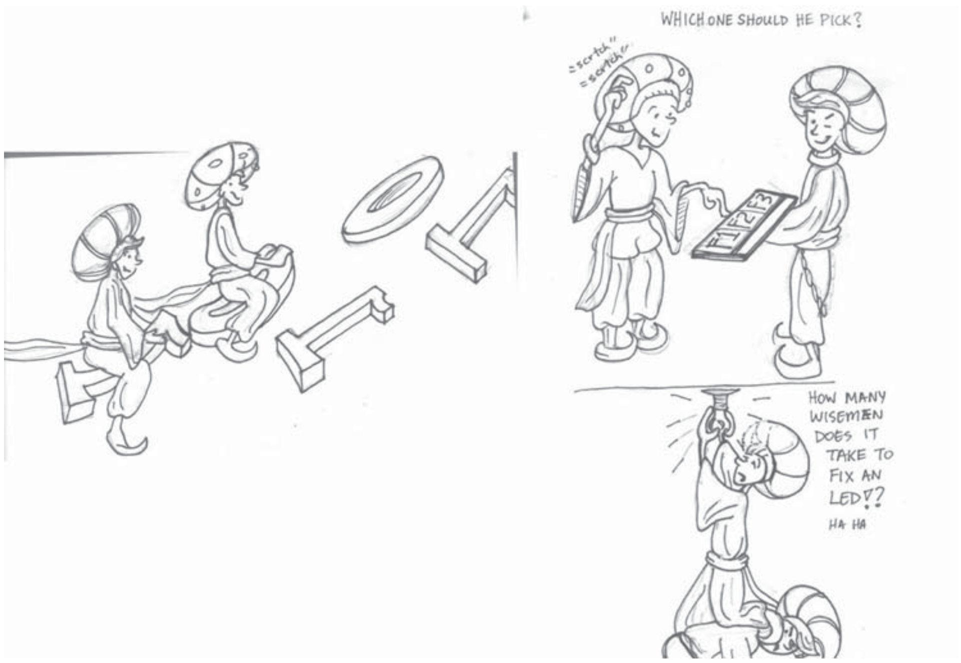
PART FOUR

Appendix

22 Existential and Universal Quantifiers


WHICH ONE SHOULD HE PICK?
"scrtch"
"scrtch"
HOW MANY WISEMEN DOES IT TAKE TO FIX AN LED?!?
HA HA

Existential and universal quantifiers provide an extremely useful language for making formal statements. You must understand them. A game between a prover and a verifier is a level of abstraction within which it is easy to understand and prove such statements.

The *Loves* Example: Suppose the relation (predicate) $Loves(p_1, p_2)$ means that person p_1 loves person p_2. Then we have

Expression	Meaning
$\exists p_2\ Loves(Sam, p_2)$	"Sam loves somebody."
$\forall p_2\ Loves(Sam, p_2)$	"Sam loves everybody."
$\exists p_1 \forall p_2\ Loves(p_1, p_2)$	"Somebody loves everybody."
$\forall p_1 \exists p_2\ Loves(p_1, p_2)$	"Everybody loves somebody."
$\exists p_2 \forall p_1\ Loves(p_1, p_2)$	"Theres one person who is loved by everybody."
$\exists p_1 \exists p_2\ (Loves(p_1, p_2)$ and $\neg Loves(p_2, p_1))$	"Somebody loves in vain."

Definition of Relation: A relation like $Loves(p_1, p_2)$ states for every pair of objects (say $p_1 = Sam$ and $p_2 = Mary$) that the relation either holds between them or does not. Though we will use the word *relation*, $Loves(p_1, p_2)$ is also considered to be a *predicate*. The difference is that a predicate takes only one argument and hence focuses on whether the property is *true* or *false* about the given tuple $\langle p_1, p_2 \rangle = \langle Sam, Mary \rangle$.

Representations: Relations (predicates) can be represented in a number of ways.

Functions: A relation can be viewed as a function mapping tuples of objects either to *true* or to *false*, for example, $Loves : \{p_1 | p_1 \text{ is a person }\} \times \{p_2 | p_2 \text{ is a person }\} \Rightarrow \{true, false\}$.

Set of Tuples: Alternatively, it can be viewed as a set containing the tuples for which it is true, for example $Loves = \{\langle Sam, Mary \rangle, \langle Sam, Ann \rangle, \langle Bob, Ann \rangle, \ldots\}$. $\langle Sam, Mary \rangle \in Loves$ iff $Loves(Sam, Mary)$ is true.

Directed Graph Representation: If the relation only has two arguments, it can be represented by a directed graph. The nodes consist of the objects in the domain. We place a directed edge $\langle p_1, p_2 \rangle$ between pairs for which the relation is true. If the domains for the first and second objects are disjoint, then the graph is bipartite. Of course, the *Loves* relation could be defined to include $Loves(Bob, Bob)$. See Figure 22.1.

Quantifiers: You will be using the following quantifiers and properties.

The Existential Quantifier: The quantifier $\exists$ means that there is at least one object in the domain with the property. This quantifier relates to the Boolean

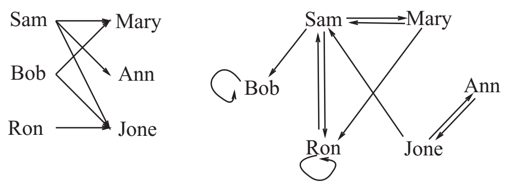
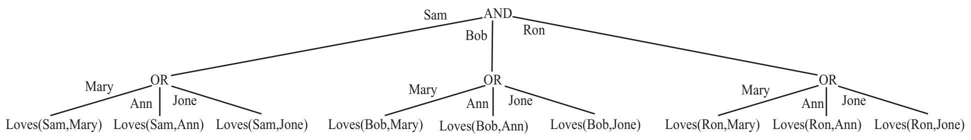


Figure 22.1: A directed graph representation of the *Loves* relation.

operator *OR*. For example, $\exists p_1$ *Loves*(*Sam*, p_1) ≡ [*Loves*(*Sam, Mary*) *OR* *Loves*(*Sam, Ann*) *OR Loves*(*Sam, Bob*) *OR*...].

The Universal Quantifier: The quantifier ∀ means that all of the objects in the domain have the property. It relates to the Boolean operator *AND*. For example, $\forall p_1$ *Loves*(*Sam*, p_1) ≡ [*Loves*(*Sam, Mary*) *AND Loves*(*Sam, Ann*) *AND* *Loves*(*Sam, Bob*) *AND*...].

Combining Quantifiers: Quantifiers can be combined. The order of operations is such that $\forall p_1 \exists p_2$ *Loves*(p_1, p_2) is understood to be bracketed as $\forall p_1 [\exists p_1$ *Loves*(p_1, p_2)], i.e., "Every person has the property 'he loves some other person'." It relates to the following Boolean formula:



Order of Quantifiers: The order of the quantifiers matters. For example, if b is the class of boys and g is the class of girls, $\forall b \exists g$ *Loves*(b, g) and $\exists g \forall b$ *Loves*(b, g) mean different things. The second one states that the same girl is loved by every boy. For it to be true, there needs to be a Marilyn Monroe sort of girl that all the boys love. The first statement says that every boy loves some girl. A Marilyn Monroe sort of girl will make this statement true. However, it is also true in a monogamous situation in which every boy loves a different girl. Hence, the first statement can be true in more different ways than the second one. In fact, the second statement implies the first one, but not vice versa.

Definition of Free and Bound Variables: The statement $\exists p_2$ *Loves*(*Sam*, p_2) means "Sam loves someone." This is a statement about Sam. Similarly, the statement $\exists p_2$ *Loves*(p_1, p_2) means "p_1 loves someone." This is a statement about person p_1. Whether the statement is true depends on who p_1 is referring to. The statement is *not* about p_2. The variable p_2 is used as a local variable (similar to $for(i = 1; i <= 10; i++)$) to express "someone." It could be a brother or a friend or a dog. In this expression, we say that the variable p_2 is *bound*, while p_1 is *free*, because p_2 has a quantifier and p_1 does not.

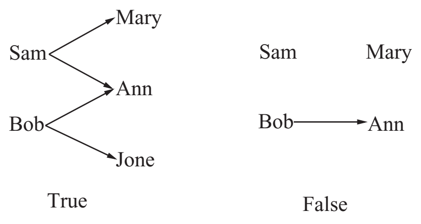


Figure 22.2: $\forall g \exists b \exists p$ ($Loves(b, g)$ and $Loves(b, p)$ and $g \neq p$). On the left is an example of a situation in which the statement is true, and on the right is one in which it is false.

Defining Other Relations: You can define other relations by giving an expression with free variables. For example, you can define the unary relation $LovesSomeone(p_1) \equiv \exists p_2\ Loves(p_1, p_2)$.

Building Expressions: Suppose you wanted to state that *every girl has been cheated on*, using the *Loves* relation. It may be helpful to break the problem into three steps.

Step 1. Assuming Other Relations: Suppose you have the relation $Cheats(Sam, Mary)$, indicating that Sam cheats on Mary. How would you express the statement that every girl has been cheated on? The advantage of using this function is that we can focus on this one part of the statement. We are not claiming that every boy cheats. One boy may have broken every girl's heart.

Given this, the answer is $\forall g \exists b\ Cheats(b, g)$.

Step 2. Constructing the Other Predicate: Here we do not have a *Cheats* function. Hence, we must construct a sentence from the *loves* function stating that Sam cheats on Mary.

Clearly, there must be someone else involved besides Mary, so let's start with $\exists p$. Now, in order for cheating to occur, who needs to love whom? (For simplicity's sake, let's assume that cheating means loving more than one person at the same time.) Certainly, Sam must love p. He must also love Mary. If he did not love her, then he would not be cheating on her. Must Mary love Sam? No. If Sam tells Mary he loves her dearly and then a moment later he tells Sue he loves *her* dearly, then he has cheated on Mary regardless of how Mary feels about him. Therefore, Mary does not have to love Sam. In conclusion, we might define $Cheats(Sam, Mary) \equiv \exists p\ (Loves(Sam, Mary)$ and $Loves(Sam, p))$. However, we have made a mistake here. In our example, the other person and Mary cannot be the same person. Hence, we must define the relation as $Cheats(Sam, Mary) \equiv \exists p\ (Loves(Sam, Mary)$ and $Loves(Sam, p)$ and $p \neq Mary)$.

Step 3. Combining the Parts: Combining the two relations together gives you $\forall g \exists b \exists p\ (Loves(b, g)$ and $Loves(b, p)$ and $p \neq g)$. This statement expresses that every girl has been cheated on. See Figure 22.2.

The Domain of a Variable: Whenever you state $\exists g$ or $\forall g$, there must be an understood set of values that the variable g might take on. This set is called the *domain* of the variable. It may be explicitly given or implied, but it must be understood. Here the domain is "the" set of girls. You must make clear whether this means all girls in the room, all the girls currently in the world, or all girls that have ever existed. For example,

$$\forall x \exists y\ x \times y = 1$$

states that every value has a reciprocal. It is certainly not true of the domain of integers, because two does not have an integer reciprocal. It seems to be true of the domain of reals. Be careful, however: zero does not have a reciprocal. It would be better to write

$$\forall x \neq 0,\ \exists y\ x \times y = 1$$

or equivalently

$$\forall x \exists y\ (x \times y = 1\ OR\ x = 0).$$

The Negation of a Statement: The negation of a statement is formed by putting a negation sign on the left-hand side. (Brackets sometimes help.) A negated statement, however, is best understood by moving the negation as deep (as far right) into the statement as possible. This is done as follows.

Negating *AND* and *OR*: A negation on the outside of an *AND* or an *OR* statement can be moved deeper into the statement using De Morgan's law. Recall that the *AND* is replaced by an *OR* and the *OR* is replaced with an *AND*.

¬(*Loves*(*S, M*) *AND Loves*(*S, A*)) iff ¬*Loves*(*S, M*) *OR* ¬*Loves*(*S, A*): The negation of "Sam loves Mary and Ann" is "Either Sam does not love Mary or he does not love Ann." He can love one of the girls, but not both.

A common mistake is to make the negation ¬*Loves*(*Sam, Mary*) *AND* ¬*Loves*(*Sam, Ann*). However, this says that Sam loves neither Mary nor Ann.

¬(*Loves*(*S, M*) *OR Loves*(*S,A*)) iff ¬*Loves*(*S, M*) *AND* ¬*Loves*(*S, A*): The negation of "Sam either loves Mary or he loves Ann" is "Sam does not love Mary and he does not love Ann."

Negating Quantifiers: Similarly, a negation can be moved past one or more quantifiers either to the right or to the left. However, you must then change these quantifiers from existential to universal and vice versa. Suppose d is the set of dogs. Then we have:

¬(∃*d Loves*(*Sam*, *d*)) iff ∀*d* ¬*Loves*(*Sam*, *d*): The negation of "There is a dog that Sam loves" is "There is no dog that Sam loves" or "All dogs are unloved by Sam." A common mistake is to state the negation as $\exists d\ \neg Loves(Sam, d)$. However, this says that "There is a dog that is not loved by Sam."

¬(∀*d Loves*(*Sam*, *d*)) iff ∃*d* ¬*Loves*(*Sam*, *d*): The negation of "Sam loves every dog" is "There is a dog that Sam does not love."

$\neg(\exists b \forall d\, Loves(b, d))$ **iff** $\forall b \neg(\forall d\, Loves(b, d))$ **iff** $\forall b \exists d \neg Loves(b, d)$**:** The negation of “There is a boy who loves every dog” is “There are no boys who love every dog” or “For every boy, it is not the case that he loves every dog” or “For every boy, there is some dog that he does not love.”

$\neg(\exists d_1 \exists d_2\ Loves(Sam, d_1)\ AND\ Loves(Sam, d_2)\ AND\ d_1 \neq d_2)$ **iff**
$\forall d_1 \forall d_2\ \neg(Loves(Sam, d_1)\ AND\ Loves(Sam, d_2)\ AND\ d_1 \neq d_2)$ **iff**
$\forall d_1 \forall d_2\ \neg Loves(Sam, d_1)\ OR\ \neg Loves(Sam, d_2)\ OR\ d_1 = d_2$: The negation of “There are two (distinct) dogs that Sam loves” is “Given any pair of (distinct) dogs, Sam does not love both” or “Given any pair of dogs, either Sam does not love the first or he does not love the second, or you gave me the same dog twice.”

The Domain Does Not Change: The negation of $\exists x \geq 5,\ x + 2 = 4$ is $\forall x \geq 5,\ x + 2 \neq 4$. The negation does *not* begin $\exists x < 5\ \ldots$. Both the statement and its negation are about numbers greater than 5. Is there or is there not a number with the property such that $x + 2 = 4$?

Proving a Statement True: There are a number of seemingly different techniques for proving that an existential or universal statement is true. The core of all these techniques, however, is the same. Personally, I like to view the proof as a strategy for winning a game against an adversary.

Techniques for Proving $\exists d\, Loves(Sam, d)$:

Proof by Example or by Construction: The classic technique to prove that something with a given property exists is by example. You either directly provide an example, or you describe how to construct such an object. Then you prove that your example has the property. For the above statement, the proof would state “Let d be Fido” and then would prove that Sam loves Fido.

Proof by Adversarial Game: Suppose you claim to an adversary that there is a dog that Sam loves. What will the adversary say? Clearly, he challenges, “Oh, yeah? What dog?” You then meet the challenge by producing a specific dog d and proving that $Loves(Sam, d)$, that is, that Sam loves d. The statement is true if you have a strategy guaranteed to beat any adversary in this game.

- If the statement is true, then you can produce some dog d.
- If the statement is false, then you will not be able to.

Techniques for Proving $\forall d\, Loves(Sam, d)$:

Proof by Example Does Not Work: Proving that Sam loves Fido is interesting, but it does not prove that he loves all dogs.

Proof by Case Analysis: The laborious way of proving that Sam loves all dogs is to consider each dog, one at a time, and prove that Sam loves it.

This method is impossible if the domain of dogs is infinite.

Proof by Arbitrary Example: The classic technique to prove that every object from some domain has a given property is to let some symbol represent an arbitrary object from the domain and then to prove that that object has the property. Here the proof would begin "Let d be any arbitrary dog." Because we don't actually know which dog d is, we must either (1) prove $Loves(Sam, d)$ simply from the properties that d has *because* d is a dog or (2) go back to doing a case analysis, considering each dog d separately.

Proof by Adversarial Game: Suppose you claim to an adversary that Sam loves every dog. What will the adversary say? Clearly he challenges, "Oh, yeah? What about Fido?" You meet the challenge by proving that Sam loves Fido. In other words, the adversary provides a dog d'. You win if you can prove that $Loves(Sam, d')$.

The only difference between this game and the one for existential quantifiers is who provides the example. Interestingly, the game only has one round. The adversary is only given one opportunity to challenge you.

A proof of the statement $\forall d\ Loves(Sam, d)$ consists of a strategy for winning the game. Such a strategy takes an arbitrary dog d', provided by the adversary, and proves that "Sam loves d'." Again, because we don't actually know *which* dog d' is, we must either (1) prove that $Loves(Sam, d')$ simply from the properties that d' has because he is a dog or (2) go back to doing a case analysis, considering each dog d' separately.

