that A is 5000 times more powerful than B.
- For small values of input array size n, the fast computer may take less time.
- But, after a certain value of input array size, the *Binary Search* will definitely start taking ***less time*** compared to the *Linear Search* even though the Binary Search is being run on a slow machine.

3.2. Asymptotic analysis

- The reason is the order of growth of Binary Search with respect to input size is ***logarithmic***, while the order of growth of Linear Search is ***linear***.
- So the *machine-dependent constants* can always be ***ignored*** after a *certain value of input size*.
- Running times for this example:
  - Linear Search running time in seconds on A: **0.2*n**
  - Binary Search running time in seconds on B: **1000*log(n)**

3.2. Asymptotic analysis

Input Size	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

3.2. Asymptotic analysis

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- To overcome the ***challenges*** in the Experimental Analysis, Asymptotic Analysis is used.

3.2. Asymptotic analysis

- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.
- For example, say there are two sorting algorithms that take 1000*n*Log(n) and 2*n*Log(n) time respectively on a machine.
- Both of these algorithms are asymptotically the same (order of growth is n*Log(n)).
- So, with Asymptotic Analysis, we can't judge which one is better, as we ignore constants in Asymptotic Analysis.

3.2. Asymptotic analysis

- Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value.
- It might be possible that those large inputs are never given to your software and an asymptotically slower algorithm always performs better for your particular situation.
- So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

3.3. Asymptotic notations

- The main idea of asymptotic analysis is to have a *measure of the efficiency of algorithms* that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared.
- ***Asymptotic notations*** are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. Asymptotic Notations allow you to analyze an algorithm's running time by identifying its behavior as its input size grows.
- This is also referred to as an algorithm's ***growth rate***.

3.3. Asymptotic notations

- Based on the above three notations of Time Complexity there are three cases to analyze an algorithm:

1. **Worst Case Analysis (Mostly used)**: In the worst-case analysis, we calculate the upper bound on the running time of an algorithm.
2. **Best Case Analysis (Very Rarely used)**: In the best-case analysis, we calculate the lower bound on the running time of an algorithm.
3. **Average Case Analysis (Rarely used)**: In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs.

3.3. Asymptotic notations

- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are typically the set N of natural numbers or the set R of real numbers.
- Such notations are convenient for describing a ***running-time function T(n)***.

3.3. Asymptotic notations

- O-notation describes an ***asymptotic upper bound.*** We use O-notation to give an upper bound on a function, to within a constant factor.
- The formal definition of O-notation: For a given function *g(n)*, we denote by O(g(n)) (pronounced "big-oh of g of n") the *set of functions*:

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that: $\boldsymbol{0 \leq f(n) \leq cg(n)}$ for all $n \geq n_0\}$

- Figure 1(a) shows the intuition behind O-notation.

3.3. Asymptotic notations

- Just as O-notation provides an *asymptotic upper bound*, Ω-*notation* provides an ***asymptotic lower bound***.
- For a given function *g(n)*, *we denote* by Ω(g(n)) (pronounced "big-omega of g of n") the *set of functions*:

$\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that: $\boldsymbol{0 \leq cg(n) \leq f(n)}$ for all $n \geq n_0\}$

- Figure 2(b) shows the intuition behind Ω-notation.

3.3. Asymptotic notations

- We use Θ-notation for ***asymptotically tight bounds.***
- For a given function *g(n)*, we denote by Θ(g(n)) (pronounced "theta of g of n") the *set of functions*:

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that: $\boldsymbol{0 \le c_1 g(n) \le f(n) \le c_2 g(n)}$ for all $n \ge n_0\}$

- Figure 1(c) shows the intuition behind Θ-notation.

3.3. Asymptotic notations

- The definitions of O-, Ω-, and Θ- notations lead to the following theorem:

Theorem 1. For any two functions f(n) and g(n), we have:

$f(n) = \Theta(g(n))$, if and only if

$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

- We typically apply Theorem 1 to prove asymptotically tight bounds from asymptotic upper and lower bounds.

