# MACHINE MODEL AND TIMING ANALYSIS NOTATION

## Introduction

This course has two major goals.

(1) To teach certain fundamental combinatorial (as opposed to numerical) algorithms.

(2) To teach general techniques for the design and analysis of algorithms.

The first question to address is "What is analysis of algorithms?". We are interested in analyzing the *complexity* of algorithms. Complexity has many aspects:

- "conceptual complexity", i.e. how difficult is it to write and verify the correctness of the algorithm in the first place. We do not have any way of quantifying the complexity of an algorithm in this sense. All other things being equal, we would like to design the simplest and easiest to understand algorithm possible, but we'll not formally talk about these notions.
- The length of the algorithm. This sense of the complexity of an algorithm can be quantified and is a useful measure in other contexts. This measure is related to Kolmogorov complexity. This measure should not be confused with the "conceptual complexity" of an algorithm. We don't consid
- How long does e *time complexity* of the algorithm. This imary focus of the complexity analysis done in this course.
- How much storage does the algorithm requi *ce complexity* of the algorithm. This measure is also a concern in t done in this course, although perhaps not as central as time complexity.



## Machine Model and Time Complexity of Algorithms

Let $N$ denote the set of non-negative integers, i.e. $\{0,1,\cdots\}$ and let $R$ denote the set of real numbers. We use $N^+$ and $R^+$, respectively, to denote the positive integers and the positive reals. Finally, $R^* = R^+ \cup \{0\}$ is used to denote the non-negative reals. The *worst-case* time complexity of an algorithm is expressed as a function

$$T : N \to R^*$$

where $T(n)$ is the maximum number of "steps" in any execution of the algorithm on inputs of "size" $n$. For example, $T(n) = 3 \cdot n^2$ means that on inputs of size $n$ the algorithm requires up to $3 \cdot n^2$ steps. To make this more precise, we must clarify what we mean by input "size" and "step".

(a) A general definition of the input size to an algorithm is the number of bits in the input. This is not a very convenient measure for most of the problems we deal with in this course, i.e. for problems considered in this course there is another measure of the input size which is more natural to talk about. When we are dealing with algorithms which solve problems of searching and sorting, the measure of input size we adopt is the number of elements which

are to be searched through or sorted. When we are dealing with algorithms which solve graph problems, the measure of input size we adopt is expressed in terms of the number of edges in the graph and the number of nodes in the graph. In this case, the worst-case time complexity of the algorithm is expressed as a function of both of these measures of the input size.

(b) A "step" of the algorithm can be defined precisely if we fix a particular machine on which the algorithm is to be run. We do not want to restrict ourselves to any particular machine, so we will be a bit vague as to exactly what a step is (in most cases, when we analyze the time complexity of a particular algorithm to solve a particular problem, we will specify in more detail exactly what the time complexity measures): basically, a step is anything we can reasonably expect a computer to do in a fixed amount of time. Typical examples are:

- Performing an arithmetic operation ($+, -, \div, \times$).
- Comparing two numbers.
- Logic: and, or, not.
- Assigning a value to a variable.
- Following a pointer or indexing into an array.

There is a problem with assuming that these steps only take a fixed amount of time, e.g. an arithmetic operation takes constant time only if the numbers involved are of bounded size.



**Uniform Cost Crite**



**Logarithmic Cost C** here $\alpha$ is the maximum number involved in t t the time to perform a simple step is proportional to the length of the numb mber of bits) which is $\log \alpha$. Even this isn't always quite true, for examp two numbers!



The uniform cost criterion makes the analysis simple count the size of the numbers, which implies that it is a bit unrealistically mic cost criterion is more realistic. In this course, unless otherwise specified, we stick to the uniform cost criterion.