- If the statement $\forall d\ Loves(Sam, d)$ is true, then you have a strategy. No matter how the adversary plays, no matter which dog d' he gives you, Sam loves it. Hence, you can win the game by proving that $Loves(Sam, d')$.
- If the statement is false, then there is a dog d' that Sam does not love. Any true adversary (not just a friend) will produce this dog, and you will lose the game. Hence, you cannot have a winning strategy.

Proof by Contradiction: A classic technique for proving the statement $\forall d\ Loves(Sam, d)$ is proof by contradiction. Except in the way that it is expressed, it is exactly the same as the proof by an adversary game.

By way of contradiction assume that the statement is false, that is, $\exists d\ \neg Loves(Sam, d)$ is true. Let d' be some such dog that Sam does not love. Then you must prove that in fact Sam does love d'. This contradicts the statement that Sam does not love d'. Hence, the initial assumption is false, and $\forall d\ Loves(Sam, d)$ is true.

Proof by Adversarial Game for More Complex Statements: The advantage to this technique is that it generalizes into a nice game for arbitrarily long statements.

The Steps of the Game:

Left to Right: The game moves from left to right, providing an object for each quantifier.

Prover Provides $\exists b$: You, as the prover, must provide any existential objects.

Adversary Provides $\forall d$: The adversary provides any universal objects.

To Win, Prove the Relation $Loves(b', d')$**:** Once all the objects have been provided, you (the prover) must prove that the innermost relation is in fact true. If you can, then you win. Otherwise, you lose.

A Proof Is a Strategy: A proof of the statement consists of a strategy such that you win the game no matter how the adversary plays. For each possible move that the adversary takes, such a strategy must specify what move you will counter with.

Negations in Front: To prove a statement with a negation in the front of it, first put the statement into *standard* form with the negation moved to the right. Then prove the statement in the same way.

Examples:

$\exists b \forall d\, Loves(b,d)$**:** To prove "There is a boy that loves every dog," you must produce a specific boy b'. Then the adversary, knowing your boy b', tries to prove that $\forall d\ Loves(b', d)$ is false. He does this by providing an arbitrary dog d' that he hopes b' does not love. You must prove "b' loves d'."

$\neg(\exists b \forall d\, Loves(b,d))$ iff $\forall b \exists d \neg Loves(b, d)$**:** With the negation moved to the right, the first quantifier is universal. Hence, the adversary first produces a boy b'. Then, knowing the adversary's boy, you produce a dog d'. Finally, you prove that $\neg Loves(b', d')$.

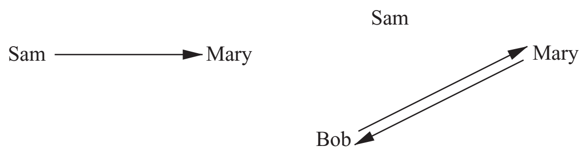
Your proof of the statement could be viewed as a function G that takes as input the boy b' given by the adversary and outputs the dog $d' = D(b')$ countered by you. Here, $d' = D(b')$ is an example of a dog that boy b' does not love. The proof must prove that $\forall b \neg Loves(b, D(b))$.

EXERCISE 22.0.1 *Let* $Loves(b, g)$ *denote that boy* b *loves girl* g*. If Sam loves Mary and Mary does not love Sam back, then we say that Sam loves in vain.*

1. *Express the following statements using universal and existential quantifiers. Move any negations to the right.*
   (a) *"Sam has loved in vain."*
   (b) *"There is a boy who has loved in vain."*

(c) *"Every boy has loved in vain."*
(d) *"No boy has loved in vain."*

2. *For each of the above statements and each of the two relations below, prove either that the statement is true for the relation or that it is false:*



EXERCISE 22.0.2 *(See solution in Part Five.) For each, prove whether true or not when each variable is a real value. Be sure to play the correct game as to who is providing what value:*

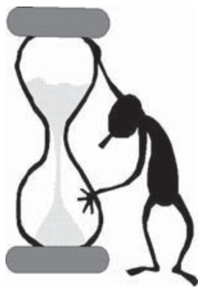
1. $\forall x\, \exists y\; x + y = 5$.
2. $\exists y\, \forall x\; x + y = 5$.
3. $\forall x\, \exists y\; x \cdot y = 5$.
4. $\exists y\, \forall x\; x \cdot y = 5$.
5. $\forall x\, \exists y\; x \cdot y = 0$.
6. $\exists y\, \forall x\; x \cdot y = 0$.
7. $[\forall x\, \exists y\; P(x, y)] \Rightarrow [\exists y\, \forall x\; P(x, y)]$.
8. $[\forall x\, \exists y\; P(x, y)] \Leftarrow [\exists y\, \forall x\; P(x, y)]$.
9. $\forall a\, \exists y\, \forall x\; x \cdot (y + a) = 0$.
10. $\exists a\, \forall x\, \exists y\; [x = 0 \text{ or } x \cdot y = 5]$.

EXERCISE 22.0.3 *The game ping has two rounds. Player A goes first. Let m_1^A denote his first move. Player B goes next. Let m_1^B denote his move. Then player A goes m_2^A, and player B goes m_2^B. The relation $AWins(m_1^A, m_1^B, m_2^A, m_2^B)$ is true iff player A wins with these moves.*

1. *Use universal and existential quantifiers to express the fact that player A has a strategy with which he wins no matter what player B does. Use m_1^A, m_1^B, m_2^A, m_2^B as variables.*
2. *What steps are required in the prover–adversary technique to prove this statement?*
3. *What is the negation of the above statement in standard form?*
4. *What steps are required in the prover–adversary technique to prove this negated statement?*

EXERCISE 22.0.4 *Why does $[\forall n_0,\ \exists n > n_0,\ P(n)]$ imply that there are an infinite number of values n for which the property $P(n)$ is true?*

23 Time Complexity



It is important to classify algorithms based whether they solve a given computational problem and, if so, how quickly. Similarly, it is important to classify computational problems based whether they can be solved and, if so, how quickly.

23.1 The Time (and Space) Complexity of an Algorithm

Purpose:

Estimate Duration: To estimate how long an algorithm or program will run.

Estimate Input Size: To estimate the largest input that can reasonably be given to the program.

Compare Algorithms: To compare the efficiency of different algorithms for solving the same problem.

Parts of Code: To help you focus your attention on the parts of the code that are executed the largest number of times. This is the code you need to improve to reduce the running time.

Choose Algorithm: To choose an algorithm for an application:

- If the input size won't be larger than six, don't waste your time writing an extremely efficient algorithm.
- If the input size is a thousand, then be sure the program runs in polynomial, not exponential, time.
- If you are working on the Gnome project and the input size is a billion, then be sure the program runs in linear time.

Time and Space Complexities Are Functions, $T(n)$ and $S(n)$: The time complexity of an algorithm is not a single number, but is a function indicating how the running time depends on the size of the input. We often denote this by $T(n)$, giving the number of operations executed on the worst case input instance of size n. An example would be $T(n) = 3n^2 + 7n + 23$. Similarly, $S(n)$ gives the size of the rewritable memory the algorithm requires.

Ignoring Details, $\Theta(T(n))$ and $O(T(n))$: Generally, we ignore the low-order terms in the function $T(n)$ and the multiplicative constant in front. We also ignore the function for small values of n and focus on the *asymptotic* behavior as n becomes very large. Some of the reasons are the following.

> **Model-Dependent:** The multiplicative constant in front of the time depends on how fast the computer is and on the precise definition of "size" and "operation."
>
> **Too Much Work:** Counting every operation that the algorithm executes in precise detail is more work than it is worth.
>
> **Not Significant:** It is much more significant whether the time complexity is $T(n) = n^2$ or $T(n) = n^3$ than whether it is $T(n) = n^2$ or $T(n) = 3n^2$.
>
> **Only Large *n* Matter:** One might say that we only consider large input instances in our analysis, because the running time of an algorithm only becomes an issue when the input is large. However, the running time of some algorithms on small input instances is quite critical. In fact, the size n of a realistic input instance depends both on the problem and on the application. The choice was made to consider only large n in order to provide a clean and consistent mathematical definition.

See Chapter 25 on the Theta and BigOh notations.

Definition of Size: The formal definition of the size of an instance is the number of binary digits (*bits*) required to encode it. More practically, the size could be considered to be the number of *digits* or *characters* required to encode it. Intuitively, the size of an instance could be defined to be the area of paper needed to write down the instance, or the number of seconds it takes to communicate the instance along a narrow channel. These definitions are all within a multiplicative constant of each other.

> **An Integer:** Suppose that the input is the value $N = 8{,}398{,}346{,}386{,}236{,}876$. The number of bits required to encode it is $Size(N) = \log_2(N) = \log_2(8{,}398{,}346{,}386{,}236{,}876) = 53$, and the number of decimal digits is $Size(N) = \log_{10}(8{,}398{,}346{,}386{,}236{,}876) = 16$. Chapter 24 explains why these are within a multiplicative constant of each other. The one definition that you must *not* use is the *value*

of the integer, $Size(N) = N = 8{,}398{,}346{,}386{,}236{,}876$, because it is exponentially different than $Size(N) = \log_2(N) = 53$.

A Tuple: Suppose that the input is the tuple of b integers $I = \langle x_1, x_2, \ldots, x_b\rangle$. The number of bits required to encode it is $Size(I) = \log_2(x_1) + \log_2(x_2) + \cdots + \log_2(x_b) \approx \log_2(x_i) \cdot b$. A natural definition of the size of this tuple is the *number of integers* in it, $Size(I) = b$. With this definition, it is a much stronger statement to say that an algorithm requires only $Time(b)$ integer operations independent of how big the integers are.

A Graph: Suppose that the input is the graph $G = \langle V, E\rangle$ with $|V|$ nodes and $|E|$ edges. The number of bits required to encode it is $Size(G) = 2\ Size(node) \cdot |E| = 2\log_2(|V|) \cdot |E|$. Another reasonable definition of the size of G is the *number of edges*, $G(n) = |E|$. Often the time is given as a function of both $|V|$ and $|E|$. This is within a log factor of the other definitions, which for most applications is fine.

Definition of an Operation: The definition of an operation can be any reasonable operation on two bits, characters, nodes, or integers, depending on whether time is measured in bits, characters, nodes, or integers. An operation could also be defined to be any reasonable line of code or the number of seconds that the computation takes on your favorite computer.

Which Input: $T(n)$ is the number of operations required to execute the given algorithm on an input of size n. However, there are 2^n input instances with n bits. Here are three possibilities:

A Typical Input: The problem with considering a typical input instance this is that different applications will have very different typical inputs.

Average or Expected Case: The problem with taking the average over all input instances of size n is that it assumes that all instances are equally likely to occur.

Worst Case: The usual measure is to consider the instance of size n on which the given algorithm is the slowest, namely, $T(n) = \max_{I\in\{I \mid |I|=n\}} Time(I)$. This measure provides a nice clean mathematical definition and is the easiest to analyze. The only problem is that sometimes the algorithm does much better than the worst case, because the worst case is not a reasonable input. One such algorithm is quick sort (see Section 9.1).

Time Complexity of a Problem: The time complexity of a problem is the running time of the fastest algorithm that solves the problem.

EXAMPLE 23.1 ***Polynomial Time vs Exponential Time***

Suppose program P_1 requires $T_1(n) = n^4$ operations and P_2 requires $T_2(n) = 2^n$. Suppose that your machine executes 10^6 operations per second. If $n = 1{,}000$, what is the running time of these programs?

Answer:

1. $T_1(n) = (1{,}000)^4 = 10^{12}$ operations, requiring 10^6 seconds, or 11.6 days.
2. $T_2(n) = 2^{(10^3)}$ operations. The number of years is $\frac{2^{(10^3)}}{10^6 \times 60 \times 60 \times 24 \times 365}$. This is too big for my calculator. The log of this number is $10^3 \times \log_{10}(2) - \log_{10} 10^6 - \log_{10}(60 \times 60 \times 24 \times 365) = 301.03 - 6 - 7.50 = 287.53$. Therefore, the number of years is $10^{287.53} = 3.40 \times 10^{287}$. Don't wait for it.

EXAMPLE 23.2 ***Instance Size N vs Instance Value N***

Two simple algorithms, summation and factoring.

The Problems and Algorithms:

Summation: The task is to sum the N entries of an array, that is, $A(1) + A(2) + A(3) + \cdots + A(N)$.

Factoring: The task is to find divisors of an integer N. For example, on input $N = 5917$ we output that $N = 97 \times 61$. (This problem is central to cryptography.) The algorithm checks whether N is divisible by 2, by 3, by 4, ... by N.

Time: Both algorithms require $T = N$ operations (additions or divisions).

How Hard? The summation algorithm is considered to be very fast, while the factoring algorithm is considered to be very time-consuming. However, both algorithms take $T = N$ time to complete. The time complexity of these algorithms will explain why.

Typical Values of N: In practice, the N for factoring is much larger than that for summation. Even if you sum all the entries on the entire 8-G byte hard drive, then N is still only $N \approx 10^{10}$. On the other hand, the military wants to factor integers $N \approx 10^{100}$. However, the measure of complexity of an algorithm should not be based on how it happens to be used in practice.

Size of the Input: The input for summation contains is $n \approx 32N$ bits. The input for factoring contains is $n = \log_2 N$ bits. Therefore, with a few hundred bits you can write down a difficult factoring instance that is seemingly impossible to solve.

Time Complexity: The running time of summation is $T(n) = N = \frac{1}{32}n$, which is linear in its input size. The running time of factoring is $T(N) = N = 2^n$, which is exponential in its input size. This is why the summation algorithm is considered to be *feasible*, while the factoring algorithm is considered to be *infeasible*.

EXERCISE 23.1.1 *(See solution in Part Five.) For each of the two programs considered in Example 23.1, if you want it to complete in 24 hours, how big can your input be?*

EXERCISE 23.1.2 *In Example 23.1, for which input size, approximately, do the programs have the same running times?*

EXERCISE 23.1.3 *This problem compares the running times of the following two algorithms for multiplying:*

```
algorithm KindergartenAdd($a$, $b$)
⟨pre-cond⟩: $a$ and $b$ are integers.
⟨post-cond⟩: Outputs $a \times b$.
begin
    $c = 0$
    loop $i = 1 \ldots a$
        $c = c + b$
    end loop
    return(c)
end algorithm
```

```
algorithm GradeSchoolAdd($a$, $b$)
⟨pre-cond⟩: $a$ and $b$ are integers.
⟨post-cond⟩: Outputs $a \times b$.
begin
    Let $a_{s-1} \ldots a_3a_2a_1a_0$ be the decimal digits of $a$, so that $a = \sum_{i=0}^{s-1} a_i \times 10^i$.
    Let $b_{t-1} \ldots b_3b_2b_1b_0$ be the decimal digits of $b$, so that $b = \sum_{j=0}^{t-1} b_j \times 10^j$.
    $c = 0$
    loop $i = 1 \ldots s$
        loop $j = 1 \ldots t$
            $c = c + a_ib_j \times 10^{i+j}$.
        end loop
    end loop
    return(c)
end algorithm
```

For each of these algorithms, answer the following questions.

1. *Suppose that each addition to c requires time 10 seconds and every other operation (for example, multiplying two single digits such as 9×8 and shifting by zero) is free. What is the running time of each of these algorithms, either as a function of a and b or as a function of s and t? Give everything for this entire question exactly, i.e., not BigOh.*

2. *Let $a = 9{,}168{,}391$ and $b = 502$. (Without handing it in, trace the algorithm.) With 10 seconds per addition, how much time (seconds, minutes, etc.) does the computation require?*
3. *The formal size of an input instance is the number n of bits needed to write it down. What is n as a function of our instance $\langle a, b\rangle$?*
4. *Suppose your job is to choose the worst case instance $\langle a, b\rangle$ (i.e., the one that maximizes the running time), but you are limited in that you can only use n bits to represent your instance. Do you set a big and b small, a small and b big, or a and b the same size? Give the worst case a and b, or s and t, as a function of n.*
5. *The running time of an algorithm is formally defined to be a function $T(n)$ from n to the time required for the computation on the worst case instance of size n. Give $T(n)$ for each of these algorithms. Is this polynomial time?*

EXERCISE 23.1.4 *(See solution in Part Five.) Suppose that someone has developed an algorithm to solve a certain problem, which runs in time $T(n, k) \in \Theta(f(n, k))$, where n is the size of the input, and k is a parameter we are free to choose (we can choose it to depend on n). In each case determine the value of the parameter $k(n)$ to achieve the (asymptotically) best running time. Justify your answer. I recommend not trying much fancy math. Think of n as being some big fixed number. Try some value of k, say $k = 1$, $k = n^a$, or $k = 2^{an}$ for some constant a. Then note whether increasing or decreasing k increases or decreases f. Recall that "asymptotically" means that we only need the minimum to within a multiplicative constant.*

1. *You might want to first prove that $g + h = \Theta(\max(g, h))$.*
2. *$f(n, k) = \frac{n+k}{\log k}$. This is needed for the radix–counting sort in Section 5.4.*
3. *$f(n, k) = \frac{n^3}{k} + k \cdot n$.*
4. *$f(n, k) = \log^3 k + \frac{2^n}{k}$.*
5. *$f(n, k) = \frac{8^n n^2}{k} + k \cdot 2^n + k^2$.*

23.2 The Time Complexity of a Computational Problem

The Formal Definition of the Time Complexity of a Problem: As said, the time complexity of a problem is the running time of the fastest algorithm that solves the problem. We will now define this more carefully, using the existential and universal quantifiers that were defined in Chapter 22.