3.3. Asymptotic notations

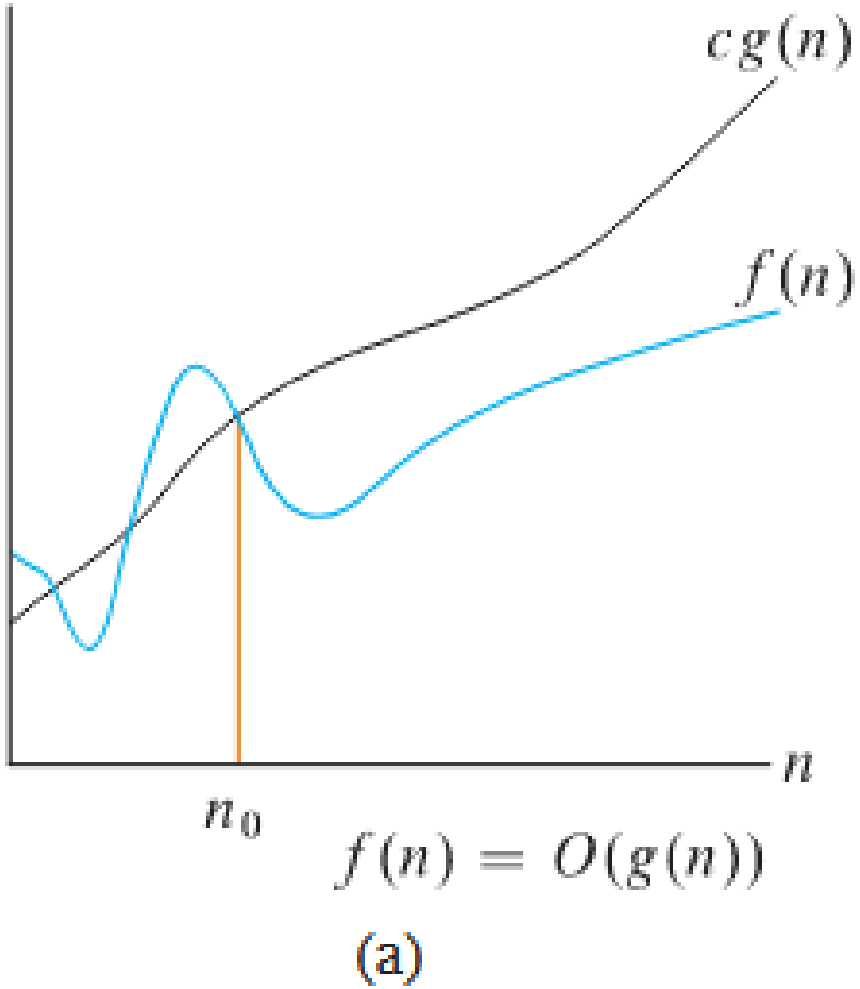

Figure 1(a) Graphic example of O-notation (upper bound)

3.3. Asymptotic notations

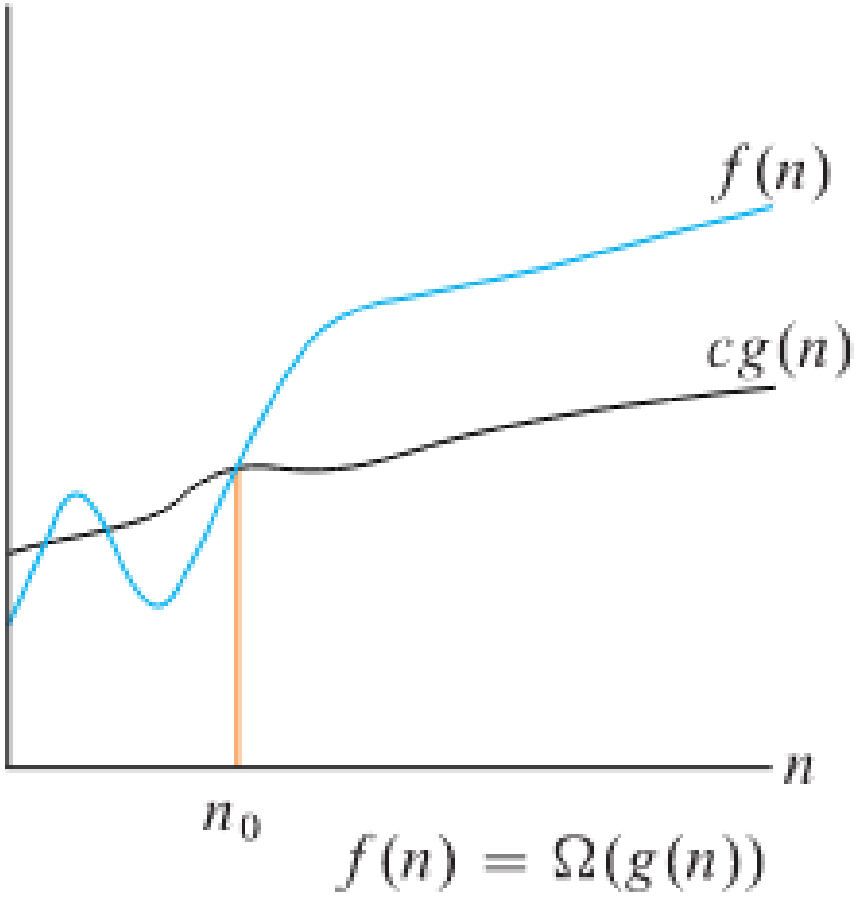

Figure 1(b) Graphic example of Ω-notation (lower bound)

