

Harvard University  
Computer Science 121  
Section Handout 9

## 1 Concept Review

### 1.1 Asymptotic Notation

**Definition 1.1 (Big-O)** For functions  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ ,  $f(n) = O(g(n))$  if there exists  $c, N$ , such that for all  $n > N$ ,  $f(n) \leq c \cdot g(n)$ . We can also write this as  $g(n) = \Omega(f(n))$ .

The intuition to have about these is that  $f = O(g)$  means “ $f \leq g$ ”, ignoring constant factors, for large enough  $n$ . Correspondingly  $g = \Omega(f)$  means “ $g \geq f$ ”, ignoring constant factors, for large enough  $n$ . These two statements are equivalent.

**Definition 1.2 (Theta)** For functions  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ ,  $f(n) = \Theta(g(n))$  if  $f(n) = O(g(n))$  and  $g(n) = O(f(n))$ .

This intuitively corresponds to  $f = g$ , ignoring constant factors, for large enough  $n$ .

**Definition 1.3 (Small-O)** For functions  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ ,  $f(n) = o(g(n))$  if for any  $c > 0$ , there exists  $N$ , such that  $n > N \Rightarrow f(n) < c \cdot g(n)$ . We can also write this as  $g(n) = \omega(f(n))$ .

The intuition for little-o is that if  $f(n) = o(g(n))$ ,  $f < g$ , again ignoring constant factors, and for large enough  $n$ .

These definitions let us compare the running time of various algorithms in a way somewhat reminiscent to the way we used mapping reductions to reason about the comparative difficulty of deciding and recognizing languages.

### 1.2 Time Complexity

$\text{TIME}(t(n)) = \{L : L \text{ is decided by a deterministic Turing machine that runs in } O(t(n))\}$ .

Does using a deterministic machine matter in this definition?

Does having a polynomial time algorithm guarantee solving the problem efficiently?

$$\mathcal{P} = \bigcup_k \text{TIME}(n^k).$$

When we say  $\langle M \rangle$  we mean a reasonable encoding of a Turing machine. When talking about decidability / recognizability, the representation often does not matter. Polynomial time algorithms, however, are on the size of the input. So, the size of your input matters, i.e. it matters how you represent inputs.

1. What is a reasonable way to represent numbers?
2. What about graphs?

## 2 Exercises

**Exercise 2.1** Which of the following relations hold?

1.  $64n + 2^5 = o(3n^4)$
2.  $12 \log_2(n) = o(2 \log_4(n^2))$
3.  $5n = \Omega(6n)$
4.  $2^n n^4 = \Theta(2^n)$
5.  $\omega(n^2) = O(2^n)$
6.  $\log_2(n) \Theta(2^n) = \omega(n^2)$

**Exercise 2.2** Give a proof or counterexample for the following claims:

1. If  $f = o(g)$  then  $f = O(g)$ .
2. If  $f \neq O(g)$  then  $g = O(f)$ .
3. If  $f = O(g)$ , and  $g = \Theta(h)$ , then  $h = \Omega(f)$
4. If  $f = O(g)$ , and  $h = O(g)$ , then  $f = \Theta(h)$

**Exercise 2.3** Recall that a TM computes a function  $f$  if on input  $w$  it halts with  $f(w)$  on the tape. Suppose we want a turing machine that computes  $F$  such that  $F(\langle n \rangle) = \langle F_n \rangle$ , where  $F_n$  is the  $n$ th fibonacci number. Show that this can't be done in polynomial time if the same encoding is used for  $\langle n \rangle$  and  $\langle F_n \rangle$ .

**Exercise 2.4** Consider an arcade game where the Pac-Man starts in the  $(1,1)$  (top left) cell of an  $n$  by  $n$  matrix, and at each step can move either downward or rightward. When Pac-Man enters cell  $(i,j)$ , it collects  $M_{i,j}$  coins. Give a polynomial-time algorithm that, given matrix  $M$ , computes a path for Pac-Man to collect as many coins as possible.