The Time Complexity of a Problem: The time complexity of a computational problem P is the minimum time needed by an algorithm to solve it.

Upper Bound: Problem P is said to be computable in time $T_{upper}(n)$ if there is an algorithm A that outputs the correct answer, namely $A(I) = P(I)$, within the bounded time, namely $Time(A, I) \leq T_{upper}(|I|)$, on every input instance I. The formal statement is

$$\exists A,\ \forall I,\ [A(I) = P(I) \text{ and } Time(A, I) \leq T_{upper}(|I|)]$$

$T_{upper}(n)$ is said to be only an *upper bound* on the complexity of the problem P, because there may be another algorithm that runs faster. For example, $P = Sorting$ is computable in $T_{upper}(n) = O(n^2)$ time. It is also computable in $T_{upper}(n) = O(n \log n)$.

Lower Bound of a Problem: A lower bound on the time needed to solve a problem states that no matter how smart you are, you cannot solve the problem faster than the stated time $T_{lower}(n)$, because such algorithm simply does not exist. There may be algorithms that give the correct answer or run sufficiently quickly on some input instances. But for every algorithm, there is at least one instance I for which either the algorithm gives the wrong answer, i.e., $A(I) \neq P(I)$, or it takes too much time, i.e., $Time(A, I) \geq T_{lower}(|I|)$. The formal statement is the negation (except for $\geq$ vs $>$) of that for the upper bound:

$$\forall A,\ \exists I,\ [A(I) \neq P(I) \text{ or } Time(A, I) \geq T_{lower}(|I|)]$$

For example, it should be clear that no algorithm can sort n values in only $T_{lower} = \sqrt{n}$ time, because in that much time the algorithm could not even look at all the values.

Proofs Using the Prover–Adversary Game: Recall the technique described in Chapter 22 for proving statements with existential and universal quantifiers.

Upper Bound: We can use the prover–adversary game to prove the upper bound statement $\exists A,\ \forall I,\ [A(I) = P(I) \text{ and } Time(A, I) \leq T_{upper}(|I|)]$ as follows: You, the prover, provide the algorithm A. Then the adversary provides an input I. Then you must prove that your A on input I gives the correct output in the allotted time. Note this is what we have been doing throughout the book: providing algorithms and proving that they work.

Lower Bound: A proof of the lower bound $\forall A,\ \exists I, [A(I) \neq\ P(I) \text{ or } Time(A, I) \geq T_{lower}(|I|)]$ consists of a strategy that, when given an algorithm A by an adversary, you, the prover, study his algorithm and provide an input I. Then you prove either that his A on input I gives the wrong output or that it runs in more than the allotted time.

EXERCISE 23.2.1 *(See solution in Part Five.) Let $Works(P, A, I)$ to true if algorithm A halts and correctly solves problem P on input instance I. Let $P = Halting$ be the halting problem that takes a Java program I as input and tells you whether or not it halts on the empty string. Let $P = Sorting$ be the sorting problem that takes a list of* *numbers I as input and sorts them. For each part, explain the meaning of what you are doing and why you don't do it another way.*

1. *Recall that a problem is* computable *if and only if there is an algorithm that halts and returns the correct solution on every valid input. Express in first-order logic that Sorting is computable.*
2. *Express in first-order logic that Halting is not computable.*
3. *Express in first-order logic that there are uncomputable problems.*
4. *Explain what the following means (not simply by saying the same in words), and either prove or disprove it:* $\forall I, \exists A, Works(Halting, A, I)$.
5. *Explain what the following means, and either prove or disprove it:* $\forall A, \exists P, \forall I, Works(P, A, I)$. *(Hint: An algorithm A on an input I can either halt and give the correct answer, halt and give the wrong answer, or run forever.)*

24 Logarithms and Exponentials

Logarithms $\log_2(n)$ and exponentials 2^n arise often when analyzing algorithms.

Uses: These are some of the places that you will see them.

Divide a Logarithmic Number of Times: Many algorithms repeatedly cut the input instance in half. A classic example is binary search (Section 1.4): You take something of size n and you cut it in half, then you cut one of these halves in half, and one of these in half, and so on. Even for a very large initial object, it does not take very long until you get a piece of size below 1. The number of divisions required is about $\log_2(n)$. Here the base 2 is because you are cutting them in half. If you were to cut them into thirds, then the number of times to cut would be about $\log_3(n)$.

A Logarithmic Number of Digits: Logarithms are also useful because writing down a given integer value n requires $\lceil \log_{10}(n+1) \rceil$ decimal digits. For example, suppose that $n = 1{,}000{,}000 = 10^6$. You would have to divide this number by 10 six times to get to 1. Hence, by definition, $\log_{10}(n) = 6$. This, however, is the number of zeros, not the number of digits. We forgot the leading digit 1. The formula $\lceil \log_{10}(n+1) \rceil = 7$ does the trick. For the value $n = 6{,}372{,}845$, the number of digits is given by $\log_{10}(6{,}372{,}846) = 6.804333$, rounded up to 7. Being in computer science, we store our values using bits. Similar arguments give that $\lceil \log_2(n+1) \rceil$ is the number of bits needed.

Height and Size of Binary Tree: A complete balanced binary tree of height h has 2^h leaves and $n = 2^{h+1} - 1$ nodes. Conversely, if it has n nodes, then its height is $h \approx \log_2 n$.

Exponential Search: Suppose a solution to your problem is represented by n digits. There are 10^n such strings of n digits. Doing a blind search through them all would take too much time.

Rules: There are lots of rules about logs and exponentials that one might learn. Personally, I like to confine them to the following:

$\boldsymbol{b^n = \overbrace{b \times b \times b \times \cdots \times b}^{n}}$**:** This is the definition of exponentiation. b^n is n b's multiplied together.

$\boldsymbol{b^n \times b^m = b^{n+m}}$**:** This is proved simply by counting the number of b's being multiplied:

$$(\overbrace{b \times b \times b \times \cdots \times b}^{n}) \times (\overbrace{b \times b \times b \times \cdots \times b}^{m}) = \overbrace{b \times b \times b \times \cdots \times b}^{n+m}.$$

$\boldsymbol{b^0 = 1}$**:** One might guess that zero b's multiplied together is zero, but it needs to be one. One argument for this is as follows. $b^n = b^{0+n} = b^0 \times b^n$. For this to be true, b^0 must be one.

$\boldsymbol{b^{\frac{1}{2}} = \sqrt{n}}$**:** By definition, $\sqrt{n}$ is the positive number that when multiplied by itself gives n. $b^{\frac{1}{2}}$ meets this definition because $b^{\frac{1}{2}} \times b^{\frac{1}{2}} = b^{\frac{1}{2}+\frac{1}{2}} = b^1 = b$.

$\boldsymbol{b^{-n} = 1/b^n}$**:** The fact that this needs to be true can be argued in a similar way. $1 = b^{n+(-n)} = b^n \times b^{-n}$. For this to be true, b^{-n} must be $1/b^n$.

$\boldsymbol{(b^n)^m = b^{n \times m}}$**:** Again we count the number of b's:

$$\overbrace{(\overbrace{b \times b \times b \times \cdots \times b}^{n}) \times (\overbrace{b \times b \times b \times \cdots \times b}^{n}) \times \cdots \times (\overbrace{b \times b \times b \times \cdots \times b}^{n})}^{m}$$
$$= \overbrace{b \times b \times b \times \cdots \times b}^{n \times m}.$$

If $\boldsymbol{x = \log_b(n)}$ then $\boldsymbol{n = b^x}$: This is the definition of logarithms.

$\boldsymbol{\log_b(1) = 0}$**:** This follows from $b^0 = 1$.

$\boldsymbol{\log_b(b^x) = x}$ **and** $\boldsymbol{b^{\log_b(n)} = n}$**:** Substituting $n = b^x$ into $x = \log_b(n)$ gives the first, and substituting $x = \log_b(n)$ into $n = b^x$ gives the second.

$\boldsymbol{\log_b(n \times m) = \log_b(n) + \log_b(m)}$**:** The number of digits to write down the product of two integers is the number to write down each of them separately (up to rounding errors). We prove it by applying the definition of logarithms and the above rules: $b^{\log_b(n \times m)} = n \times m = b^{\log_b(n)} \times b^{\log_b(m)} = b^{\log_b(n)+\log_b(m)}$. It follows that $\log_b(n \times m) = \log_b(n) + \log_b(m)$.

$\boldsymbol{\log_b(n^d) = d \times \log_b(n)}$**:** This is an extension of the above rule.

$\boldsymbol{\log_b(n) - \log_b(m) = \log_b(n) + \log_b(\frac{1}{m}) = \log_b(\frac{n}{m})}$**:** This is another extension of the above rule.

$\boldsymbol{d^{c \log_2(n)} = n^{c \log_2(d)}}$**:** This rule states that you can move things between the base and the exponent as long as you insert or remove a log. The proof is as follows.

$d^{c\log_2(n)} = (2^{\log_2(d)})^{c\log_2(n)} = 2^{\log_2(d)\times c\log_2(n)} = 2^{\log_2(n)\times c\log_2(d)} = (2^{\log_2(n)})^{c\log_2(d)} = n^{c\log_2(d)}$.

$\log_2(n) = 3.32\ldots \times \log_{10}(n)$: The number of bits needed to express the integer n is 3.32... times the number of decimal digits needed. This can be seen as follows. Suppose $x = \log_2 n$. Then $n = 2^x$, giving $\log_{10} n = \log_{10}(2^x) = x \cdot \log_{10} 2$. Finally,

$$x = \frac{1}{\log_{10} 2} \log_{10}(n) = 3.32\ldots \log_{10} n$$

Which Base: We will write $\Theta(\log(n))$ without giving an explicit base. A high school student might use base 10 as the default, a scientist base $e = 2.718\ldots$, and a computer scientist base 2. My policy is to exclude the base when it does not matter. As seen above, $\log_{10}(n)$, $\log_2(n)$, and $\log_e(n)$ differ only by multiplicative constants. In general, we ignore multiplicative constants, and hence the base used is irrelevant. I only include the base when the base matters. For example, 2^n and 10^n differ by much more than a multiplicative constant.

The Ratio $\frac{\log a}{\log b}$: When computing the ratio between two logarithms, the base used does not matter, because changing the base will introduce the same constant on both the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose whichever base makes the calculation the easiest. For example, to compute $\frac{\log 16}{\log 8}$, the obvious base to use is 2, because $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$. On the other hand, to compute $\frac{\log 9}{\log 27}$, the obvious base to use is 3, because $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$.

EXERCISE 24.0.1 *(See solution in Part Five.) Simplify the following exponentials:* $a^3 \times a^5$, $3^a \times 5^a$, $3^a + 5^a$, $2^{6\log_4 n+7}$, $n^{3/\log_2 n}$.

25 Asymptotic Growth

Classes of Growth Rates: It is important to be able to classify functions $f(n)$ based on how quickly they grow: The following table outlines the few easy rules with which to classify functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^{en})$.

c	b^a	d	e	Class	Θ	Examples
> 0	> 1	Any	Any	**Exponentials**	$2^{\Theta(n)}$	2^n, $\frac{3^{0.001n}}{n^{100}}$
	$= 1$	> 0	Any	**Polynomials:**	$n^{\Theta(1)}$	n^4, $\frac{5n^{0.0001}}{\log^{100}(n)}$
		$= 2$	Any	• **Quadratic**	$\Theta(n^2)$	$5n^2, 2n^2 + 7n + 8$
		$= 1$	$= 1$	• **Sorting time**	$\Theta(n \log n)$	$5n \log n + 3n$
		$= 1$	$= 0$	• **Linear**	$\Theta(n)$	$5n + 3$
		$= 0$	> 0	**Polylogarithms:**	$\log^{\Theta(1)}(n)$	$5 \log^3(n)$
			$= 1$	• **Logarithms**	$\Theta(\log n)$	$5 \log(n)$
			$= 0$	**Constants**	$\Theta(1)$	$5, 5 + \sin n$
		< 0	Any	**Decreasing polynomials**	$\frac{1}{n^{\Theta(1)}}$	$\frac{1}{n^4}$, $\frac{5\log^{100}(n)}{n^{0.0001}}$
	< 1	Any	Any	**Decreasing exponentials**	$\frac{1}{2^{\Theta(1)}}$	$\frac{1}{2^n}$, $\frac{n^{100}}{3^{0.001n}}$

Asymptotic Notation: When we want to bound the growth of a function while ignoring multiplicative constants, we use the following notation:

Name	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = O(g(n))$	$f(n) \leq O(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$

Purpose:

Time Complexity: Generally, the functions that we will be classifying will be the time or space complexities of programs. On the other hand, these ideas can also be used to classify any function.

Function Growth: The purpose of classifying a function is to give an idea of how fast it grows without going into too much detail.

Asymptotic Growth Rate: When classifying animals, Darwin decided not to consider whether the animal sleeps during the day, but to consider whether it has hair. When classifying functions, complexity theorists decided to not consider its behavior for small values of n or even whether it is monotone increasing, but how quickly it grows when its input n grows really big. This is referred to as the *asymptotics* of the function. Here are some examples of different growth rates:

Function	**Approximate value of $T(n)$ for $n=$**				
$T(n)$	**10**	**100**	**1,000**	**10,000**	**Animal**
5	5	5	5	5	Virus
$\log_2 n$	3	6	9	13	Amoeba
$\sqrt{n}$	3	10	31	100	Bird
n	10	100	1,000	10,000	Human
$n \log n$	30	600	9,000	130,000	Giant
n^2	100	10,000	10^6	10^8	Elephant
n^3	1,000	10^6	10^9	10^{12}	Dinosaur
2^n	1,024	10^{30}	10^{300}	10^{3000}	The universe

Note: The universe contains approximately 10^{80} particles.

Exponential vs Polynomial: The table shows that an exponential function like $f(n) = 2^n$ grows extremely quickly. In fact, for sufficiently big n, this exponential 2^n grows much faster than any polynomial, even $n^{1,000,000}$. To take this to an extreme, the function $f(n) = 2^{0.001n}$ is also an exponential. It too grows much faster than $n^{1,000,000}$ for sufficiently large n.

Polynomial vs Logarithmic: The table also shows that a logarithmic function like $f(n) = \log_2 n$ grows, but very slowly. Hence, for sufficiently large n, it is bigger than any constant, but smaller than any polynomial.

EXERCISE 25.0.1 *Give a value of n for which* $n^{1,000,000} < 10^n$.
Give a value of n for which $n^{1,000} < 10^{0.001n}$.

EXERCISE 25.0.2 *Give a value of n for which* $(\log_{10} n)^{1,000,000} < n$.
Give a value of n for which $(\log_{10} n)^{1,000} < n^{0.001}$.

25.1 Steps to Classify a Function

Given a function $f(n)$, we will classify it according to its growth using the following steps.

1) Put $f(n)$ into Basic Form: Though there are strange functions out there, most functions $f(n)$ can be put into a basic form consisting of the sum of a number of terms, where each term has the basic form $c \cdot b^{an} \cdot n^d \cdot (\log n)^e$, where a, b, c, d, and e are real constants.

Examples:

- If $f(n) = 3 \cdot 2^{4n} \cdot n^7 \cdot (\log n)^5$, then $a = 4$, $b = 2$, $c = 3$, $d = 7$, and $e = 5$.
- Suppose $f(n) = n^2$. This has no exponential part b^{an}, but can be viewed as having $a = 0$, or $b = 1$, or $a^b = 1$. (Recall $x^0 = 1$ and $1^x = 1$.) The exponent on the polynomial n is $d = 2$. There is no logarithmic factor, so we have $e = 0$. Finally, the constant in front is $c = 1$.
- In $f(n) = 1/n^6 = n^{-6}$, it is also useful to see that $d = -6$.
- If $f(n) = n^2/\log n + 5$, then the function has two terms. In the first, $a^b = 1$, $c = 1$, $d = 2$, and $e = -1$. In the second, $a^b = 1$, $c = 5$, $d = 0$, and $e = 0$.

2) Get the Big-Picture Growth: We classify the set of all vertebrate animals into mammals, birds, reptiles, and so on. Similarly, we will classify functions into the major groups exponentials $2^{\Theta(n)}$, polynomials $n^{\Theta(1)}$, polylogarithms $\log^{\Theta(1)}(n)$, and constants $\Theta(1)$.

Exponentials $2^{\Theta(n)}$: If the function $f(n)$ is the sum of a bunch of things, one of which is $c \cdot b^{an} \cdot n^d \cdot (\log n)^e$, where $b^a > 1$, then $f(n)$ is considered to be an exponential.

