

## Worksheet: Asymptotic Bounds

---

### WORKSHEET WEEK 01: ASYMPTOTIC BOUNDS

#### 1.1 Introduction

**Goal:** The focus of this course is efficiency – creating algorithms that can work on large input data and handle complex structures sufficiently fast.

**Why use Big-O Notation?** It is convenient to measure speed of algorithms – for example, to find the best algorithms for a given problem. Or to find out which problems are easy (have fast algorithms) and which ones are hard (have only slow or unfeasible algorithms).

- Measuring the speed should not depend on the speed of the computing hardware – do not care about constant factors.
- Measuring the speed should not depend on how fast it works on very short inputs. (One can “cheat” for short inputs – just remember a large lookup table containing values for inputs of length  $n < n_0$  with precomputed correct answers. Clearly, this does not tell us anything about the performance of this algorithm for arbitrary inputs.)
- Measuring the speed should be conceptually easy, it should not take into account insignificant optimizations or count too many extra factors.

**Example:** Energy needed to lift a stone of mass  $m$  to the height  $h$  is  $mgh$ . (Is this the best-case estimate? The worst-case estimate? The exact value?)

##### 1.1.1 Running Time as a Complexity Measure

**The Worst Running Time function:** Given an algorithm, denote by  $T(n)$  the number of elementary steps that are needed to complete the algorithm for any input of length  $n$ . (It can be called the *upper bound* of the running time.)

**Discussions on the Worst-Case Running Time:**

- Is  $T(n)$  the upper bound also for inputs shorter than  $n$ ?
- Is  $T(n)$  a non-decreasing function (i.e. do longer inputs always imply a longer running time)?
- What counts as an elementary step? (Any CPU instruction? One line in a pseudocode? One comparison in a sorting algorithm?)
- Let  $T(n)$  be a numeric algorithm receiving single natural numbers as input. Does the worst running time function change, if the input numbers are provided in binary (instead of decimal) notation?
- What is the worst running time to multiply two square matrices of size  $n \times n$ ? What is the size of input in this case?

Sometimes it is common to have  $n$  as some important parameter of the input data (not necessarily the exact size of its encoding). For matrix tasks – the size of the matrices  $n$ . For graph problems – the number of vertices  $n$  and the number of edges  $m$ .

**Example:** In order to multiply two  $n \times n$  matrices using the “school algorithm”, we spend  $n$  multiplications and  $n - 1$  additions to calculate one entry in the result matrix. For example:

$$c_{ij} = a_{i1} \cdot b_{1j} + a_{i2} \cdot b_{2j} + \dots + a_{in} \cdot b_{nj} = \sum_{k=1}^n a_{ik} \cdot b_{kj}.$$

That is  $O(n)$  time. To complete the calculation for the entire matrix  $C = A \cdot B$  we should compute  $n^2$  such entries (for each pair  $i, j \in \{1, \dots, n\}$ ). So the total running time for matrix multiplication is  $O(n^3)$ .

**Note 1:** In strong accordance to the definition of the runtime, we should take into account that the length of input for a matrix multiplication task is  $2n^2$  – you need to input two matrices of size  $n \times n$ . If we denote this input length by  $m = 2n^2$ , then the running time becomes  $O(n^3) = O(m^{1.5}) = O(m\sqrt{m})$ , so it no longer looks as terrible. In theory books people still express the running time for matrix multiplication in terms of matrix size (not the square of the matrix size) – just because it is a more convenient parameter.

**Note 2:** There exist faster algorithms than the “school algorithm”. For example, [Strassen algorithm](#). It has runtime  $O(n^{\log_2 7}) \approx O(n^{2.807})$  where  $n$  is the size of the matrices being multiplied. The exponent  $\log_2 7$  is smaller/better than  $\log_2 8 = 3$ . To see actual performance gains (where Strassen algorithm is faster than the “school algorithm”), the matrices should be huge – the size of matrix  $n$  is multiple thousand.

In 2022 the fastest bound for matrix multiplication was discovered. It is  $O(n^{2.37188})$ ; see [Matrix Multiplication Algorithm](#).