3.3. Asymptotic notations

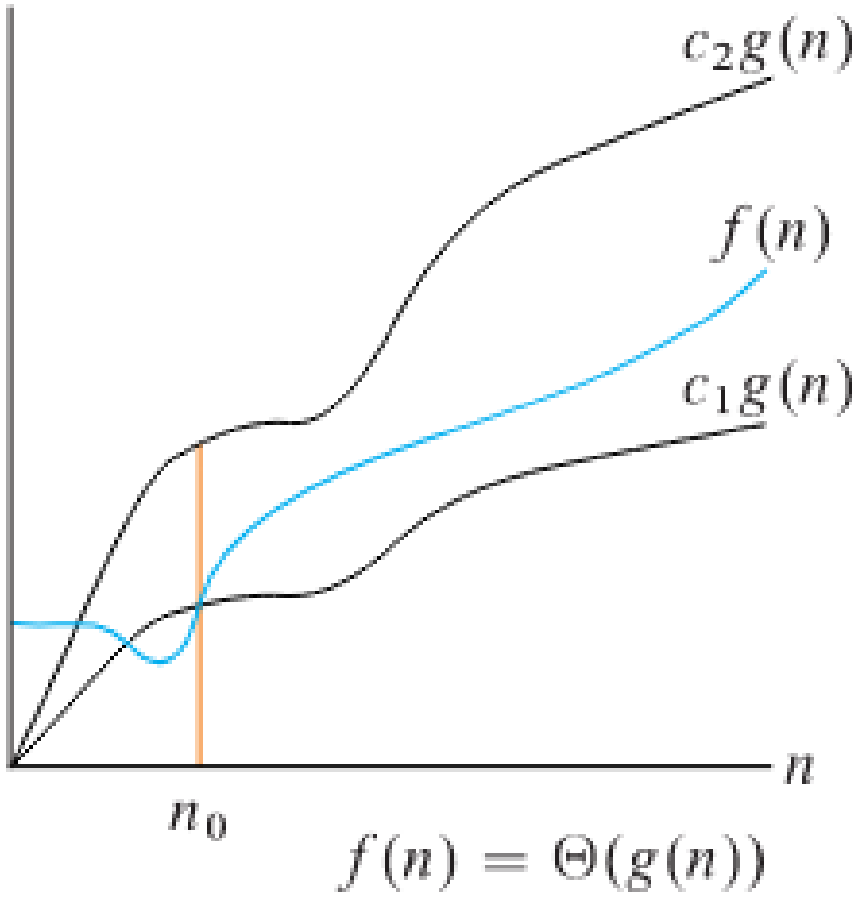

Figure 1(c) Graphic example of Θ-notation (tight bounds)

3.3. Asymptotic notations

- **Example 1**: The ***sum function*** has two statements. The first statement (line 2) runs in constant time, i.e., $\Theta(1)$, and second statement (line 3) also runs in constant time $\Theta(1)$. These two statements are consecutive statements, so the total running time is $\Theta(1)+\Theta(1)=\Theta(1)$.

```
1    int sum(int a, int b) {
2        int c = a + b;
3        return c
4    }
```

3.3. Asymptotic notations

- **Example 2**: The **sum** of n numbers in an array. Line 2 is a variable declaration. The cost is Θ(1). Line 3 is a variable declaration and assignment. The cost is Θ(2). Line 4-6 is a for loop that repeats *n* times. The body of the for loop requires Θ(1) to run. The total cost is Θ(*n*). Line 7 is a return statement. The cost is Θ(1). Steps:1, 2, 3, 4 are consecutive statements so the overall cost is Θ(*n*).

3.3. Asymptotic notations


```
1  int array_sum(int a, int n) {
2      int i;
3      int sum = 0;
4      for (i = 0; i < n; i++) {
5          sum = sum + a[i]
6      }
7      return sum;
8  }
```

3.3. Asymptotic notations

- **Recursive Calls:** To calculate the cost of a recursive call, we first transform the ***recursive function*** to a ***recurrence relation*** and then solve the recurrence relation to get the complexity.
- There are many techniques to solve the recurrence relation.
- **Example 3:** The factorial of number n.

3.3. Asymptotic notations


```
1    int fact(int n) {
2        if (n <= 2) {
3            return n;
4        }
5
6        return n * fact(n - 1);
7    }
```

3.3. Asymptotic notations

- We can transform the code into a *recurrence relation* as follows:

$$T(n) = \begin{cases} a & \text{if } n \leq 2 \\ b + T(n-1) & \text{otherwise} \end{cases}$$

- When **n** is 1 or 2, the factorial of **n** is *n* itself. We return the result in constant time ***a***. Otherwise, we calculate the factorial of $n-1$ and multiply the result by *n*. The multiplication takes a constant time ***b***. We use one of the techniques called ***back substitution*** to find the complexity.

3.3. Asymptotic notations

$$
\begin{aligned}
T(n) &= b + T(n-1) \\
&= b + b + T(n-2) \\
&= b + b + b + T(n-3) \\
&= 3b + T(n-3) \\
&= kb + T(n-k) \\
&= nb + T(0) \\
&= nb + a \\
&= \Theta(n)
\end{aligned}
$$