Examples Included:

- $f(n) = 2^n$ and $f(n) = 3^{5n}$
- $f(n) = 2^n \cdot n^2 \log_2 n - 7n^8$ and $f(n) = \frac{2^n}{n^2}$

Examples Not Included:

- $f(n) = 1^n = 2^{0 \cdot n} = 1$, $f(n) = 2^{-1 \cdot n} = \left(\frac{1}{2}\right)^n$, and $f(n) = n^{1,000,000}$ (too small)
- $f(n) = n! \approx n^n = 2^{n \log_2 n}$ and $f(n) = 2^{n^2}$ (too big)

Definition of an Infeasible Algorithm: An algorithm is considered to be *infeasible* if it runs in exponential time. This is because such functions grow extremely quickly as n gets larger.

$(b^a)^n$: We require $b^a > 1$ because $b^{an} = (b^a)^n$, which grows as long as the base b^a is at least one.

The Notation $2^{\Theta(n)}$: We will see later that $\Theta(1)$ denotes any constant greater than zero. The notation $2^{\Theta(n)} = 2^{\Theta(1) \cdot n}$ is used to represent the class of exponentials, because $b^{an} = 2^{(a \log_2 b) \cdot n}$ and the constant $a \log_2 b = log_2(b^a)$ is greater than zero as long as b^a is greater than 1. Recall $\log_2 1 = 0$.

Bounded Between: By these rules, $f(n) = c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ is exponential if $b^a > 1$, no matter what the constants c, d, e, and f are. Consider $f(n) = 2^n/n^{100}$. The rule states that it is an exponential because $b^a = 2^1 > 1$ and $d = -100$. We might question this, thinking that dividing by n^{100} would not let it grow faster enough to be considered to be an exponential. We see that it does grow fast enough by proving that it is bounded between the two exponential functions $2^{0.5n}$ and 2^n.

Polynomial $n^{\Theta(1)}$: If $f(n) = c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ is such that $b^a = 1$, then we can ignore b^{an}, giving $f(n) = c \cdot n^d \cdot (\log n)^e$. If $d > 0$, then the function $f(n)$ is considered to be a polynomial.

Examples Included:

- $f(n) = 3n^2$ and $f(n) = 7n^2 - 8n \log n + 2n - 17$
- $f(n) = \sqrt{n} = n^{1/2}$ and $f(n) = n^{3.1}$
- $f(n) = n^2 \log_2 n$ and $f(n) = \frac{n^2}{\log_2 n}$
- $f(n) = 7n^3 \log^7 n - 8n^2 \log n + 2n - 17$

Examples Not Included:

- $f(n) = n^0 = 1$, $f(n) = n^{-1} = \frac{1}{n}$, and $f(n) = \log n$ (too small)
- $f(n) = n^{\log n}$ and $f(n) = 2^n$ (too big)

Definition of a Feasible Algorithm: An algorithm is considered to be *feasible* if it runs in polynomial time. (This is not actually true if $f(n) = n^{1,000,000}$.)

The Notation $n^{\Theta(1)}$: $\Theta(1)$ denotes any constant greater than zero, and hence $n^{\Theta(1)}$ represents any function $f(n) = n^d$ where $d > 0$.

Bounded Between: Though it would not be considered one in a mathematical study of polynomials, we also consider $f(n) = 3n^2 \log n$ to be a polynomial, because it is bounded between n^2 and n^3, which clearly are polynomials.

Polylogarithms $\log^{\Theta(1)}(n)$: Powers of logs like $(\log n)^3$ are referred to as *polylogarithms*. These are often written as $\log^3 n = (\log n)^3$. This is different than $\log(n^3) = 3 \log n$.

Example Included:

- $f(n) = 7(\log_2 n)^5$, $f(n) = 7 \log_2 n$, and $f(n) = 7\sqrt{\log_2 n}$
- $f(n) = 7(\log_2 n)^5 + 6(\log_2 n)^3 - 19 + 7(\log_2 n)^2/n$

Example Not Included:

- $f(n) = n$ (too big)

Constants $\Theta(1)$: A constant function is one whose output does not depend on its input, for example, $f(n) = 7$. One algorithm for which this function arises is popping an element off a stack that is stored as a linked list. This takes maybe seven operations, independent of the number n of elements in the stack.

The Notation $n^{\Theta(1)}$: We use the notation $\Theta(1)$ to replace any constant when we do not care what the actual constant is because determining whether it is 7, 9, or 8.829 may be more work and more detail than we need. On the other hand, in most applications being negative [$f(n) = -1$] or zero [$f(n) = 0$] would be quite a different matter. Hence, these are excluded.

Bounded Between: A function like $f(n) = 8 + \sin n$ changes continuously between 7 and 9, and $f(n) = 8 + \frac{1}{n}$ changes continuously on approaching 8. However, if we don't care whether it is 7, 9, or 8.829, why should we care if it is changing between them? Hence, both of these functions are included in $\Theta(1)$. On the other hand, the function $f(n) = \frac{1}{n}$ is not included, because the only constant that it is bounded below by is zero and the zero function is not included.

Examples Included:

- $f(n) = 7$ and $f(n) = 8.829$
- $f(n) = 8 + \sin n$, $f(n) = 8 + \frac{1}{n}$

Examples Not Included:

- $f(n) = -1$ and $f(n) = 0$ (fails $c > 0$)
- $f(n) = \sin n$ (fails $c > 0$)
- $f(n) = \frac{1}{n}$ (too small)
- $f(n) = \log_2 n$ (too big)

3) Determine $\Theta(f(n))$: We further classify mammals into humans, cats, dogs, and so on. Similarly, we further classify the polynomials $n^{\Theta(1)}$ into linear functions $\Theta(n)$, the time for sorting $\Theta(n \log n)$, quadratics $\Theta(n^2)$, and so on. These are classes that ignore the multiplicative constant.

Steps: One "takes the Theta" of a function $f(n)$ by dropping the low-order terms and then dropping the multiplicative constant c in front of the largest term.

Dropping Low-Order Terms: If $f(n)$ is a set of things added or subtracted together, then each of these things is called a *term*. We determine which of the terms grows fastest as n gets large. The slower-growing terms are referred to as *low-order terms*. We drop them because they are not significant.

Ordering Terms: The fastest-growing term is determined by first taking the term $c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ with the largest b^a value. If the b^a's of terms are equal, then we take the term with the largest d value. If the d's are also equal, then we take the term with the largest e value.

Dropping the Multiplicative Constant: The running time of an algorithm might be $f(n) = 3n^2$ or $f(n) = 100n^2$. We say it is $\Theta(n^2)$ when we do not care what the multiplicative constant c is. The function $f(n) = c \cdot b^{an} \cdot n^d \cdot (\log n)^e$ is in the class of functions denoted $\Theta(b^{an} \cdot n^d \cdot (\log n)^e)$.

Examples of Functions:

- $f(n) = 3n^3 \log n - 1000n^2 + n - 29$ is in the class $\Theta(n^3 \log n)$.
- $f(n) = 7 \cdot 4^n \cdot n^2/\log^3 n + 8 \cdot 2^n + 17 \cdot n^2 + 1000 \cdot n$ is in the class $\Theta(4^n \cdot n^2/\log^3 n)$.
- $\frac{1}{n} + 18$ is in the class $\Theta(1)$. Since $\frac{1}{n}$ is a lower-order term than 18, it is dropped.
- $\frac{1}{n^2} + \frac{1}{n}$ is in the class $\Theta(\frac{1}{n})$, because $\frac{1}{n^2}$ is a smaller term.

Examples of Classes:

Linear Functions $\Theta(n)$: The classic linear function is $f(n) = c \cdot n + b$. The notation $\Theta(n)$ excludes any with $c \leq 0$ but includes any function that is bounded between two such functions.

What Can Be Done in $\Theta(n)$ Time: Given an input of n items, it takes $\Theta(n)$ time simply to look at the input. Looping over the items and doing a constant amount of work for each takes another $\Theta(n)$ time. Say we take $t_1(n) = 2n$ and $t_2(n) = 4n$ for a total of $6n$ time. Now if you do not want to do more than linear time, are you allowed to do any more work? Sure. You can do something that takes $t_3(n) = 3n$ time and something else that takes $t_4(n) = 5n$ time. You are even allowed to do a few things that take a constant amount of time, totaling say $t_5(n) = 13$. The entire algorithm then takes the sum of these, $t(n) = 14n + 13$ time. This is still considered to be linear time.

Examples Included:

- $f(n) = 7n$ and $f(n) = 8.829n$
- $f(n) = (8 + \sin n)n$ and $f(n) = 8n + \log^{10} n + \frac{1}{n} - 1{,}000{,}000$

Examples Not Included:

- $f(n) = -n$ and $f(n) = 0n$ (fails $c > 0$)
- $f(n) = \frac{n}{\log_2 n}$ (too small)
- $f(n) = n \log_2 n$ (too big)

Quadratic Functions $\Theta(n^2)$: Two nested loops from 1 to n take $\Theta(n^2)$ time if each inner iteration takes a constant amount of time. An $n \times n$ matrix requires $\Theta(n^2)$ space if each element takes constant space.

Time for Sorting, $\Theta(n \log n)$: Another running time that arises often in algorithms is $\Theta(n \log n)$. For example, this is the number of comparisons needed to sort n elements.

Not Linear: The function $f(n) = n \log n$ grows slightly too quickly to be in the linear class of functions $\Theta(n)$. This is because $n \log n$ is $\log n$ times n, and $\log n$ is not constant.

A Polynomial: The classes $\Theta(n)$, $\Theta(n \log n)$, and $\Theta(n^2)$ are subclasses of the class of polynomial functions $n^{\Theta(1)}$. For example, though the function $f(n) = n \log n$ is too big for $\Theta(n)$ and too small $\Theta(n^2)$, it is in $n^{\Theta(1)}$ because it is bounded between n^1 and n^2, both of which are in $n^{\Theta(1)}$.

Logarithms $\Theta(\log(n))$: See Chapter 24 for how logarithmic functions like $\log_2(n)$ arise and for some of their rules.

Which Base: We write $\Theta(\log(n))$ without giving an explicit base. As shown in the list of rules about logarithms, $\log_{10}(n)$, $\log_2(n)$, and $\log_e(n)$ differ only by a multiplicative constant. Because we are ignoring multiplicative constants anyway, which base is used is irrelevant. The rules also indicate that $8\ \log_2(n^5)$ also differs only by a multiplicative constant. All of these functions are include in $\Theta(\log(n))$.

EXERCISE 25.1.1 *Which grows faster, 3^{4n} or 4^{3n}?*

EXERCISE 25.1.2 *Does the notation $(\Theta(1))^n$ mean the same thing as $2^{\theta(n)}$?*

EXERCISE 25.1.3 *Prove that $2^{0.5n} \leq 2^n/n^{100} \leq 2^n$ for sufficiently big n.*

EXERCISE 25.1.4 *(See solution in Part Five.) Prove that $n^2 \leq 3n^2 \log n \leq n^3$ for sufficiently big n.*

EXERCISE 25.1.5 *(See solution in Part Five.) Sort the terms in $f(n) = 100n^{100} + 3^{4n} + \log^{1,000} n + 4^{3n} + 2^{0.001n}/n^{100}$.*

EXERCISE 25.1.6 *For each of the following functions, sort its terms by growth rate. Get the big picture growth by classifying it into $2^{\Theta(n)}$, $n^{\Theta(1)}$, $\log^{\Theta(1)}(n)$, $\Theta(1)$ or into a similar and appropriate class. Also give its Theta approximation.*

1. $f(n) = 5n^3 - 17n^2 + 4$
2. $f(n) = 5n^3 \log n + 8n^3$
3. $f(n) = 2^{2^{5n}}$
4. $f(n) = 7^{3 \log_2 n}$
5. $f(n) = \{$ *1 if n is odd, 2 if n is even* $\}$
6. $f(n) = 2 \cdot 2^n \cdot n^2 \log_2 n - 7n^8 + 7\frac{3^n}{n^2}$

7. $f(n) = 100n^{100} + 3^{4n} + \log^{1,000}(n) + 4^{3n}$
8. $f(n) = 6\frac{n^4}{\log^3 n} + 8n^{100}2^{-5n} + 17$
9. $f(n) = \frac{1}{n^2} + \frac{5\log n}{n}$
10. $f(n) = 7\sqrt[5]{n} + 6\sqrt[3]{n}$
11. $f(n) = \frac{6n^{5.2}+7n^{7.5}}{2n^{3.1}+7n^{2.4}}$
12. $f(n) = -2n$
13. $f(n) = 5n^{\log^3 n}$

EXERCISE 25.1.7 *For each pair of classes of functions, how are they similar? How are they different? If possible, give a function that is included in the first of these but not included in the second. If possible, do the reverse, giving a function that is included in the second but not in the first.*

1. $\Theta(2^{2n})$ *and* $\Theta(2^{3n})$
2. $\Theta(2^n)$ *and* $3^{\Theta(n)}$

25.2 More about Asymptotic Notation

Other Useful Notations:

Name	Standard Notation	My Notation	Meaning
Theta	$f(n) = \Theta(g(n))$	$f(n) \in \Theta(g(n))$	$f(n) \approx c \cdot g(n)$
BigOh	$f(n) = O(g(n))$	$f(n) \leq O(g(n))$	$f(n) \leq c \cdot g(n)$
Omega	$f(n) = \Omega(g(n))$	$f(n) \geq \Omega(g(n))$	$f(n) \geq c \cdot g(n)$
Little Oh	$f(n) = o(g(n))$	$f(n) << o(g(n))$	$f(n) << g(n)$
Little Omega	$f(n) = \omega(g(n))$	$f(n) >> \omega(g(n))$	$f(n) >> g(n)$
Tilde	$f(n) = \tilde{\Theta}(g(n))$	$f(n) \in \tilde{\Theta}(g(n))$	$f(n) \approx \log^{\Theta(1)} \cdot g(n)$

Same: $7 \cdot n^3$ is within a constant of n^3. Hence, it is in $\Theta(n^3)$, $O(n^3)$, and $\Omega(n^3)$. However, because it is not much smaller than n^3, it is not in $o(n^3)$, and because it is not much bigger, it not in $\omega(n^3)$.

Smaller: $7 \cdot n^3$ is asymptotically much smaller than n^4. Hence, it is in $O(n^4)$ and in $o(n^4)$, but it is not in $\Theta(n^4)$, $\Omega(n^4)$, or $\omega(n^4)$.

Bigger: $7 \cdot n^3$ is asymptotically much bigger than n^2. Hence, it is in $\Omega(n^2)$ and in $\omega(n^2)$, but it is not in $\Theta(n^2)$, $O(n^2)$, or $o(n^2)$.

Log Factors: $7n^3 \log^2 n = \tilde{\Theta}(n^3)$ ignores the logarithmic factors.

Notation Considerations:

"∈" vs "=": I consider $\Theta(n)$ to be a class of functions, so I *ought* to use the set notation, $f(n) \in \Theta(g(n))$, to denote membership.

On the other hand, ignoring constant multiplicative factors, $f(n)$ has the same asymptotic growth as $g(n)$. Because of this, the notation $7n = \Theta(n)$ makes sense. This notation is standard.

Even the statements $3n^2 + 5n - 7 = n^{\Theta(1)}$ and $2^{3n} = 2^{\Theta(n)}$ make better sense when you think of the symbol Θ to mean "some constant." However, be sure to remember that $4^n \cdot n^2 = 2^{\Theta(n)}$ is also true.

"=" vs "≤": $7n = O(n^2)$ is also standard notation. This makes less sense to me. Because it means that $7n$ is at most some constant times n^2, a better notation would be $7n \le O(n^2)$. The standard notation is even more awkward, because $O(n) = O(n^2)$ should be true, but $O(n^2) = O(n)$ should be false. What sense does that make?

More Details: You can decide how much information about a function you want to reveal. If $f(n) = 5n^2 + 3n$, you could say

- $f(n) \in n^{\Theta(1)}$, i.e., a polynomial
- $f(n) \in \Theta(n^2)$, i.e., a quadratic
- $f(n) \in (5 + o(1))n^2 = 5n^2 + o(n^2)$, i.e., $5n^2$ plus some low-order terms.
- $f(n) \in 5n^2 + O(n)$, i.e., 5^2 plus at most some linear terms.

The Formal Definitions of Theta and BigOh:

$f(n) \in \Theta(g(n))$	iff	$\exists c_1, c_2 > 0$	$\exists n_0\ \forall n \ge n_0,\ c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$
$f(n) \in O(g(n))$	iff	$\exists c > 0$	$\exists n_0\ \forall n \ge n_0,\ 0 \le f(n) \le c \cdot g(n)$
$f(n) \in \Omega(g(n))$	iff	$\exists c > 0$	$\exists n_0\ \forall n \ge n_0,\ c \cdot g(n) \le f(n)$
$f(n) \in n^{\Theta(1)}$	iff	$\exists c_1, c_2 > 0$	$\exists n_0\ \forall n \ge n_0,\ n^{c_1} \le f(n) \le n^{c_2}$
$f(n) \in 2^{\Theta(n)}$	iff	$\exists c_1, c_2 > 0$	$\exists n_0\ \forall n \ge n_0,\ 2^{c_1 n} \le f(n) \le 2^{c_2 n}$
$f(n) \notin \Theta(g(n))$	iff	$\forall c_1, c_2 > 0$	$\forall n_0\ \exists n \ge n_0,\ [c_1 \cdot g(n) > f(n) \text{ or } f(n) > c_2 \cdot g(n)]$

Bounded Between: The statement $f(n) \in \Theta(g(n))$ means that the function $f(n)$ is bounded between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$. See Figure 25.1.