**Note 3:** Matrix multiplication is of large practical importance (computer graphics, neural networks, etc). The theory of Big-O notation disregards constant factors – runtime of  $T(n) = n^3$  or  $T(n) = 1000n^3$  or  $T(n) = 0.001n^3$  is considered to be of the same “cubic complexity”. But in practice it is common to use GPU (massive parallel computations) to multiply matrices. Parallelism can only speed up an algorithm by a constant factor, but sometimes even constant factors matter.

## 1.1.2 Definitions

**Definition of Big-O:** Let  $g: \mathbb{N} \rightarrow \mathbb{R}_{0+}$  be a function from natural numbers (non-negative integers) to non-negative real numbers. Then  $O(g)$  is the set of all functions  $f: \mathbb{N} \rightarrow \mathbb{R}^{0+}$  such that there exist real constants  $c > 0$  and  $n_0 \in \mathbb{N}$  satisfying

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0.$$

**Definition of Big-Omega:** Let  $g: \mathbb{N} \rightarrow \mathbb{R}_{0+}$  be a function from natural numbers to non-negative real numbers. Then  $\Omega(g(n))$  is the set of all functions  $f: \mathbb{N} \rightarrow \mathbb{R}$  such that there exist real constants  $c > 0$  and  $n_0 \in \mathbb{N}$  satisfying  $\forall n \in \mathbb{N} (n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$ .

**Definition of Big Theta:** Let  $g: \mathbb{N} \rightarrow \mathbb{R}_{0+}$  be a function from natural numbers to non-negative real numbers. Then  $\Theta(g)$  is the set of all functions  $f: \mathbb{N} \rightarrow \mathbb{R}$  such that there exist positive constants  $c_1, c_2 > 0$  and  $n_0 \in \mathbb{N}$  satisfying

$$\forall n \in \mathbb{N} (n \geq n_0 \rightarrow 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)).$$

Informally, the following terms are also usable:

- If  $f(n) \in O(g(n))$ , then  $g(n)$  is called *asymptotic upper bound* of  $f(n)$ .
- If  $f(n) \in \Omega(g(n))$ , then  $g(n)$  is called *asymptotic lower bound* of  $f(n)$ .

- If  $f(n) \in \Theta(g(n))$ , then  $g(n)$  is called *asymptotic growth order* of  $f(n)$ .

All these concepts (Big-O, Big-Omega, Big-Theta) are related to calculus (real analysis); it is functional behavior as  $n \rightarrow \infty$ . Predicting the speed of an algorithm for short input lengths  $n$ , the dependence on  $n$  is typically quite complex (and we cannot ignore “lower order” terms). As  $n$  becomes very large, only the “dominant parts” in the expression  $f(n)$  matter.

## 1.2 Properties of Big-O, Big-Omega, Big-Theta

**Big-O and Limit of the Ratio:** If the following limit exists and is finite:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C < +\infty,$$

then  $f(n)$  is in  $O(g(n))$ .

**Big-O is transitive:** If  $f(n) \in O(g(n))$  and  $g(n) \in O(h(n))$ , then  $f(n) \in O(h(n))$ .

**Sum of two functions:** If  $f(n) \in O(h(n))$  and  $g(n) \in O(h(n))$ , then  $f(n) + g(n) = O(h(n))$ .

**All polynomials:** Any  $k$ -th degree polynomial  $P(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_1 n + a_0$  is in  $O(n^k)$ .

**Logarithms of any base:** If  $a, b > 1$  are any real numbers, then  $\log_a n = O(\log_b n)$ . Typically use just one base (usually, it is base 2 or base  $e$  of the natural logarithm, if you prefer that), and write just  $O(\log n)$  without specifying base at all.

The last result directly follows from the formula to change the base of a logarithm:  $\forall a, b, m > 1 \left( \log_a b = \frac{\log_m b}{\log_m a} \right)$ .

## 1.3 Problems

**Problem 1:** Are the following statement true or false? Prove or disprove them using the definitions of  $O(g(n))$ ,  $\Omega(g(n))$  or  $\Theta(g(n))$ :

- (A)  $f(n) = 13n + 7$  is in  $\Theta(n)$ .
- (B)  $f(n) = 3n^2 - 100n + 6$  is in  $O(n^2)$ . (Verify the definition that  $f(n) \in O(g(n))$ , where  $g(n) = n^2$ .)
- (C)  $f(n) = 3n^2 + 100n + 6000$  is in  $O(n^2)$ .
- (D)  $f(n) = 3n^2 - 100n + 6$  is in  $O(n\sqrt{n})$ .