Recall that $T(n)$ is the maximum number of steps required for inputs of size $n$, i.e. the worst-case time complexity of the algorithm. Another useful measure of time complexity is *average-case* time complexity. The average-case time complexity of an algorithm, $\bar{T}(n)$, is the *expected* number of steps required for inputs of size $n$. For this to be a well-defined concept, we must define a probability space on the set of inputs of size $n$ for each value of $n$. The main advantage of measuring average-case time complexity instead of worst-case time complexity is that the average-case time complexity of an algorithm gives a more realistic idea of what to expect when the algorithm is run on a problem of size $n$, given that it is possible to determine a probability distribution on the set of inputs to the algorithm which reflects the true probability distribution on inputs of size $n$. The disadvantage of measuring average-case time complexity instead of worst-case time complexity are:

- The probability distribution on inputs depends on the application where the algorithm is to be used, and is usually not even known.
- Even when the probability distribution is known, average-case time complexity analysis is usually much more difficult than the worst-case time complexity analysis for the same algorithm.

† in this course, unless otherwise stated, all logarithms are base two. Also, $lg$ indicates $\log_2$.

- In some real-time applications, it is critical that on each input the algorithm is guaranteed to finish within a particular time bound. The worst-case time complexity analysis is useful to see if the criterion is met, whereas average-case time complexity analysis is not at all useful for this purpose.

There is another time complexity measure known as the *amortized* time complexity, which is studied in the COSC 4101/5101 course. In this course we concentrate mostly on worst-case time complexity, but we'll see some examples of average-case.

**Notations for asymptotic growth rate**

We are often only interested in obtaining a rough estimate of an algorithm's time complexity. Since in the interest of generality, we measure time in somewhat vaguely defined "steps", there is little point in most cases of spending too much effort to obtain a precise expression for this number of steps. We are happy with a reasonable approximation.

A style of time complexity analysis that is very common in algorithm analysis is the so called *asymptotic analysis*. Except on rare occasions, this is the type of analysis we'll work within this course (you'll be happy to know it's the easiest!). In asymptotic analysis we cavalierly drop low order terms and multiplicative constants from the time complexity function $T(n)$. For example, if the function is $3 \cdot n^2 + 7 \cdot \log n$, we are only interested in the "$n^2$" part. $7 \cdot \log n$ is a low order term and 3 is "just" a multiplicative constant. Some special mathematical notation has been developed to ex below.



**"Big-Oh" Notation**

To express *asymptotic upper bounds* on running tim troduce the "big-oh" notation defined as follows:

Consider functions $f : N \to R^*$ and $g : N \to R^*$. We say $g = O(f)$ iff $\exists$ constants $c > 0$, $n_o \geq 0$ such that $\forall n \geq n_o$, $g(n) \leq c \cdot f(n)$.

Informally $g(n) = O(f(n))$ (pronounced gee is big oh of ef, or simply, gee is oh of ef) says that $g$ grows no faster than $f$, to within a constant factor. In words, $g(n) = O(f(n))$ if for all sufficiently large $n$ (i.e., $\forall\ n \geq n_o$ ) $g(n)$ is bounded from above by $f(n)$ – possibly multiplied by a positive constant. We say $f(n)$ is an *asymptotic upper bound* for $g(n)$. Figure 1 shows in terms of a chart what it means for $g(n)$ to be $O(f(n))$.

**Example 1:** $f(n) = 3 \cdot n^2 + 4 \cdot n^{3/2}$ is $O(n^2)$. This is because $3 \cdot n^2 + 4 \cdot n^{3/2} \leq 3 \cdot n^2 + 4 \cdot n^2 \leq 7 \cdot n^2$ . Thus, pick $n_o = 0$ and $c = 7$. For all $n \geq n_o$ , $f(n) \leq c \cdot n^2$ . □

**Example 2:** $f(n) = (n+5)^2$ is $O(n^2)$ . This is because $(n+5)^2 = n^2 + 10 \cdot n + 25$ . Check (with elementary algebra) that for all $n \geq 7$ , $n^2 + 10 \cdot n + 25 \leq 3 \cdot n^2$ . Thus, pick $n_o = 7$ and $c = 3$. For all $n \geq n_o$ , $f(n) \leq c \cdot n^2$ . □