Requirements on c_1 and c_2: The only requirements on the constants are that c_1 be sufficiently small (e.g., 0.001) but positive and c_2 be sufficiently large (e.g., 1,000) to work, and that they be fixed (that is, do not depend on n). We allow unreasonably extreme values like $c_2 = 10^{100}$, to make the definition mathematically clean and not geared to a specific application.

Sufficiently Large n: Given fixed c_1 and c_2, the statement $c_1 g(n) \le f(n) \le c_2 g(n)$ should be true for all sufficiently large values of n, (i.e., $\forall n \ge n_0$).

Definition of Sufficiently Large n_0: Again to make the mathematics clean and not geared to a specific application, we will simply require that there exist some definition of sufficiently large n_0 that works. Exercise 25.0.2 gives an example in which n_0 needs to be unreasonably large.

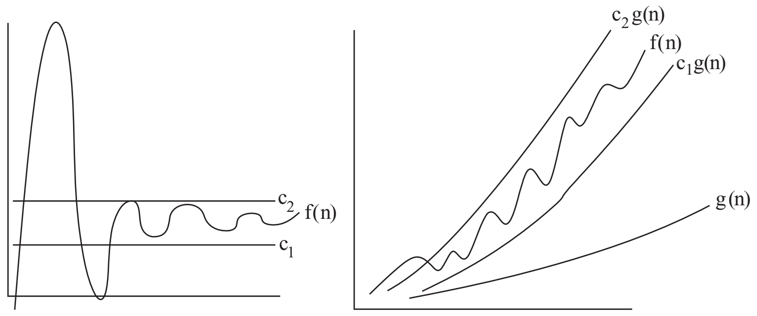


Figure 25.1: $f(n) \in \Theta(1)$ and $f(n) \in \Theta(g(n))$.

Proving $f(n) \in \Theta(g(n))$: Use the prover–adversary game.

- You as the prover provide c_1, c_2, and n_0.
- Some adversary gives you an n that is at least your n_0.
- You then prove that $c_1 g(n) \le f(n) \le c_2 g(n)$.

Example: For example, $2n^2 + 100n = \Theta(n^2)$. Let $c_1 = 2$, $c_2 = 3$, and $n_0 = 100$. Then, for all $n \ge 100$, we have $c_1 g(n) = 2n^2 \le 2n^2 + 100n = f(n)$ and $f(n) = 2n^2 + 100n \le 2n^2 + n \cdot n = 3n^2 = c_2 g(n)$. The values of c_1, c_2, and n_0 are not unique. For example, $n_0 = 1$, $c_2 = 102$, and $n_0 = 1$ also work, because for all $n \ge 1$ we have $f(n) = 2n^2 + 100n \le 2n^2 + 100n^2 = 102n^2 = c_2 g(n)$.

The Formal Definitions of Little Oh and Little Omega:

Class	$\lim_{n\to\infty} \frac{f(n)}{g(n)} =$	A practically equivalent definition
$f(n) = \Theta(g(n))$	Some constant	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
$f(n) = o(g(n))$	Zero	$f(n) = O(g(n))$, but $f(n) \neq \Omega(g(n))$
$f(n) = \omega(g(n))$	∞	$f(n) \neq O(g(n))$, but $f(n) = \Omega(g(n))$

Examples:

- $2n^2 + 100n = \Theta(n^2)$ and $\lim_{n\to\infty} \frac{2n^2+100n}{n^2} = 2$
- $2n + 100 = o(n^2)$ and $\lim_{n\to\infty} \frac{2n+100}{n^2} = 0$
- $2n^3 + 100n = \omega(n^2)$ and $\lim_{n\to\infty} \frac{2n^3+100n}{n^2} = \infty$

EXERCISE 25.2.1 *As in Exercise 25.1.7, compare the classes* $(5 + o(1))n^2$ *and* $5n^2 + O(n)$.

EXERCISE 25.2.2 *(See solution in Part Five.) Formally prove or disprove the following:*

1. $14n^9 + 5{,}000n^7 + 23n^2 \log n \in O(n^9)$
2. $2n^2 - 100n \in \Theta(n^2)$
3. $14n^8 - 100n^6 \in O(n^7)$

4. $14n^8 + 100n^6 \in \Theta(n^9)$
5. $2^{n+1} \in O(2^n)$
6. $2^{2n} \in O(2^n)$

EXERCISE 25.2.3 *Prove that if* $f_1(n) \in \Theta(g_1(n))$ *and* $f_2(n) \in \Theta(g_2(n))$*, then* $f_1(n) + f_2(n) \in \max(\Theta(g_1(n)), \Theta(g_2(n)))$.

EXERCISE 25.2.4 *Prove that if* $f_1(n), f_2(n) \in n^{\Theta(1)}$*, then* $f_1(n) \cdot f_2(n) \in n^{\Theta(1)}$.

EXERCISE 25.2.5 *Let* $f(n)$ *be a function. As you know,* $\Theta(f(n))$ *drops low-order terms and the leading coefficient. Explain what each of the following does:* $2^{\Theta(\log_2 f(n))}$ *and* $\log_2(\Theta(2^{f(n)}))$*. For each, explain to what extent the function is approximated.*

EXERCISE 25.2.6 *Let* x *be a real value. As you know,* $\lfloor x \rfloor$ *rounds it down to the next integer. Explain what each of the following does:* $2 \cdot \lfloor \frac{x}{2} \rfloor$, $\frac{1}{2} \cdot \lfloor 2 \cdot x \rfloor$*, and* $2^{\lfloor \log_2 x \rfloor}$.

EXERCISE 25.2.7 *Suppose that* $y = \Theta(\log x)$*. Which of the following are true:* $x = \Theta(2^y)$ *and* $x = 2^{\Theta(y)}$*? Why?*

EXERCISE 25.2.8 *(See solution in Part Five.) It is impossible to algebraically solve the equation* $x = 7y^3(\log_2 y)^{18}$ *for* y.

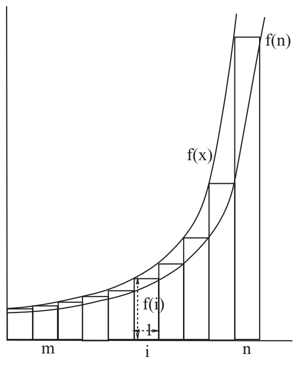
1. *Approximate* $7y^3(\log_2 y)^{18}$ *and then solve for* y*. This approximates the value of* y.
2. *Get a better approximation as follows. Plug in your above approximation for* y *to express* $(\log_2 y)^{18}$ *in terms of* x*. Plug this into* $x = 7y^3(\log_2 y)^{18}$*. Now solve for* y *again. (You could repeat this step for better and better approximations.)*
3. *Observe how a similar technique was used in Exercises 25.0.1 and 25.0.2 to approximate a solution for* $(\log_{10} n)^{1{,}000{,}000} = n$.

26 Adding-Made-Easy Approximations

```
algorithm $Eg(n)$
    loop $i = 1..n$
        loop $j = 1..i$
            loop $k = 1..j$
                put “Hi”
            end loop
        end loop
    end loop
end algorithm
```

The inner loop requires time $\sum_{k=1}^{j} 1 = j$.
The next requires $\sum_{j=1}^{i} \sum_{k=1}^{j} 1 = \sum_{j=1}^{i} j = \Theta(i^2)$.
The total is $\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} 1 = \sum_{i=1}^{n} \Theta(i^2) = \Theta(n^3)$.

Sums arise often in the study of computer algorithms. For example, if the ith iteration of a loop takes time $f(i)$ and it loops n times, then the total time is $f(1) + f(2) + f(3) + \cdots + f(n)$. This we denote as $\sum_{i=1}^{n} f(i)$. It can be approximated by the integral $\int_{x=1}^{n} f(x)\ \delta x$, because the first is the area under the stairs of height $f(i)$ and the second under the curve $f(x)$. (In fact, both $\sum$ (from the Greek letter sigma) and $\int$ (from the old long S) are S for sum.) Note that, even though the individual terms are indexed by i (or x), the total is a function of n. The goal now is to approximate $\sum_{i=1}^{n} f(i)$ for various functions $f(i)$.



Beyond learning the classic techniques for computing $\sum_{i=1}^{n} 2^i$, $\sum_{i=1}^{n} i$, and $\sum_{i=1}^{n} \frac{1}{i}$, we do not study how to evaluate sums exactly, but only how to approximate them to within a constant factor. We develop easy rules that most computer scientists use but for some reason are not usually taught, partly because they are not always true. We have formally proven when they are true and when not. We call them collectively the *adding-made-easy technique*.

26.1 The Technique

The following table outlines the few easy rules with which you will be able to compute $\Theta(\sum_{i=1}^{n} f(i))$ for functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$. (We consider more general functions at the end of this section.)

b^a	d	e	Type of Sum	$\sum_{i=1}^{n} f(i)$	Examples	
> 1	Any	Any	**Geometric Increase (dominated by last term)**	$\Theta(f(n))$	$\sum_{i=0}^{n} 2^{2^i}$	$\approx 1 \cdot 2^{2^n}$
					$\sum_{i=0}^{n} b^i$	$= \Theta(b^n)$
					$\sum_{i=0}^{n} 2^i$	$= \Theta(2^n)$
$= 1$	> -1	Any	**Arithmetic-like (half of terms approximately equal)**	$\Theta(n \cdot f(n))$	$\sum_{i=1}^{n} i^d$	$= \Theta(n \cdot n^d) = \Theta(n^{d+1})$
					$\sum_{i=1}^{n} i^2$	$= \Theta(n \cdot n^2) = \Theta(n^3)$
					$\sum_{i=1}^{n} i$	$= \Theta(n \cdot n) = \Theta(n^2)$
					$\sum_{i=1}^{n} 1$	$= \Theta(n \cdot 1) = \Theta(n)$
					$\sum_{i=1}^{n} \frac{1}{i^{0.99}}$	$= \Theta(n \cdot \frac{1}{n^{0.99}}) = \Theta(n^{0.01})$
	$= -1$	$=0$	**Harmonic**	$\Theta(\ln n)$	$\sum_{i=1}^{n} \frac{1}{i}$	$= \log_e(n) + \Theta(1)$
	< -1	Any	**Bounded tail (dominated by first term)**	$\Theta(1)$	$\sum_{i=1}^{n} \frac{1}{i^{1.001}}$	$= \Theta(1)$
					$\sum_{i=1}^{n} \frac{1}{i^2}$	$= \Theta(1)$
< 1	Any	Any			$\sum_{i=1}^{n} (\frac{1}{2})^i$	$= \Theta(1)$
					$\sum_{i=0}^{n} b^{-i}$	$= \Theta(1)$

Four Different Classes of Solutions: All of the sums that we will consider have one of four different classes of solutions. The intuition for each is quite straightforward.

Geometrically Increasing: If the terms grow very quickly, the total is dominated by the last and biggest term $f(n)$. Hence, one can approximate the sum by only considering the last term: $\sum_{i=1}^{n} f(i) = \Theta(f(n))$.

Examples: Consider the classic sum in which each of the n terms is twice the previous, $1 + 2 + 4 + 8 + 16 + \cdots + 2^n$. Either by examining areas within Figure 26.1.a or 26.1.b or using simple induction, one can prove that the total is always one less than twice the biggest term: $\sum_{i=0}^{n} 2^i = 2 \times 2^n - 1 = \Theta(2^n)$. More generally, $\sum_{i=0}^{n} b^i \approx \frac{b}{b-1} \cdot b^n$, which can be approximated by $\Theta(f(n)) = \Theta(b^n)$. (Similarly, $\int_{x=0}^{n} b^x \, \delta x = \frac{1}{\ln b} b^n$.) The same is true for even fastergrowing functions like $\sum_{i=0}^{n} 2^{2^i} \approx 1 \times 2^{2^n}$.

Basic-Form Exponentials: The same technique, $\sum_{i=1}^{n} f(i) = \Theta(f(n))$, works for all basic-form exponentials, i.e., for $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ with $b^a > 1$, we have that $\sum_{i=1}^{n} f(i) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$.

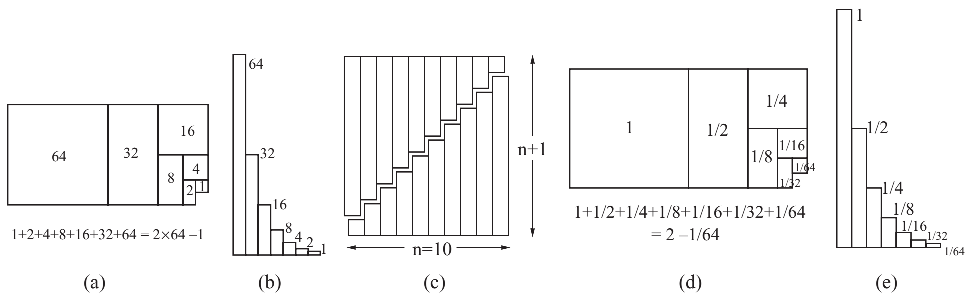


Figure 26.1: Examples of geometrically increasing, arithmetic-like, and bounded-tail function.

Arithmetic-like: If half of the terms are roughly the same size, then the total is roughly the number of terms times the last term, namely $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$.

Examples:

Constant: Clearly the sum of n ones is n, i.e., $\sum_{i=1}^{n} 1 = n$. This is $\Theta(n \cdot f(n))$.

Linear: The classic example is the sum in which each of the n terms is only one bigger than the previous, $\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + 5 + \cdots + n = \frac{n(n+1)}{2}$. This can be approximated using $\Theta(n \cdot f(n)) = \Theta(n^2)$. See Figure 26.1.

Polynomials: Both $\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$ and more generally $\sum_{i=1}^{n} i^d = \frac{1}{d+1}n^{d+1} + \Theta(n^d)$ can be approximated with $\Theta(n \cdot f(n)) = \Theta(n \cdot n^d) = \Theta(n^{d+1})$. (Similarly, $\int_{x=0}^{n} x^d \,\delta x = \frac{1}{d+1}n^{d+1}$.)

Above Harmonic: $\sum_{i=1}^{n} \frac{1}{n^{0.999}} \approx 1{,}000\ n^{0.001}$ can be approximated with $\Theta(n \cdot f(n)) = \Theta(n \cdot n^{-0.999}) = \Theta(n^{0.001})$.

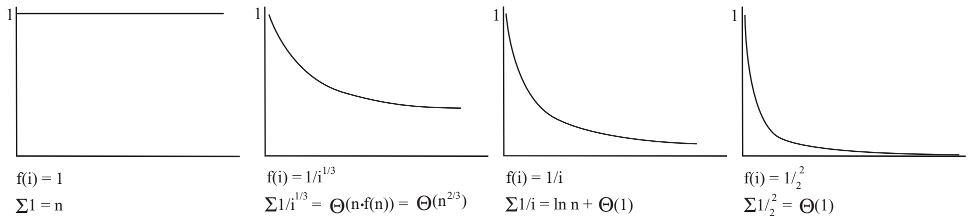
Basic-Form Polynomials: The same technique, $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$, works for all basic-form polynomials, constants, and slowly decreasing functions, i.e., for $f(n) = \Theta(n^d \cdot \log^e n)$ with $d > 1$ we have that $\sum_{i=1}^{n} f(i) = \Theta(n^{d+1} \cdot \log^e n)$.

Bounded Tail: If the terms shrink quickly, the total is dominated by the first and biggest term $f(1)$, which is assumed here to be $\Theta(1)$, i.e., $\sum_{i=1}^{n} f(i) = \Theta(1)$.

Examples: The classic sum here is when each term is half of the previous, $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots + \frac{1}{2^n}$. See Figure 26.1.d and 26.1.e. The total approaches but never reaches 2, so that $\sum_{i=0}^{n} (\frac{1}{2})^i = 2 - (\frac{1}{2})^n = \Theta(1)$. Similarly, $\sum_{i=1}^{n} \frac{1}{n^{1.001}} \approx 1{,}000 = \Theta(1)$ and $\sum_{i=1}^{n} \frac{1}{n^2} \approx \frac{\Pi}{6} \approx 1.5497 = \Theta(1)$.

Basic-Form with Bounded Tail: The same technique, $\sum_{i=1}^{n} f(i) = \Theta(1)$, works for all basic-form polynomially or exponentially decreasing functions, i.e., for $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ with $b^a = 1$ and $d < 1$ or with $b^a < 1$.

The Harmonic Sum: The sum $\sum_{i=1}^{n} \frac{1}{i}$ is referred to as the *harmonic sum* because of its connection to music. It arises surprisingly often and it has an unexpected total:



On the Boundary: The boundary between those sums for which $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$ and those for which $\sum_{i=1}^{n} f(i) = \Theta(1)$ occurs when these approximations meet, i.e., when $\Theta(n \cdot f(n)) = \Theta(1)$. This occurs at the harmonic function $f(n) = \frac{1}{n}$. Given that both approximations say the total is $\sum_{i=1}^{n} \frac{1}{i} = \Theta(1)$, it is reasonable to think that this is the answer, but it is not.