**Answer:**

(A) True. Select  $c_1 = 13$ ,  $c_2 = 14$ ,  $n_0 = 7$ . We should verify

$$0 \leq c_1 \cdot n \leq f(n) \leq c_2 \cdot n.$$

Indeed,  $13n \leq 13n + 7$ , since  $0 \leq 7$ .

Also,  $13n + 7 \leq 14n$ , since  $7 \leq n$  (whenever  $n \geq n_0 = 7$ ).

(B) True. Select  $c = 3$  and  $n_0 = 34$ . Let us verify the inequalities:

$$0 \leq 3n^2 - 100n + 6 \leq 3n^2.$$

For  $n \geq 34$ , then

$$3n^2 - 100n + 6 \geq 3 \cdot 34 \cdot n - 100n + 6 = 2n + 6 \geq 0.$$

So, the expression is non-negative for sufficiently large  $n$  ( $n \geq 34$ ).

The other inequality  $3n^2 - 100n + 6 \leq 3n^2$ , since  $-100n + 6 \leq 0$  and  $6 \leq 100n$ .

(C) True. Select  $c = 5$  and  $n_0 = 100$ . At this point (when  $n \geq 100$ ) you can prove that  $100n \leq n^2$  and  $6000 \leq n^2$  and also  $3n^2 \leq 3n^2$ . Add all three inequalities to get  $3n^2 + 100n + 6000 \leq 5n^2$ .

(D) False. To disprove that  $f(n) = 3n^2 - 100n + 6$  is not in  $O(g(n))$  where  $g(n) = n\sqrt{n}$ , we show how to find an example value  $n \geq n_0$  such that  $3n^2 - 100n + 6 > c \cdot n\sqrt{n}$  for any positive constants  $n_0, c$ .

First, notice that  $3n^2 - 100n + 6$  is in  $\Omega(n^2)$ , for example  $3n^2 - 100n + 6 > n^2$  whenever  $n \geq 34$ .

Let us consider some positive constant  $c$ . Then  $n^2 \geq cn\sqrt{n}$  can be rewritten as  $\sqrt{n} \geq c$  or  $n \geq c^2$ . Once you consider natural numbers  $n \geq \max(34, c^2)$ , you will have **both** inequalities  $3n^2 - 100n + 6 \geq n^2$  and  $n^2 > cn\sqrt{n}$ ; combining them gives  $3n^2 - 100n + 6 > cn\sqrt{n}$ .

**Problem 2:** Let us have a zero-based dictionary  $D$  with  $n$  items from  $D[0]$  to  $D[n-1]$ .

```

LINEARSEARCH( $D, w$ )
1.   for  $i$  in RANGE( $0, n$ ):
2.       if  $w == D[i]$ :
3.           return FOUND  $w$  at location  $i$ 
4.   return NOT FOUND

```

Let  $T(n)$  be the worst-case running time for this algorithm. Find some asymptotic upper bound for  $T(n)$  – the “smallest” set  $O(g(n))$  such that  $T(n)$  is in  $O(g(n))$ .

**Answer:**

You can pick  $g(n) = n$  and argue that  $T(n)$  is in  $O(n)$ , i.e. the search time is linear in size of the array  $n$ .

If you can assume that the dictionary  $D$  contains entries in a sorted order, you could use binary search instead. It has much better runtime complexity:  $T(n) = \log_2 n$ .

**Problem 3:** What is the worst running time to find, if the given input  $m$  is a prime number. Assume that the input  $m$  is written in decimal notation using  $n$  digits.

Primality testing is done by the following algorithm testing divisibility by all numbers  $d \in \{2, 3, \dots, \lfloor \sqrt{m} \rfloor\}$ :

```

ISPRIME( $m$ )
1.   for  $d$  in RANGE( $2, \sqrt{m} + 1$ ):
2.       if  $m \% d == 0$ :
3.           return FALSE
4.   return TRUE

```

**Answer:**

Consider the number  $m$  containing, say,  $n = 100$  digits. In this case  $m \leq 10^{100}$  and  $\sqrt{m} \leq 10^{50}$ . To check, if such number is a prime number (in the worst case), we need to make about  $10^{50}$  operations.