Why is this useful? In many cases obtaining the exact time complexity function $T(n)$ of an algorithm is a very laborious task and yields a messy function. However, it is often relatively easy to find a much simpler function $f(n)$ that is an asymptotic upper bound for $T(n)$ and prove that

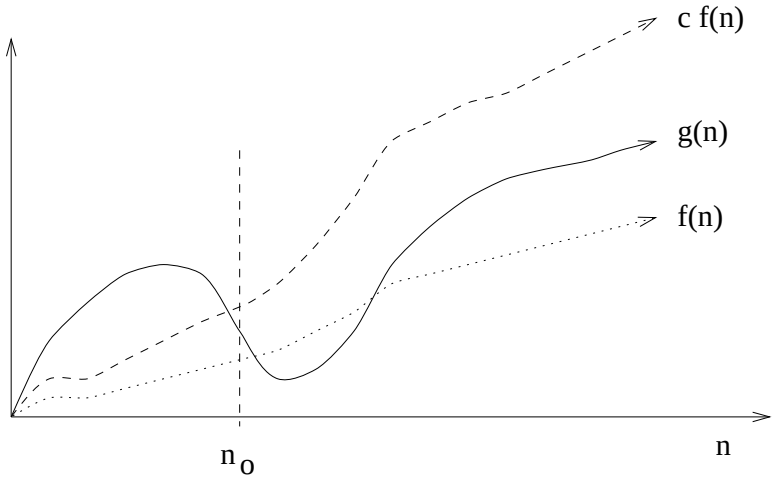


**Figure 1:** $g(n)$ is $O(f(n))$.

$T(n)$ is $O(f(n))$ and leave it at that (without bothering to find an exact expression for $T(n)$). Of course, we want to find the best asymptotic upper bound on $T(n)$ possible. For example, if $T(n) = 3 \cdot n^2 + 4 \cdot n^{3/2} + 5 \cdot \log n + 16$, then $T(n)$ is $O(n^2)$. But $T(n)$ is also $O(n^3)$, $O(n^4)$, ..., $O(n^{100})$, ... To say that $T(n)$ is $O(n^{100})$ is true but quite irrelevant. We want to find a simple asymptotic upper bound that is as small as possible. $f(n)$ is as small as possible for $T(n)$ if $T(n)$ is $O(f(n))$ and for a s also the case that $f(n)$ is ble for $O(g(n))$. In words, $f$ $T(n)$ if $f(n)$ is an pto as upper bounds for asymptotic upper bounds for $T(n)$ are also asymptotic upper bounds for $f(n)$. ve, $f(n) = n^2$ is as small an asymptotic upper bound as possible and so $n$. We much prefer to state $f(n)$ as an asymptotic upper bound on $T(n)$ r $f(n)$ is simpler. On the other hand, if $h(n) = n^2 \log n$, then $T(n)$ is $O(h$ small an asymptotic upper bound as possible (try to prove this!).



**"Big-Omega" Notation**

There is a similar notation for *asymptotic lower bounds*, the "big-omega" notation.

Consider functions $f : N \to R^*$ and $g : N \to R^*$. We say $g = \Omega(f)$ iff $\exists$ constants $c > 0$, $n_o \geq 0$ such that $\forall n \geq n_o$, $g(n) \geq c \cdot f(n)$.

If $g = \Omega(f)$ we say $g$ is $\Omega(f)$. In words, $g(n)$ is $\Omega(f(n))$ if for all sufficiently large $n$, $g(n)$ is bounded from below by $f(n)$ – possibly multiplied be a positive constant. We say $f(n)$ is an *asymptotic lower bound* for $g(n)$. Analogous to the case for asymptotic upper bounds, we are interested in finding a simple asymptotic lower bound which is as large as possible.