The Total: It turns out that the total is within 1 of the natural logarithm, $\sum_{i=1}^{n} \frac{1}{i} = \log_e n + \Theta(1)$. (Similarly, $\int_{x=1}^{n+1} \frac{1}{x}\, \delta x = \log_e n + \Theta(1)$.) See Figure 26.2.

More Examples:

Geometric Increasing:

- $\sum_{i=1}^{n} 8\frac{2^i}{i^{100}} + i^3 = \Theta(\frac{2^n}{n^{100}})$
- $\sum_{i=1}^{n} 3^{i \log i} + 5^i + i^{100} = \Theta(3^{n \log n})$
- $\sum_{i=1}^{n} 2^{i^2} + i^2 \log i = \Theta(2^{n^2})$
- $\sum_{i=1}^{n} 2^{2^i - i^2} = \Theta(2^{2^n - n^2})$

Arithmetic (Increasing):

- $\sum_{i=1}^{n} i^4 + 7i^3 + i^2 = \Theta(n^5)$
- $\sum_{i=1}^{n} i^{4.3} \log^3 i + i^3 \log^9 i = \Theta(n^{5.3} \log^3 n)$

Arithmetic (Decreasing):

- $\sum_{i=2}^{n} \frac{1}{\log i} = \Theta(\frac{n}{\log n})$
- $\sum_{i=1}^{n} \frac{\log^3 i}{i^{0.6}} = \Theta(n^{0.4} \log^3 n)$

Bounded Tail:

- $\sum_{i=1}^{n} \frac{\log^3 i}{i^{1.6} + 3i} = \Theta(1)$
- $\sum_{i=1}^{n} \frac{i^{100}}{2^i} = \Theta(1)$
- $\sum_{i=1}^{n} \frac{1}{2^{2^i}} = \Theta(1)$

Stranger Examples:

- A useful fact is $\sum_{i=m}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{m-1} f(i)$. Hence, $\sum_{i=m}^{n} \frac{1}{i} = \Theta(\log n) - \Theta(\log m) = \Theta(\log \frac{n}{m})$.

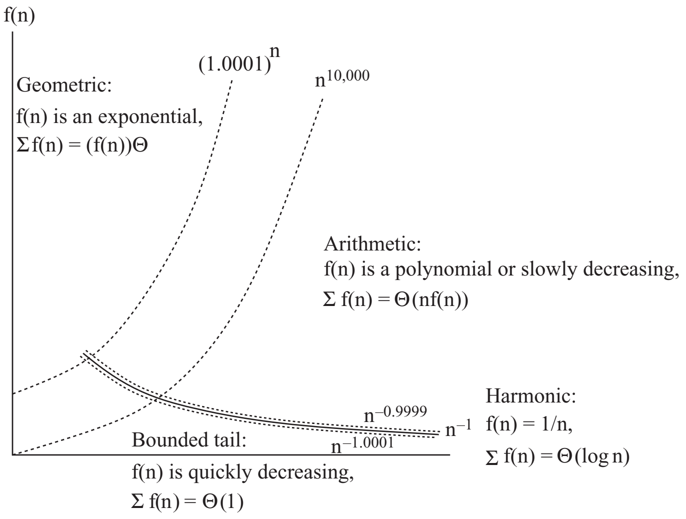


Figure 26.2: Boundaries between geometric, arithmetic, harmonic, and bounded tail.

- If the sum is arithmetic, then the sum is the number of terms times the largest term. This gives $\sum_{i=m}^{m+n} i^2 = \Theta(n \cdot (m+n)^2)$.
- To solve $\sum_{i=1}^{5n^2+n} i^3 \log i$, let $N = 5n^2 + n$ denote the number of terms. Then $\sum_{i=1}^{N} i^3 \log i = \Theta(N \cdot f(N)) = \Theta(N^4 \log N)$. Substituting back in for N gives $\sum_{i=1}^{5n^2+n} i^3 \log i = \Theta((5n^2+n)^4 \log(5n^2+n)) = \Theta(n^8 \log n)$.
- Between terms, i changes, but n does not. Hence, n can be treated like a constant. For example, $\sum_{i=1}^{n} i \cdot n \cdot m = nm \cdot \sum_{i=1}^{n} i = nm \cdot \Theta(n^2) = \Theta(n^3 m)$.
- In $\sum_{i=\frac{n}{2}}^{n} \frac{1}{i^2}$, the terms are decreasing fast enough to be bounded by the first term. Here, however, the first term is not $\Theta(1)$, but is

$$\Theta\left(\frac{1}{(\frac{n}{2})^2}\right) = \Theta\left(\frac{1}{n^2}\right)$$

- When in doubt, start by determining the first term, the last term, and the number of terms. In $\sum_{i=1}^{\log_2 n} 2^{\log_2 n - i} \cdot i^2$, the first term is $f(1) = 2^{\log n - 1} \cdot 1^2 = \Theta(n)$, and the last term is $f(\log n) = 2^{\log n - \log n} \cdot (\log n)^2 = \Theta(\log^2 n)$. The terms decrease geometrically in i. The total is then $\Theta(f(1)) = \Theta(n)$.
- $\sum_{i=1}^{n} \sum_{j=0}^{n} i^2 j^3 = \sum_{i=1}^{n} i^2 [\sum_{j=0}^{n} j^3] = \sum_{i=1}^{n} i^2 \Theta(n^4) = \Theta(n^4)[\sum_{i=1}^{n} i^2] = \Theta(n^4)\Theta(n^3) = \Theta(n^7)$.

EXERCISE 26.1.1 *Give the Θ approximation of the following sums. Indicate which rule you use, and show your work.*

1. $\sum_{i=0}^{n} 7i^3 - 300i^2 + 16$
2. $\sum_{i=0}^{n} i^8 + \frac{2^{3i}}{i^2}$
3. $\sum_{i=0}^{n} \frac{1}{i^{1.1}}$

4. $\sum_{i=0}^{n} \frac{1}{i^{0.9}}$
5. $\sum_{i=0}^{n} 7\frac{i^{3.72}}{\log^2 i} - 300i^2 \log^9 i$
6. $\sum_{i=1}^{n} \frac{\log^e i}{i}$
7. $\sum_{i=1}^{\log n} n \cdot i^2$
8. $\sum_{i=0}^{n} \sum_{j=0}^{m} \frac{j}{i}$
9. $\sum_{i=1}^{n} \sum_{j=1}^{i} j^i$
10. $\sum_{i=1}^{n} \sum_{j=1}^{i^2} ij \log(i)$

26.2 Some Proofs for the *Adding-Made-Easy* Technique

This section presents a few of the classic techniques for summing and sketches the proof of the adding-made-easy technique.

Simple Geometric Sums:

Theorem: When $b > 1$, $\sum_{i=1}^{n} b^i = \Theta(f(n))$ and when $b < 1$, $\sum_{i=1}^{n} f(i) = \Theta(1)$.

Proof:

$$S = 1 + b + b^2 + \cdots b^n$$
$$b \cdot S = b + b^2 + b^3 + \cdots b^{n+1}.$$

Subtracting those two equations gives

$$(1-b) \cdot S = 1 - b^{n+1}$$
$$S = \frac{1-b^{n+1}}{1-b} \text{ or } \frac{b^{n+1}-1}{b-1}$$
$$= \Theta(\max(f(0), f(n)))$$

Ratio between Terms: To prove that a geometric sum is not more than a constant times the biggest term, we must compare each term $f(i)$ with this biggest term. One way to do this is to first compare each consecutive pairs of terms $f(i)$ and $f(i+1)$.

Theorem: If for all sufficiently large i, the ratio between terms is bounded away from one, i.e., $\exists b > 1,\ \exists n_0,\ \forall i \geq n_0,\ f(i+1)/f(i) \geq b$, then $\sum_{i=1}^{n} f(i) = \Theta(f(n))$.
Conversely, if $\exists b < 1,\ \exists n_0,\ \forall i \geq n_0,\ \frac{f(i+1)}{f(i)} \leq b$, then $\sum_{i=1}^{n} f(i) = \Theta(1)$.

Examples:

Typical: With $f(i) = 2^i/i$, the ratio between consecutive terms is

$$\frac{f(i+1)}{f(i)} = \frac{2^{i+1}}{i+1} \cdot \frac{i}{2^i} = 2 \cdot \frac{i}{i+1} = 2 \cdot \frac{1}{1+\frac{1}{i}}$$

which is at least 1.99 for sufficiently large i. Similarly for any $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$ with $b^a > 1$.

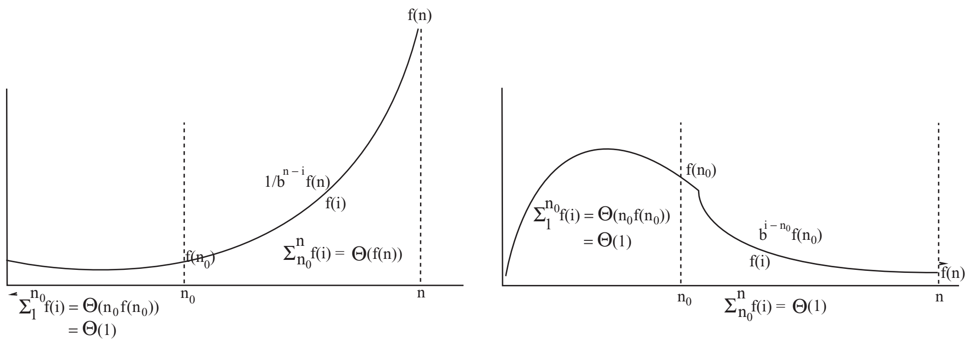


Figure 26.3: *In both pictures, the total before n gets sufficiently large is some constant. On the left, the total for large n is bounded by an growing exponential, and on the right by a decreasing exponential.*

Not Bounded Away: On the other hand, the arithmetic function $f(i) = i$ has a ratio between the terms of $\frac{i+1}{i} = 1 + \frac{1}{i}$. Though this is always bigger than one, it is not bounded away from one by any constant $b > 1$.

Proof: If $\forall i \geq n_0,\ f(i+1)/f(i) \geq b > 1$, then it follows either by unwinding or induction that

$$f(i) \leq \left(\frac{1}{b}\right)^1 f(i+1) \leq \left(\frac{1}{b}\right)^2 f(i+2) \leq \left(\frac{1}{b}\right)^3 f(i+3) \leq \cdots \leq \left(\frac{1}{b}\right)^{n-i} f(n)$$

See Figure 26.3. This gives that

$$\sum_{i=1}^{n} f(i) = \sum_{i=1}^{n_0} f(i) + \sum_{i=n_0}^{n} f(i) \leq \Theta(1) + \sum_{i=n_0}^{n} \left(\frac{1}{b}\right)^{n-i} f(n) \leq \Theta(1) + f(n) \cdot \sum_{j=0}^{n} \left(\frac{1}{b}\right)^{j}$$

which we have already proved is $\Theta(f(n))$.

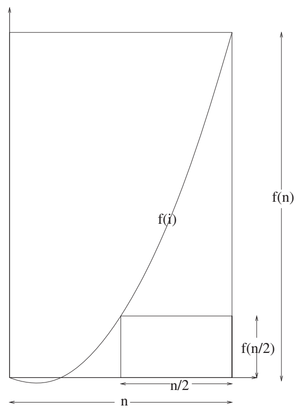
A Simple Arithmetic Sum: We prove as follows that $\sum_{i=1}^{n} i = \Theta(n \cdot f(n)) = \Theta(n^2)$:

$$\begin{aligned}
S &= \quad 1 \quad + \quad 2 \quad + \quad 3 \quad + \cdots + n-2 + n-1 + \quad n \\
S &= \quad n \quad + n-1 + n-2 + \cdots + \quad 3 \quad + \quad 2 \quad + \quad 1 \\
2S &= n+1 + n+1 + n+1 + \cdots + n+1 + n+1 + n+1 \\
&= n \cdot (n+1) \\
S &= \tfrac{1}{2} n \cdot (n+1)
\end{aligned}$$

Arithmetic Sums: We will now justify the intuition that if half of the terms are roughly the same size, then the total is roughly the number of terms times the last term, namely $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$.

Theorem: If for sufficiently large n, the function $f(n)$ is nondecreasing, and $f(\frac{n}{2}) = \Theta(f(n))$, then $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$.

Examples:



Typical: The function $f(n) = n^d$ for $d \geq 0$ is nondecreasing and $f(\frac{n}{2}) = \left(\frac{n}{2}\right)^d = \frac{1}{2^d} f(n)$. Similarly for $f(n) = \Theta(n^d \cdot \log^e n)$. We consider $-1 < d < 0$ later.

Without the Property: The function $f(n) = 2^n$ does not have this property, because $f(\frac{n}{2}) = 2^{n/2} = \frac{1}{2^{n/2}} f(n)$.

Proof: Because $f(i)$ is nondecreasing, half of the terms are at least the middle term $f(\frac{n}{2})$, and all of the terms are at most the biggest term $f(n)$. Hence, $\frac{n}{2} \cdot f(\frac{n}{2}) \leq \sum_{i=1}^{n} f(i) \leq n \cdot f(n)$. Because $f(\frac{n}{2}) = \Theta(f(n))$, these bounds match, giving $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$.

The Harmonic Sum: The harmonic sum is a famous sum that arises surprisingly often. The total $\sum_{i=1}^{n} \frac{1}{i}$ is within 1 of $\log_e n$. However, we will not bound it quite so closely.

Theorem: $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$.

Proof: One way of approximating the harmonic sum is to break it into $\log_2 n$ blocks with 2^k terms in the kth block, and then to prove that the total for each block is between $\frac{1}{2}$ and 1:

$$\sum_{i=1}^{n} \frac{1}{i} = \overbrace{\underbrace{\frac{1}{1}}_{\leq 1\cdot 1=1}}^{\geq 1\cdot\frac{1}{2}=\frac{1}{2}} + \overbrace{\underbrace{\frac{1}{2}+\frac{1}{3}}_{\leq 2\cdot\frac{1}{2}=1}}^{\geq 2\cdot\frac{1}{4}=\frac{1}{2}} + \overbrace{\underbrace{\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}}_{\leq 4\cdot\frac{1}{4}=1}}^{\geq 4\cdot\frac{1}{8}=\frac{1}{2}} + \overbrace{\underbrace{\frac{1}{8}+\cdots+\frac{1}{15}}_{\leq 8\cdot\frac{1}{8}=1}}^{\geq 8\cdot\frac{1}{16}=\frac{1}{2}} + \cdots$$

From this, it follows that $\frac{1}{2} \cdot \log_2 n \leq \sum_{i=1}^{n} \frac{1}{i} \leq 1 \cdot \log_2 n$.

Close to Harmonic: We will now use a similar technique to prove the remaining two cases of the adding-made-easy technique.

Theorem: $\sum_{i=1}^{n} 1/i^{d'}$ is $\Theta(1)$ if $d' > 1$ and is $\Theta(n \cdot f(n))$ if $d' < 1$. (Similarly for $f(n) = \Theta(n^d \cdot \log^e n)$ with $d < -1$ or > -1.)

Proof: As we did with the harmonic sum, we break the sum $\sum_{i=1}^{n} f(n)$ into blocks where the kth block has the 2^k terms $\sum_{i=2^k}^{2^{k+1}-1} f(i)$. Because the terms are decreasing, the total for the block is at most $F(k) = 2^k \cdot f(2^k)$. The total overall is then at most

$$\sum_{k=0}^{\log_2 n} F(k) = \sum_{k=0}^{\log_2 n} 2^k \cdot f(2^k) = \sum_{k=0}^{\log_2 n} \frac{2^k}{(2^k)^{d'}} = \sum_{k=0}^{N} \frac{1}{2^{k\cdot(d'-1)}}.$$

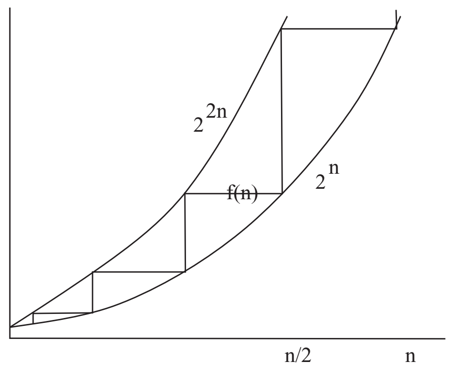
If $d' > 1$, then this sum is exponentially decreasing and converges to $\Theta(1)$. If $d' < 1$, then this sum is exponentially increasing and diverges to $\Theta(F(N)) = \Theta(2^{\log_2 n} \cdot f(n)) = \Theta(n \cdot f(n)))$.