In general, the time complexity of this algorithm is  $O(10^{\frac{n}{2}})$ . We see that the algorithm is extremely inefficient for (moderately) long inputs. On the other hand, checking primality of 100-digit numbers can be done very fast using an efficient, but probabilistic algorithm by Rabin-Miller. Here is Python code to find the largest 100-digit prime number:

```
import sympy

for i in range(1, 1000):
    if sympy.isprime(10**100 - i):
        print('Prime number 10**100-{}'.format(i))
```

Output looks like this:

```
Prime number 10**100-797
Prime number 10**100-911
```

**Problem 4:** Answer the following Yes/No questions:

- (A) For any  $g(n)$ , is the set of functions  $\Theta(g(n))$  the intersection of  $O(g(n))$  and  $\Omega(g(n))$ ?
- (B) Does every function  $f(n)$  defined for all natural numbers and taking positive values belong to the set  $\Omega(1)$ ?
- (C) Let  $f(n), g(n)$  be two functions from natural numbers to non-negative real numbers. Is it true that we have either  $f(n)$  in  $O(g(n))$  or  $g(n)$  in  $f(n)$  (or both)?
- (D) Does the definition of  $f(n)$  in  $O(g(n))$  make sense, if  $f(n)$  and  $g(n)$  can take negative values?
- (E) Are these two sets of functions  $O(\log_2 n)$  and  $O(\log_{10} n)$  the same? If not, find which one is larger (contains more functions)?
- (F) Let  $f(n)$  be a function from natural numbers to non-negative real numbers. Do we always have that  $f(n)$  is in  $O(f(n))$ , and  $f(n)$  is in  $\Omega(f(n))$  and  $f(n)$  is in  $\Theta(f(n))$ ? (In other words, is being in Big-O, in Big-Omega and in Big-Theta a reflexive relation?)
- (G) Let  $f(n), g(n), h(n)$  be functions from natural numbers to non-negative real numbers. It is known that  $f(n)$  is in  $O(g(n))$  and also  $g(n)$  is in  $h(n)$ . Can we always imply that  $f(n)$  is in  $O(h(n))$ . (In other words, is being in Big-O, in Big-Omega and in Big-Theta a transitive relation?)
- (H) Let  $f(n), g(n)$  be functions from natural numbers to non-negative real numbers. It is known that  $f(n)$  is in  $\Theta(g(n))$ . Can we always imply that  $g(n)$  is in  $\Theta(f(n))$ ? (In other words, is being in Big-Theta an equivalence relation?)
- (I) A function  $f(n)$  is defined for natural arguments and takes natural values. It is known that  $f(n)$  is in  $O(1)$ . Is it true that  $f(n)$  is a constant function:  $f(n) = C$  for all  $n \in \mathbb{N}$ .

**Answer:**

(A) True. We need to prove in two directions.

(1) If  $f(n) \in O(g(n))$  and  $f(n) \in \Omega(g(n))$ , then also  $f(n) \in \Theta(g(n))$ .

Indeed, if we have  $f(n)$  bound from above by  $c_1 \cdot g(n)$  for all  $n \geq n_1$ , and also  $f(n)$  bound from below by  $c_2 \cdot g(n)$  for all  $n \geq n_2$ , then we also have  $c_1 g(n) \geq f(n) \geq c_2 g(n)$  as soon as  $n \geq \max(n_1, n_2)$ .

This means that for all sufficiently large  $n$  all the values  $f(n)$  will be bound from both sides which is same as  $f(n) \in \Theta(g(n))$ .

(2) If  $f(n) \in \Theta(g(n))$  then both  $f(n) \in O(g(n))$  and  $f(n) \in \Omega(g(n))$  must hold.

Indeed, if there are constants  $c_1, c_2$  such that for all  $n \geq n_0$  we have  $c_1 g(n) \leq f(n) \leq c_2 g(n)$ , then the function  $f(n)$  is bound from above and from below. I.e. it belongs to  $O(g(n))$  and  $\Omega(g(n))$  where the same constants can be used.