Notice the symmetry between "big-oh" and "big-omega": if $g(n)$ is $O(f(n))$ then $f(n)$ is $\Omega(g(n))$.

**Definition:** The tight asymptotic bound: $g = \Theta(f)$ if and only if $g$ is both $O(f)$ and $\Omega(f)$.

Thus, if $g(n)$ is $\Theta(f(n))$ then asymptotically $g(n)$ and $f(n)$ grow at the same rate within a positive multiplicative constant. In terms of this notation our previous concept simplifies to: $f(n)$ is an asymptotic upper bound (lower bound) on $T(n)$ which is as small as possible (as large as possible) iff $T(n)$ is $\Theta(f(n))$.

We can also extend these notations so that they include functions that may be negative or undefined on some finite initial segment of $N$. It is only necessary that there be a constant $k \geq 0$ above which the function is always defined and nonnegative. For example, we can talk about $O(\frac{n}{\log n})$ without worrying that this function is not defined when $n$=0 or 1.

**Some Examples:**

$n = O(n^2)$
$n^2 = \Omega(n)$
$3n + 1 = \Theta(n)$
$\log_2 n = \Theta(log_3 n)$
$\log n = O(n)$
$\log n \neq \Theta(n)$
$n = \Theta(n)$
$n \log n = \Omega(n)$
$lg\ n = O(n^{\varepsilon})$, for any constant real $\varepsilon > 0$ no matter how small.
$n^k = O(n^l)$ for all constants $k \leq l$
$n^k = O(2^n)$ for all constants $k > 0$
$2n^2 + 3n = O(n^2)$
$\sum_{i=1}^{n} \log i = \Theta(n$



The proof of the last 

$$\sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n =$$



$$\sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq \sum_{i=n/2}^{n} \log n/2 \geq n(log \qquad \forall n \geq 4.$$

The sets $O$, $\Omega$, and $\Theta$ satisfy a lot of nice properties. Here are some of them. Try to prove them.

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

(2) $g(n) = \Theta(f(n))$ if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

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

(4) If $a$>0 and $b$ are constants and $f(n) \geq 1$, then $a \cdot f(n) + b = \Theta(f(n))$.

(5) Transitivity: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$. Transitivity holds for $\Theta$ and $\Omega$ as well.

(6) $f(n) + g(n) = \Theta(max\{f(n), g(n)\})$. This is called the rule of sum I.

(7) If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then
(7a) $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ (the rule of sum II), and
(7b) $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$ (the rule of product).

Facts (6) and (7) are often used in the time complexity analysis of algorithms. Facts (6) and (7a) are useful for finding an asymptotic upper bound on the time complexity of an algorithm which is composed of the execution of one algorithm followed by the execution of another algorithm. fact (7b)is useful for determining the time complexity of nested loops.

In the analysis of an algorithm, each of these rules can be applied *only a constant number of times* for the analysis to be valid.

**Proofs of some of the above Facts:**

(5) Transitivity:
$f(n) = O(g(n))$ and $g(n) = O(h(n))$ imply there exist constants $c_1 > 0$, $c_2 > 0$, $n_1 \geq 0$, $n_2 \geq 0$ such that for all $n \geq n_1$, $f(n) \leq c_1 g(n)$, and for all $n \geq n_2$, $g(n) \leq c_2 h(n)$. These imply that $f(n) \leq c\ h(n)$ for all $n \geq n_o$, where $c = c_1 c_2$, and $n_o = \max\{n_1, n_2\}$.

(6) Rule of Sum I:
$f(n) + g(n) = \Theta(max\{f(n), g(n)\})$ is implied by the fact that $[f(n) + g(n)] / 2 \leq \max\{f(n), g(n)\} \leq [f(n) + g(n)]$. (Note that $f$ and $g$ by assumption are nonnegative functions.)