Functions without the Basic Form: (Warning: This topic may be a little hard.) Until now we have only considered functions with the basic form $f(n) = \Theta(b^{an} \cdot n^d \cdot \log^e n)$. We would like to generalize the adding-made-easy technique as follows:

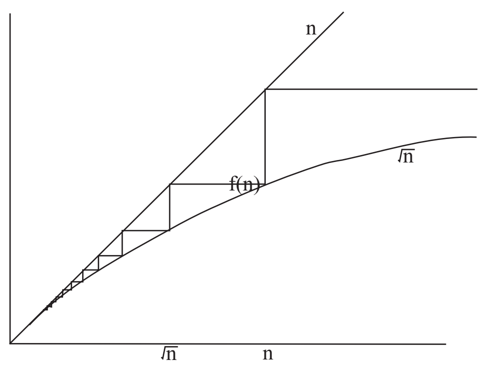
	Geometric Increasing	**Arithmetic**	**Harmonic**	**Bounded Tail**
If	$f(n) \geq 2^{\Omega(n)}$	$f(n) = n^{\Theta(1)-1}$	$f(n) = \Theta(\frac{1}{n})$	$f(n) \leq n^{-1-\Omega(1)}$
then	$\sum_{i=1}^{n} f(i) = \Theta(f(n))$	$\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$	$\sum_{i=1}^{n} f(i) = \Theta(\log n)$	$\sum_{i=1}^{n} f(i) = \Theta(1)$

Example: Consider $f(n) = n^{8+\frac{1}{n}}$ or $f(n) = n^{-\frac{1}{n}}$. They are bounded between n^{d_1} and n^{d_2} for constants $d_2 \geq d_1 > 0 - 1$, and hence for both we have $f(n) \in n^{\Theta(1)-1}$. Adding made easy then gives that $\sum_{i=1}^{n} f(i) = \Theta(n \cdot f(n))$, so that $\sum_{i=1}^{n} i^{8+\frac{1}{i}} = \Theta(n^{9+\frac{1}{n}})$ and $\sum_{i=1}^{n} i^{-\frac{1}{i}} = \Theta(n^{1-\frac{1}{n}})$.

Counterexample: The goal here is to predict the sum $\sum_{i=1}^{n} f(i)$ from the value of the last term $f(n)$. We are unable to do this if the terms oscillate like those created with sines, cosines, floors, and ceilings. Exercise 26.2.4 proves that $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$ and $f(n) = 2^{[\frac{1}{2}\cos(\pi \log_2 n)+1.5]\cdot n}$ are counterexamples for the geometric case and that $f(n) = 2^{2^{\lfloor \log \log n \rfloor}}$ is one for the arithmetic case.



(Not to scale)



(Not to scale)

Simple Analytical Functions: We can prove that the adding-made-easy technique works for all functions $f(n)$ that can be expressed with n, real constants, plus, minus, times, divide, exponentiation, and logarithms. Such functions are said to be *simple analytical.*

Proof Sketch: I will only give a sketch of the proof here. For the geometric case, we must prove that if $f(n)$ is simple analytical and $f(n) \geq 2^{\Omega(n)}$, then $\exists b > 1$,

$\exists n_0,\ \forall n \geq n_0,\ f(n+1)/f(n) \geq b$. From this, the ratio-between-terms theorem above gives that $\sum_{i=1}^{n} f(i) = \Theta(f(n))$.

Because the function is growing exponentially, we know that generally it grows at least as fast as fast as b^n for some constant $b > 1$ and hence $f(n+1)/f(n) \geq b$, or equivalently $h(n) = \log f(n+1) - \log f(n) - \log b > 0$ for an infinite number of values for n.

A deep theorem about simple analytical functions is that they cannot oscillate forever and hence can change sign at most a finite number of places. It follows that there must be a last place n_0 at which the sign changes. We can conclude that $\forall n \geq n_0,\ h(n) > 0$ and hence $f(n+1)/f(n) \geq b$.

The geometrically decreasing case is the same except $f(n+1)/f(n) \leq b$. The arithmetic case is similar except that it proves that if $f(n)$ is simple analytical and $f(n) = n^{\Theta(1)-1}$, then $f(\frac{n}{2}) = \Theta(f(n))$.

EXERCISE 26.2.1 *(See solution in Part Five.) Zeno's classic paradox is that Achilles is traveling* 1 km/hr *and has* 1 km *to travel. First he must cover half his distance, then half of his remaining distance, then half of this remaining distance, ... He never arrives. By Bryan Magee states, "People have found it terribly disconcerting. There must be a fault in the logic, they have said. But no one has yet been fully successful in demonstrating what it is." Resolve this ancient paradox by adding up the time required for all steps.*

EXERCISE 26.2.2 *Prove that if* $\exists b < 1,\ \exists n_0,\ \forall i \geq n_0,\ f(i+1)/f(i) \leq b$, *then* $\sum_{i=n_0}^{n} f(i) = \Theta(f(n_0)) = \Theta(1)$.

EXERCISE 26.2.3 *A seeming paradox is how one could have a vessel that has finite volume and infinite surface area. This (theoretical) vessel could be filled with a small amount of paint but require an infinite amount of paint to paint. For* $h \in [1, \infty)$, *its cross section at* h *units from its top is a circle with radius* $r = \frac{1}{h^c}$ *for some constant* c. *Integrate (or add up) its cross-sectional circumference to compute its surface area, and integrate (or add up) its cross-sectional area to compute its volume. Give a value for* c *such that its surface area is infinite and its volume is finite.*

EXERCISE 26.2.4 *(See solution in Part Five.)*

1. *For* $f(n) = 2^{2^{\lceil \log_2 n \rceil}}$, *prove that* $f(n) \geq 2^{\Omega(n)}$
2. *and that* $\sum_{i=1}^{n} f(i) \neq \Theta(f(n))$.
3. *For* $f(n) = 2^{2^{\lfloor \log \log n \rfloor}}$, *prove that* $f(n) = n^{\Theta(1)-1}$
4. *and that* $\sum_{i=1}^{n} f(i) \neq \Theta(n \cdot f(n))$.
5. *Plot* $f(n) = 2^{[\frac{1}{2}\cos(\pi \log_2 n)+1.5]\cdot n}$, *and prove that it is also a counterexample for the geometric case.*

27 Recurrence Relations

A wise man told the king to give him one grain of rice one for the first square of a chessboard and for the each remaining square to give him twice the number for the previous square. Thirty-two days later, the king realized that there is not enough rice in all of world to reward him. The number of grains on the *n*th square is given by the recurrence relation $T(1) = 1$ and $T(n) = 2T(n-1)$.

The algebraic equation $x^2 = x + 2$ specifies the value of an unknown real that must be found. The differential equation $\frac{\delta f(x)}{\delta x} = f(x)$ specifies functions from reals to reals that must be found. Similarly, a recurrence relations like $T(n) = 2 \times T(n-1)$ specifies functions from integers to reals. One way to solve each of these is to guess a solution and check to see if it works. Here $T(n) = 2^n$ works, i.e., $2^n = 2 \times 2^{n-1}$. However, $T(n) = c \cdot 2^n$ also works for each value of c. Making the further requirement that $T(1) = 1$ narrows the solution set to only $T(n) = \frac{1}{2} \cdot 2^n = 2^{n-1}$.

27.1 The Technique

Timing of Recursive Programs: Recursive relations are used to determine the running time of recursive programs. (See Chapter 8.) For example, if a routine, when given an instance of size n, does $f(n)$ work itself and then recurses a times on subinstances of size $\frac{n}{b}$, then the running time is $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$.

See Section 8.6 to learn more about the *tree of stack frames*. Each stack frame consists of one execution of the routine on a single instance, ignoring subroutine calls. The top-level stack frame is called by the user on the required input instance. It recurses on a number of subinstances, creating the next level of stack frames. These in turn recurse again until the instance is sufficiently small that the stack frame returns without recursing. These final stack frames are referred to as *base cases*.

Let $T(n)$ denote the number of "Hi"s that the entire tree of stack frames, given the following code, prints on an instance of size n. The top level stack frame prints "Hi" $f(n)$ times. It then recurses a times on subinstances of size $\frac{n}{b}$. If $T(n)$ is the number of

"Hi"s for instances of size n, then it follows that $T\left(\frac{n}{b}\right)$ is the number for instances of size $\frac{n}{b}$. Repeating this a times will take time $a \cdot T\left(\frac{n}{b}\right)$. It follows that the total number satisfies the recursive relation $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. The goal of this section is to determine which function $T(n)$ satisfies this relation.

If instead the routine recurses a times on instances of size $n - b$, then the related recurrence relation will be $T(n) = a \cdot T(n - b) + f(n)$.

```
algorithm  $Eg(I_n)$
⟨pre-cond⟩: $I_n$ is an instance of size $n$.
⟨post-cond⟩: Prints $T(n)$ "Hi"s.
begin
    $n = |I_n|$
    if( $n \le 1$) then
        put "Hi"
    else
        loop $i = 1..f(n)$
            put "Hi"
        end loop
        loop $i = 1..a$
            $I_{\frac{n}{b}}$ = an input of size $\frac{n}{b}$
            $Eg(I_{\frac{n}{b}})$
        end loop
    end if
end algorithm
```

When the input has size zero or one, only one "Hi" is printed. In general, we will assume that recursive programs spend $\Theta(1)$ time for instances of size $\Theta(1)$. We express this as $T(1) = 1$, or more generally as $T(\Theta(1)) = \Theta(1)$.

Solving Recurrence Relations: Consider $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$, where $f(n) = \Theta(n^c \cdot \log^d n)$ or $f(n) = 0$.

$\frac{\log a}{\log b}$ **vs** c	d	**Dominated by**	$T(n)$	**Example** $\left(\frac{\log_3 9}{\log_3 3} = 2\right)$	**Solution**
$<$	Any	Top level	$\Theta(f(n))$	$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n^4$	$\Theta(n^4)$
$=$	> -1	All levels	$\Theta(f(n) \log n)$	$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n^2$	$\Theta(n^2 \log n)$
	< -1	Base cases	$\Theta\left(n^{\frac{\log a}{\log b}}\right)$	$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + \frac{n^2}{log^2 n}$	$\Theta(n^2)$
$>$	Any			$t(n) = 9 \cdot T\left(\frac{n}{3}\right)$	

Consider $T(n) = a \cdot T(n - b) + f(n)$, where $f(n) = \Theta(n^c \cdot \log^d n)$ or $f(n) = 0$.

a	$f(n)$	Dominated by	$T(n)$	Example	Solution
> 1	Any	Base cases	$\Theta(a^{\frac{n}{b}})$	$T(n) = 9 \cdot T(n-3) + n^4$	$\Theta(9^{\frac{n}{3}})$
$= 1$	≥ 1	All levels	$\Theta(n \cdot f(n))$	$T(n) = T(n-3) + n^4$	$\Theta(n^5)$
	$= 0$	Base cases	$\Theta(1)$	$T(n) = T(n-3)$	$\Theta(1)$

A Growing Number of Subinstances of Shrinking Size: Each instance having a subinstances means that the number of subinstances grows exponentially by a factor of a. On the other hand, the sizes of the subinstances shrink exponentially by a factor of b. The amount of work that the instance must do is the function f of this instance size. Whether the growing or the shrinking dominates this process depends upon the relationship between a, b, and $f(n)$.

Dominated By: When total work $T(n)$ done in the tree of stack frames is dominated by the work $f(n)$ done by the top stack frame, we say that the work is *dominated by the top level* of the recursion. The solution in this case will be $T(n) = \Theta(f(n))$. Conversely, we say that it is *dominated by the base cases* when the total is dominated by the sum of the work done by the base cases. Because each base case does only a constant amount of work, the solution will be $T(n) = \Theta(\text{\# of base cases})$, which is $\Theta(n^{\log a / \log b})$, $\Theta(a^{\frac{n}{b}})$, or $\Theta(1)$ in the above examples. Finally, if the amounts of work at the different levels of recursion are sufficiently close to each other, then we say that the total work is *dominated by all the levels* and the total is the number of levels times this amount of work, namely $T(n) = \Theta(\log n \cdot f(n))$ or $\Theta(n \cdot f(n))$.

The Ratio $\frac{\log a}{\log b}$: See Chapter 24 for a discussion about logarithms. One trick that it gives us is that when computing the ratio between two logarithms, the base used does not matter, because changing the base will introduce the same constant both on the top and the bottom, which will cancel. Hence, when computing such a ratio, you can choose whichever base makes the calculation the easiest. For example, to compute $\frac{\log 16}{\log 8}$, the obvious base to use is 2, because $\frac{\log_2 16}{\log_2 8} = \frac{4}{3}$. This is useful in giving that $T(n) = 16 \cdot T(\frac{n}{8}) + f(n) = \Theta(n^{\log 16 / \log 8}) = \Theta(n^{4/3})$. On the other hand, to compute $\frac{\log 9}{\log 27}$, the obvious base to use is 3, because $\frac{\log_3 9}{\log_3 27} = \frac{2}{3}$, and hence we have $T(n) = 9 \cdot T(\frac{n}{27}) + f(n) = \Theta(n^{2/3})$. Another interesting fact given is that $\log 1 = 0$, which gives that $T(n) = 1 \cdot T(\frac{n}{2}) + f(n)$, $T(n) = \Theta(n^{\log 1/\log 2}) = \Theta(n^0) = \Theta(1)$.

EXERCISE 27.1.1 *(See solution in Part Five.) Give solutions for the following examples:*

1. $T(n) = 2T(\frac{n}{2}) + n$
2. $T(n) = 2T(\frac{n}{2}) + 1$
3. $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^3}{\log^3 n})$

4. $T(n) = 32T(\frac{n}{4}) + \Theta(\log n)$
5. $T(n) = 27T(\frac{n}{3}) + \Theta(n^3 \log^4 n)$
6. $T(n) = 8T(\frac{n}{4}) + \Theta((\frac{n}{\log n})^{1.5})$
7. $T(n) = 4T(\frac{n}{2}) + \Theta(\frac{n^2}{\log n})$

EXERCISE 27.1.2 *Give solutions for the following stranger examples:*

1. $T(n) = 4T(\frac{n}{2}) + \Theta(n^3 \log\log n)$
2. $T(n) = 4T(\frac{n}{2}) + \Theta(2^n)$
3. $T(n) = 4T(\frac{n}{2}) + \Theta(\log\log n)$
4. $T(n) = 4T(\frac{n}{2} - \sqrt{n} + \log n - 5) + \Theta(n^3)$

27.2 Some Proofs

I now present a few of the classic techniques for computing recurrence relations. As our example we will solve $T(n) = GT(n/0) + f(n)$, for $f(n) = n^c$.

Guess and Verify: To begin consider the example $T(n) = 4T\left(\frac{n}{2}\right) + n$ and $T(1) = 1$.

Plugging In: If we can guess $T(n) = 2n^2 - n$, the first way to verify that this is the solution is to simply plug it into the two equations and make sure that they are satisfied:

Left Side	Right Side
$T(n) = 2n^2 - n$	$4T(\frac{n}{2}) + n = 4\left[2\left(\frac{n}{2}\right)^2 - \left(\frac{n}{2}\right)\right] - n = 2n^2 - n$
$T(1) = 2n^2 - n = 1$	1

Proof by Induction: Similarly, we can use induction to prove that this is the solution for all n (at least for $n = 2^i$).

Base Case: Because $T(1) = 2(1)^2 - 1 = 1$, it is correct for $n = 2^0$.

Induction Step: Let $n = 2^i$. Assume that it is correct for $2^{i-1} = \frac{n}{2}$. Because $T(n) = 4T(\frac{n}{2}) + n = 4\left[2\left(\frac{n}{2}\right)^2 - \left(\frac{n}{2}\right)\right] + n = 2n^2 - n$, it is also true for n.

Calculate Coefficients: Suppose that instead we are only able to guess that the formula has the form $T(n) = an^2 + bn + c$ for some constants a, b, and c:

Left Side	Right Side
$T(n) = an^2 + bn + c$	$4T(\frac{n}{2}) + n = 4\left[a\left(\frac{n}{2}\right)^2 + b\left(\frac{n}{2}\right) + c\right] - n = an^2 + (2b+1)n + 4c$
$T(1) = a + b + c$	1

These left and right sides must be equal for all n. Both have a as the coefficient of n^2, which is good. To make the coefficient in front n be the same, we need that $b = 2b + 1$, which gives $b = -1$. To make the constant coefficient be the same, we need that $c = 4c$, which gives $c = 0$. To make $T(1) = a(1)^2 + b(1) + c = a(1)^2 - (1) + 0 = 1$, we need that $a = 2$. This gives us the solution $T(n) = 2n^2 - n$ that we had before.

Calculate Exponent: If we were to guess that $a \cdot T\left(\frac{n}{b}\right)$ is much bigger than $f(n)$, then $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \approx a \cdot T\left(\frac{n}{b}\right)$. Further we guess that $T(n) = n^\alpha$ for some constant α. Plugging this into $T(n) = a \cdot T\left(\frac{n}{b}\right)$ gives $n^\alpha = a \cdot \left(\frac{n}{b}\right)^\alpha$, or $b^\alpha = a$. Taking the log gives $\alpha \cdot \log b = \log a$, and solving gives $\alpha = \frac{\log a}{\log b}$. In conclusion, $T(n) = \Theta(n^{\log a/\log b}) = \Theta(n^{\log 4/\log 2}) = \Theta(n^2)$.