(B) False. You could take any function  $f(n) = \frac{1}{n}$ . In this case all the values are non-negative, but there does not exist a positive constant  $c$  such that all the values  $\frac{1}{n} \geq c \cdot 1$  (in fact these values converge to 0).

Class  $\Omega(1)$  includes only those functions that have a positive (or infinite) lower limit. See [Lower Limit](#).

- (C) False. Most functions encountered in algorithm analysis are comparable, using Big-O notation: namely, either  $g(n)$  bounds  $f(n)$  from above (for all sufficiently large  $n$ ) or vice versa.

But it is not difficult to build functions that are not comparable. For example,

$$\left\{ \begin{array}{l} f(n) = n \\ g(n) = n(1 + \sin n) \end{array} \right\}$$

Here the function  $g(n)$  is not monotonous (it takes “random” values between  $n^0$  and  $n^2$ ). It is also possible to create two functions that are monotonous (non-decreasing) and still incomparable so that  $f(n) \notin O(g(n))$  and also  $g(n) \notin O(f(n))$ .

- (D) True. All the definitions still make sense, if functions  $f(n)$  and  $g(n)$  can take negative values. The important requirement is that functions  $f(n)$  and  $g(n)$  be *asymptotically non-negative* – i.e. they only take finitely many negative values. In this case  $n_0$  can be selected sufficiently large, so that  $0 \geq f(n)$  and  $0 < g(n)$  whenever  $n > n_0$ . (Which means that negative values  $f(n) < 0$  etc. can be simply ignored as the arguments  $n$  are not sufficiently large.)

**Problem 5:** Given a sequence  $a_i$  ( $i = 0, \dots, n-1$ ) we call its element  $a_i$  a *peak* iff it is a local maximum (at least as big as any of its neighbors):

$$a_i \geq a_{i-1} \text{ and } a_i \geq a_{i+1}$$

(In case if  $i = 0$  or  $i = n-1$ , one of these neighbors does not exist; and in such cases we only compare  $a_i$  with neighbors that do exist.)

- (A) Suggest an algorithm to find some peak in the given array  $A[0], \dots, A[n-1]$  and find its worst-case running time.
- (B) Suggest an algorithm that is faster than linear time to find peaks in an array. Namely, its worst-case running time should satisfy the limit:

$$\lim_{n \rightarrow \infty} \frac{T(n)}{n} = 0.$$

**Problem 6:** Order these functions in increasing order regarding Big-O complexity ( $f_i$  is considered “not larger” than  $f_j$  iff  $f_i \in O(f_j)$ ).

- $f_1(n) = n^{0.9999} \log_2 n$
- $f_2(n) = 10000n$
- $f_3(n) = 1.0001^n$
- $f_4(n) = n^2$

**Problem 7:** Order these functions in increasing order regarding Big-O complexity:

- $f_1(n) = 2^{2^{10000}}$
- $f_2(n) = 2^{10000n}$
- $f_3(n) = \binom{n}{2} = C_n^2$
- $f_4(n) = \binom{n}{\lfloor n/2 \rfloor}$
- $f_5(n) = \binom{n}{n-2}$
- $f_6(n) = n!$
- $f_7(n) = n\sqrt{n}$

**Problem 8:** Order these functions in increasing order regarding Big-O complexity:

- $f_1(n) = n^{\sqrt{n}}$
- $f_2(n) = 2^n$
- $f_3(n) = n^{10} \cdot 2^{n/2}$
- $\sum_{i=1}^n (i + 1).$

**Problem 9:** A black box  $\mathcal{B}$  receives two numbers  $k_1, k_2 \in \{1, \dots, n\}$  as inputs and returns a value  $v = \mathcal{B}(k_1, k_2)$ . What is the worst-case time complexity to find the maximum possible value  $v = \mathcal{B}(k_1, k_2)$  for any two inputs.

What if the black box receives permutations of  $n$  elements as its inputs?

**Problem 10:** Prove or disprove the following statement: If  $f(n)$  is in  $O(g(n))$  and also  $g(n)$  is in  $O(f(n))$ , then  $f(n)$  is also in  $\Theta(g(n))$  (and  $g(n)$  is in  $\Theta(f(n))$ ). (You can assume that  $f(n)$  and  $g(n)$  always take positive values.)