(7b) Rule of Product:
From the premise we have $f_1(n) \leq c_1 g_1(n)$ for all $n \geq n_1$, and $f_2(n) \leq c_2 g_2(n)$ for all $n \geq n_2$. These imply that $f_1(n) f_2(n) \leq c g_1(n) g_2(n)$ for all $n \geq n_o$, where $c = c_1 c_2$, and $n_o = \max\{n_1, n_2\}$. □



**Example of a bad analysis:**

Let $g(n) = 2$ for $i=1$ to $n$ by (7b) $n$ times with respect to $g(n)$ to analyze an $\prod_{i=1}^{n} g(n)$ y) conclude that $\prod_{i=1}^{n} g(n)$ is $O(1)$. It is easy to s (1). What went wrong is that we are applying rule (7b) a non-constant number of the algorithm. □



**Other Asympotic Notations**

*Little oh*: $g(n) = o(f(n))$ means $g(n) = O(f(n))$, but the upper bound $f(n)$ is growing much faster than $g(n)$ asymptotically. That is, for any constant $c > 0$ (no matter how small it is), there exists a constant $n_o$, such that for all $n \geq n_o$, we have $g(n) < c\ f(n)$.

*Little ω*: $g(n) = \omega(f(n))$ if $f(n) = o(g(n))$. That is, asymptotically $f(n)$ grows much slower than $g(n)$. That is, for any constant $c > 0$ (no matter how large it is), there exists a constant $n_o$, such that for all $n \geq n_o$, we have $g(n) > c\ f(n)$.

**Asymptotics by Limits:**

The relationship between the rates of growth of certain functions can be obtained by using the following facts. Suppose $\lim_{n \to \infty} \frac{f(n)}{g(n)} = L$. If this limit exists, then we infer the following:

If $L = 0$, then $f(n) = o(g(n))$.
If $0 < L < \infty$, then $f(n) = \Theta(g(n))$.
If $L = \infty$, then $f(n) = \omega(g(n))$.

**Example 3:** $n=o(n^2)$ and $n^{1.3}=\omega(n \log n)$.

**Example 4:** $f(n)=o(g(n))$ is not equivalent to [ $f(n) = O(g(n)$ *and* $f(n)\neq\Theta(g(n))$ ].
To see this, consider the following example. $g(n) = n$, and $f(n) = n$ for even values of $n$, and $f(n) = 1$ for odd values of $n$. We see that $f(n) \leq g(n)$ for all $n \geq 1$. Hence, $f(n) = O(g(n))$. We also see $f(n)\neq\Theta(g(n))$. However $f(n)\neq o(g(n))$ since it is not true that for all constants $c > 0$ $f(n)<c\ g(n)$ for all sufficiently large $n$. As a counter-example, take $c=0.5$.
Similarly, $f(n)=\omega(g(n))$ is not equivalent to the statement [ $f(n) = \Omega(g(n)$ *and* $f(n)\neq\Theta(g(n))$ ].
□

**Time Complexity of Problems**

So far, we have talked about the time complexity of particular algorithms to solve a problem. In general, we are much more interested in characterizing the time complexity of the problem itself.

**Definition:** $f(n)$ is an asymptotic upper bound on the time complexity of a *problem* if there exists an algorithm to solve the problem with time complexity $T(n)$ such that $T(n)$ is $O(f(n))$.

**Definition:** $f(n)$ is an asymptotic lower bound on the time complexity of a *problem* if for *every* algorithm which solve the problem, the time complexity $T(n)$ of the algorithm is $\Omega(f(n))$.



Our goal is to completely characterize the time complexity of a problem, i.e. we would love to say that the time complexity of the problem is $\Theta(f(n))$, where $f(n)$ is a simple function. In general, we are not able so completely . To prove that $f(n)$ is an asym blem, we must design an algorithm which solv )which is $O(f(n))$. This is the most common t lower bound on the time complexity of a problem is in general a much harde e must sho w that *any* algorithm, whatsoever, which solves the proble its time complexity $T(n)$ is $\Omega(f(n))$. There are very few known non-tri ounds for important problems, because they are so hard to prove. We present only one such bound in this course (see the discussion on the decision tree lower-bound model).