Unwinding: A useful technique is to unwind a recursive relation for a few steps and to look for a pattern:

$$\begin{aligned} T(n) &= f(n) + a \cdot T\left(\frac{n}{b}\right) \;=\; f(n) + a \cdot \left[f\left(\frac{n}{b}\right) + a \cdot T\left(\frac{n}{b^2}\right)\right] \\ &= f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot T\left(\frac{n}{b^2}\right) \;=\; f(n) + af\left(\frac{n}{b}\right) + a^2 \cdot \left[f\left(\frac{n}{b^2}\right) + a \cdot T\left(\frac{n}{b^3}\right)\right] \\ &= f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + a^3 \cdot T\left(\frac{n}{b^3}\right) \;=\; \cdots \\ &= \sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right) + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^{h} a^i \cdot f\left(\frac{n}{b^i}\right)\right). \end{aligned}$$

Filling the Table: My recommended way to evaluate recursive relations is to fill out a table like that in Figure 27.1.

(a) Number of Stack Frames at the *i*th Level: Level 0 contains the one initial stack frame at the top of the tree of stack frame. It recursively calls a times. Hence, level 1 has a stack frames. Each of these recursively calls a times, giving a^2 stack frames at level 2. Each successive level, the number of stack frames goes up by a factor of a, giving a^i at level i.

(b) Size of Instance at the *i*th Level: The top stack frame at level 0 is given an instance of size n. It recurses on a subinstances of size $\frac{n}{b}$. Stack frames at level 1, given instances of size $\frac{n}{b}$, recurse on subinstance of size n/b^2. Each successive level decreases the instance size by a factor of b, giving size n/b^i at level i.

(c) Time within One Stack Frame: On an instance of size n, a single stack frame requires $f(n)$ time. Hence, a stack frame at the ith level, with an instance of size n/b^i, requires $f(n/b^i)$ time.

(d) Number of Levels: The recursive program stops recursing when the instance becomes sufficiently small, say of size 0 or 1. Let h denote the level at which this

Example	$\boldsymbol{T(n) = 4T(n/2) + n}$	$\boldsymbol{T(n) = 9T(n/3) + n^2}$	$\boldsymbol{T(n) = 2T(n/4) + n^2}$
(a) No. of frames at the ith level	4^i	9^i	2^i
(b) Instance size at ith level	$\frac{n}{2^i}$	$\frac{n}{3^i}$	$\frac{n}{4^i}$
(c) Time within one stack frame	$f\left(\frac{n}{2^i}\right) = \left(\frac{n}{2^i}\right)$	$f\left(\frac{n}{3^i}\right) = \left(\frac{n}{3^i}\right)^2$	$f\left(\frac{n}{4^i}\right) = \left(\frac{n}{4^i}\right)^2$
(d) No. of levels	$\frac{n}{2^h} = 1$ $h = \frac{\log n}{\log 2} = \Theta(\log n)$	$\frac{n}{3^h} = 1$ $h = \frac{\log n}{\log 3} = \Theta(\log n)$	$\frac{n}{4^h} = 1$ $h = \frac{\log n}{\log 4} = \Theta(\log n)$
(e) No. of base case stack frames	$4^h = 4^{\frac{\log n}{\log 2}}$ $= n^{\frac{\log 4}{\log 2}} = n^2$	$9^h = 9^{\frac{\log n}{\log 3}}$ $= n^{\frac{\log 9}{\log 3}} = n^2$	$2^h = 2^{\frac{\log n}{\log 4}}$ $= n^{\frac{\log 2}{\log 4}} = n^{\frac{1}{2}}$
(f) $T(n)$ as a sum	$\sum_{i=0}^{h}(\text{\# at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{\Theta(\log n)} 4^i \cdot \left(\frac{n}{2^i}\right)$ $= n \cdot \sum_{i=0}^{\Theta(\log n)} 2^i$	$\sum_{i=0}^{h}(\text{\# at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{\Theta(\log n)} 9^i \cdot \left(\frac{n}{3^i}\right)^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} 1$	$\sum_{i=0}^{h}(\text{\# at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{\Theta(\log n)} 2^i \cdot \left(\frac{n}{4^i}\right)^2$ $= n^2 \cdot \sum_{i=0}^{\Theta(\log n)} \left(\frac{1}{8}\right)^i$
(g) Dominated by?	Geometric increase: base cases	Arithmetic sum: all levels	Geometric decrease: top level
(h) $\Theta(\mathrm{T(n)})$	$T(n) = \Theta\left(n^{\log a/\log b}\right)$ $= \Theta\left(n^{\frac{\log 4}{\log 2}}\right) = \Theta\left(n^2\right)$	$T(n) = \Theta\left(f(n)\log n\right)$ $= \Theta\left(n^2 \log n\right)$	$T(n) = \Theta\left(f(n)\right)$ $= \Theta\left(n^2\right)$

Figure 27.1: Solving $T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^c)$ by filling in the table.

occurs. We have seen that the instances at level h have size n/b^h. Setting $n/b^h = 1$ and solving for h gives $h = \frac{\log n}{\log b}$.

(e) Number of Base Case Stack Frames: The number of stack frames at level i is a^i. Hence, the number of base case stack frames is $a^h = a^{\log n/\log b}$. Though this looks ugly, Chapter 24 gives $a^{\log n/\log b} = (2^{\log a})^{\log n/\log b} = 2^{\log a \cdot \log n/\log b} = (2^{\log n})^{\log a/\log b} = n^{\log a/\log b}$. Given that $\frac{\log a}{\log b}$ is simply some constant, $n^{\log a/\log b}$ is a simple polynomial in n.

(f) $T(n)$ as a Sum: There are a^i stack frames at level i, and each requires $f(n/b^i)$ time, for a total of $a^i \cdot f(n/b^i)$ at the level. We obtain the total time $T(n)$ for the recursion by summing the times at all of these levels. This gives

$$T(n) = \left[\sum_{i=0}^{h-1} a^i \cdot f\left(\frac{n}{b^i}\right)\right] + a^h \cdot T(1) = \Theta\left(\sum_{i=0}^{h} a^i \cdot f\left(\frac{n}{b^i}\right)\right).$$

Plugging in $f(n) = n^c$ gives

$$T(n) = \Theta\left(\sum_{i=0}^{h} a^i \cdot \left(\frac{n}{b^i}\right)^c\right) = \Theta\left(n^c \cdot \sum_{i=0}^{h} \left(\frac{a}{b^c}\right)^i\right)$$

(g) Dominated By: The key things to remember about this sum are that it has $\Theta(\log n)$ terms, the top term being $a^0 f(n/b^0) = f(n)$ and the base case term being $a^n f(n/b^n) = a^{\log n/\log b} f(n) = n^{\log a/\log b}\ \Theta(n^{\log a/\log b})$. According to the adding-made-easy approximations given in Chapter 26, if either the top term or the base case term is sufficiently bigger then the other, then the total is dominated by this term. On the other hand, if they are roughly the same, then the total is approximately the number of terms times a typical term.

(h) Evaluating the Sum: If $\frac{\log a}{\log b} < c$, then $a/b^c < 1$, giving that the terms in $T(n) =$
$\Theta(n^c \cdot \sum_{i=0}^{h} (\frac{a}{b^c})^i)$ decrease exponentially, giving $T(n) = \Theta(\text{top term}) = \Theta(f(n))$. Similarly, if $\frac{\log a}{\log b} > c$, then the terms increase exponentially, giving $T(n) = \Theta(\text{base case term}) = \Theta(n^{\log a/\log b})$. If $\frac{\log a}{\log b} = c$, then $\frac{a}{b^c} = 1$, giving $T(n) = \Theta(n^c \cdot \sum_{i=0}^{h} (\frac{a}{b^c})^i) = \Theta(n^c \cdot \sum_{i=0}^{h} 1) = \Theta(n^c \cdot h) = \Theta(f(n) \log n)$.

EXERCISE 27.2.1 *(See solution in Part Five.) Solve the famous Fibonacci recurrence relation $Fib(0) = 0$, $Fib(1) = 1$, and $Fib(n) = Fib(n-1) + Fib(n-2)$ by plugging in $Fib(n) = \alpha^n$ and solving for α.*

EXERCISE 27.2.2 *(See solution in Part Five.) Solve the following by unwinding them:*

1. $T(n) = T(n-1) + n$
2. $T(n) = 2 \cdot T(n-1) + 1$

Example	$\boldsymbol{T(n) = aT(n-b) + n^c}$	$\boldsymbol{T(n) = T(n-b) + n^c}$	$\boldsymbol{T(n) = T(n-b) + 0}$
(a) No. of frames at the ith level	a^i	1	
(b) Instance size at ith level	$n - i \cdot b$		
(c) Time within one stack frame	$f(n - i \cdot b) = (n - i \cdot b)^c$		$f(n - i \cdot b) = 0$ except for the base case, which has work $\Theta(1)$
(d) No. of levels	$n - h \cdot b = 0,\ h = \frac{n}{b}$ Having a base case of size zero makes the math the cleanest.		
(e) No of base case stack frames	$a^h = a^{\frac{1}{b}n}$	1	
(f) $T(n)$ as a sum	$\sum_{i=0}^{h}(\text{\# at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{n/b} a^i \cdot (n - i \cdot b)^c$	$\sum_{i=0}^{h}(\text{\# at level}) \cdot (\text{time each})$ $= \sum_{i=0}^{n/b} 1 \cdot (n - i \cdot b)^c$	$\sum_{i=0}^{h}(\text{\# at level}) \cdot (\text{time each})$ $= \left(\sum_{i=0}^{n/b-1} 1 \cdot 0\right) + 1 \cdot \Theta(1)$ $= \Theta(1)$
(g) Dominated by	Geometric increase: base cases	Arithmetic sum: all levels	Geometric decrease: base cases
(h) $\Theta(T(n))$	$T(n) = \Theta\left(a^{n/b}\right)$	$T(n) = \Theta\left(\frac{n}{b} \cdot n^c\right)$ $= \Theta(n^{c+1})$	$T(n) = \Theta(1)$

Figure 27.2: Solving $T(n) = a \cdot T(n-b) + \Theta(n^c)$ by filling in the table.

EXERCISE 27.2.3 *Does setting the size of the base case to 5 have any practical effect? How about setting the size to zero, i.e., $n/b^h = 0$?*

Why does this happen? If instead instances at the ith *level had size n − itshape AB, would an instance size of 0, 1, or 2 be better? How many levels h are there?*

EXERCISE 27.2.4 *(See solution in Part Five.) Section 27.2 solves $T(n) = aT(n/b) + f(n)$ for $f(n) = n^c$. If $f(n) = n^c \log^d n$ and $\frac{\log a}{\log b} = c$, then the math is harder. Compute the sum for $d > -1$, $d = -1$, $d < -1$. (Hint: Reverse the order of the terms.)*

EXERCISE 27.2.5 *Use the method in Figure 27.2 to compute each of the following recursive relations.*

1. $T(n) = nT(n-1) + 1$
2. $T(n) = 2T(\sqrt{n}) + n$
3. $T(n) = T(u \cdot n) + T(v \cdot n) + \Theta(n)$ *where* $u + v = 1$.

EXERCISE 27.2.6 *Running time:*

```
algorithm Careful(n)
⟨pre-cond⟩: n is an integer.
⟨post-cond⟩: Q(n) "Hi"s are printed for some odd function Q
begin
    if( n ≤ 1 )
        PrintHi(1)
    else
        loop i = 1 ... n
            PrintHi(i)
        end loop
        loop i = 1 ... 8
            Careful($\frac{n}{2}$)
        end loop
    end if
end algorithm
```

```
algorithm PrintHi(n)
⟨pre-cond⟩: n is an integer.
⟨post-cond⟩: $n^2$ "Hi"s are printed
begin
    loop i = 1 ... $n^2$
        Print("Hi")
    end loop
end algorithm
```

1. *Give and solve the recurrence relation for the number of "Hi"s, $Q(n)$. Show your work. Give a sentence or two giving the intuition.*
2. *What is the running time (time complexity) of this algorithm as a function of the size of the input?*

28 A Formal Proof of Correctness

Though I mean is not to be too formal, it is useful to at least understand the required steps in a formal proof of correctness.

Specifications: Before we prove that an algorithm is correct, we need to know precisely what it is supposed to do.

> ***Preconditions:*** Assertions that are promised be true about the input instance.
>
> ***Postconditions:*** Assertions that must be true about the output.

Correctness: Consider some instance. If this instance meets the preconditions, then after the code has been run, the output must meet the postconditions:

$$\langle \textit{pre-cond} \rangle \ \& \ \textit{code}_{\textit{alg}} \Rightarrow \langle \textit{post-cond} \rangle$$

The correctness of an algorithm is only with respect to the stated specifications. It does not guarantee that it will work in situations that are not taken into account by this specification.

Breaking the Computation Path into Fragments: The method to prove that an algorithm is correct is as follows. Assertions are inserted into the code to act as checkpoints. Each assertion is a statement about the current state of the computation's data structures that is either true or false. If it is false, then something has gone wrong in the logic of the algorithm. These assertions break the path of the computation into fragments. For each such fragment, we prove that if the assertion at the beginning of the fragment is true and the fragment gets executed, then the assertion at the end of the fragment will be true. Combining all these fragments back together gives that if the first assertion is true and the entire computation is executed, then the last assertion will be true.

A Huge Number of Paths: There are likely an exponential number or even an infinite number of different paths that the computation might take, depending on the input instance and the tests that occur along the way. In contrast, there are not many different computation path fragments. Hence, it is much easier to prove the correctness of each fragment than of each path.

The following table outlines the computational path fragments that need to be tested for different code structures.

Single Line of Code:	
$\langle \textit{pre-assignment-cond} \rangle$: The variables x and y have meaningful values. $z = x + y$ $\langle \textit{post-assignment-cond} \rangle$: The variable z takes on the sum of the value of x and the value of y. The previous value of z is lost.	
Blocks of Code:	
$\langle \textit{assertion}_0 \rangle$ $\textit{code}_1$ $\langle \textit{assertion}_1 \rangle$ $\textit{code}_2$ $\langle \textit{assertion}_2 \rangle$	$\left.\begin{array}{l}[\langle \textit{assertion}_0 \rangle \;\&\; \textit{code}_1 \Rightarrow \langle \textit{assertion}_1 \rangle] \\ [\langle \textit{assertion}_1 \rangle \;\&\; \textit{code}_2 \Rightarrow \langle \textit{assertion}_2 \rangle]\end{array}\right\}$ $\Rightarrow [\langle \textit{assertion}_0 \rangle \;\&\; \textit{code}_{1\&2} \Rightarrow \langle \textit{assertion}_2 \rangle]$
If Statements:	
$\langle \textit{pre-if-cond} \rangle$ if($\langle \textit{test} \rangle$) then $\textit{code}_{\textit{true}}$ else $\textit{code}_{\textit{false}}$ end if $\langle \textit{post-if-cond} \rangle$	$\left.\begin{array}{l}[\langle \textit{pre-if-cond} \rangle \;\&\; \langle \textit{test} \rangle \;\&\; \textit{code}_{\textit{true}} \Rightarrow \langle \textit{post-if-cond} \rangle] \\ [\langle \textit{pre-if-cond} \rangle \;\&\; \neg\langle \textit{test} \rangle \;\&\; \textit{code}_{\textit{false}} \Rightarrow \langle \textit{post-if-cond} \rangle]\end{array}\right\}$ $\Rightarrow [\langle \textit{pre-if-cond} \rangle \;\&\; \textit{code} \Rightarrow \langle \textit{post-if-cond} \rangle]$
Loops:	
$\langle \textit{pre-loop-cond} \rangle$ *loop* $\langle \textit{loop-invar} \rangle$ exit when $\langle \textit{exit-cond} \rangle$ $\textit{code}_{\textit{loop}}$ end loop $\langle \textit{post-loop-cond} \rangle$	$\left.\begin{array}{l}[\langle \textit{pre-loop-cond} \rangle \Rightarrow \langle \textit{loop-invar} \rangle] \\ [\langle \textit{loop-invar}' \rangle \;\&\; \neg\langle \textit{exit-cond} \rangle \;\&\; \textit{code}_{\textit{loop}} \Rightarrow \langle \textit{loop-invar}'' \rangle] \\ [\langle \textit{loop-invar} \rangle \;\&\; \langle \textit{exit-cond} \rangle \Rightarrow \langle \textit{post-loop-cond} \rangle] \\ \textit{Termination}\end{array}\right\}$ $\Rightarrow [\langle \textit{pre-loop-cond} \rangle \;\&\; \textit{code} \Rightarrow \langle \textit{post-loop-cond} \rangle]$
Function Call:	
$\langle \textit{pre-call-cond} \rangle$ $\textit{output} = \textit{Func}(\textit{input})$ $\langle \textit{post-call-cond} \rangle$	$\left.\begin{array}{l}[\langle \textit{pre-call-cond} \rangle \Rightarrow \langle \textit{pre-cond} \rangle_{\textit{Func}}] \\ [\langle \textit{post-cond} \rangle_{\textit{Func}} \Rightarrow \langle \textit{post-call-cond} \rangle]\end{array}\right\}$ $\Rightarrow [\langle \textit{pre-call-cond} \rangle \;\&\; \textit{code} \Rightarrow \langle \textit{post-call-cond} \rangle]$