**Example of the time complexity analysis of a simple algorithm**

Selection sort is an algorithm for sorting a list. It works roughly as follows: First it finds the smallest element in the list and interchanges it with the first element of the list; then it finds the second smallest element and interchanges it with the second element of the list, etc. Let $A$ be the array filled with keys that is to be sorted. The procedure *swap*$(a, b)$ interchanges the value of $a$ with the value of $b$.

```
     procedure Selection-Sort ( var A : array [1 .. n] );
     var i , j , smallest ;
     begin
(1)      for i ← 1 to n − 1 do begin
(2)          smallest ← i
(3)          for j ← i to n do
(4)              if A[j] < A[smallest] then smallest ← j;
(5)          swap ( A[i] , A[smallest] )
```

```
        end
    end
```

We first calculate the time for the $i^{th}$ iteration of the main loop (which starts at line (1)). Then we obtain the time complexity of the entire algorithm by summing up the time complexities of all the iterations ($i = 1, 2, \cdots, n-1$). In the $i^{th}$ iteration, we have to account for:

- line (2): takes constant or $O(1)$ time.
- line (3): takes time equal to the number of the repetitions of the loop times the time for line (4). The loop is repeated $n - i + 1$ times. The time for line (4) is $O(1)$. Thus, by the rule of products, the time for line (3) is $O(n - i + 1)$.
- line (5): takes $O(1)$ time.

Thus, the $i^{th}$ iteration of the main loop is the total time for lines (2)–(5), which is $O(n - i + 1)$ by the rule of sum I. The total time of the algorithm is,

$$O\left( \sum_{i=1}^{n-1} ( n - i + 1 ) \right)$$

But,



$$\sum_{i=1}^{n-1} (n - i + 1) = n + (n-1) + (n-2) + \cdots + 2 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1 .$$

Thus, $T(n)$ is $\Theta(n^2)$. □





**Exercises:**

(1) Assume $f(n)$ and $g(n)$ are arbitrary functions to positive integers. Are the following statements true or false? Expl

(a) Either $f(n) = O(g(n))$ or $g(n) = O(f(n))$ must hold.

(b) If $f(n) = O(F(n))$ and $g(n) = O(G(n))$, then $f(n)/g(n) = O(F(n)/G(n))$.

(c) If $f(n) = \Theta(F(n))$ and $g(n) = \Theta(G(n))$, then $f(n)/g(n) = \Theta(F(n)/G(n))$.

(2) Give the most simplified answers:

(a) $4\,n^3 + 6\,n^2\,lg\,n + 1200 = \Theta(\ ?\ )$.

(b) $\dfrac{5\,n^6 + O(n^3\,lg\,n)}{3\,n^2\,lg\,n + O(n)} = \Theta(\ ?\ )$.

(c) $10^9 n^2 + 10^{10^{10}}\,n\,lg\,n + 3 \cdot 2^n = \Theta(\ ?\ )$.

(d) $n^{\frac{lg\,lg\,n}{lg\,n}} = \ ?\ .$

(3) What is the difference between $O(2^n)$ and $2^{O(n)}$ ?

(4) Show that $2^{O(lg\ lg\ n)} = lg^{O(1)}\ n$.

(5) The **set disjointness problem** is defined as follows: We are given two sets $A$ and $B$, each contaning $n$ arbitrary numbers. We want to determine whether the intersection of $A$ and $B$ is empty, i.e., whether they have any common element.

(a) Design the most efficient algorithm you can for this problem and analyze its worst-case time complxity.

(b) Is your algorithm asymptotically the fastest possible? (To answer this question negatively all you need to do is to come up with a faster algorithm. But to answer it affirmatively, you need to establish a lower bound on the time complexity of the *problem* itself. This is a rather hard task and you are not yet equipped to answer such a question! So, continue the course.)