Lecture II
RECURRENCES

Recurrences arise naturally in analyzing the complexity of recursive algorithms and in probabilistic analysis. We introduce some basic techniques for solving recurrences. A recurrence is a recursive relation for a complexity function $T(n)$. Here are two examples:

$$F(n) = F(n-1) + F(n-2) \tag{1}$$

and

$$T(n) = n + 2T(n/2). \tag{2}$$

Looks familiar?

The reader may recognize the first as the recurrence for Fibonacci numbers, and the second as the complexity of the Merge Sort, described in Lecture 1. These recurrences have[1] the following "separable form":

$$T(n) = G(n, T(n_1), \ldots, T(n_k)) \tag{3}$$

where $G(x_0, x_1, \ldots, x_k)$ is a function or expression in $k+1$ variables and $n_1, \ldots, n_k$ are all strictly less than n. Each n_i is a function of n. E.g., in (1), we have $k = 2$ and $n_1 = n-1, n_2 = n-2$. But in (2), we have $k = 1$ and $n_1 = n/2$.

What does it mean to "solve" recurrences such as equations (1) and (2)? The Fibonacci recurrence and the Mergesort recurrence has the following well-known solutions:

$$F(n) = \Theta(\phi^n)$$

where $\phi = (1+\sqrt{5})/2 = 1.618\ldots$ is the golden ratio, and

Solve up to Θ-order

$$T(n) = \Theta(n \log n).$$

In this book, we generally estimate complexity functions $T(n)$ only up to its Θ-order. If only an upper bound or lower bound is needed, and we determine $T(n)$ up to its O-order or to Ω-order. In rare cases, we may be able to derive the exact solution (in fact, this is possible for $T(n)$ and $F(n)$ above). One benefit of Θ-order solutions is this − most of the recurrences we treat in this book can be solved by only elementary methods, without assuming continuity or using calculus.

The variable "n" is called the **designated variable** of the recurrence (3). If there are non-designated variables, they are supposed to be held constant. In mathematics, we usually reserve "n" for natural numbers or perhaps integers. In the above examples, this is the natural interpretation for n. But one of the first steps we take in solving recurrences is to re-interpret n (or whatever is the designated variable) to range over the real numbers. The corresponding recurrence equation (3) is then called a **real recurrence**. For this reason, we may prefer the symbol "x" as our designated variable, since x is normally viewed as a real variable.

All recurrences are real

What does an extension to real numbers mean? In the Fibonacci recurrence (1), what is $F(2.5)$? In Merge Sort (2), what does $T(\pi) = T(3.14159\ldots)$ represent? The short answer is, we don't really care.

In addition to the recurrence (3), we generally need the **boundary conditions** or **initial values** of the function $T(n)$. They give us the values of $T(n)$ *before* the recurrence (3) becomes valid. Without initial values, $T(n)$ is generally under-determined. For our example (1), if n ranges over natural numbers, then the initial conditions

$$F(0) = 0, \qquad F(1) = 1$$

give rise to the standard Fibonacci numbers, *i.e.*, $F(n)$ is the nth Fibonacci number. Thus $F(2) = 1, F(3) = 2, F(4) = 3$, etc. On the other hand, if we use the initial conditions $F(0) = F(1) = 0$, then the solution is trivial: $F(n) = 0$ for all $n \geq 0$. Thus, our assertion earlier that $F(n) = \Theta(\phi^n)$ is the solution to (1) is not[2] really true without knowing the initial conditions. On the other hand, $T(n) = \mathcal{O}(n \log n)$ can be shown to hold for (2) regardless of the initial conditions.

Some initial conditions lead to trivial solutions

Exercises

Exercise 0.1: Consider the non-homogeneous version of Fibonacci recurrence $F(n) = F(n-1) + F(n-2) + f(n)$ for some function $f(n)$. If $f(n) = 1$, show that $F(n) = \Omega(c^n)$ for some $c > 1$, regardless of the initial conditions. Try to find the largest value for c. Does your bound hold if we have $f(n) = n$ instead? ♢

Exercise 0.2: Let $T(n) = aT(n/b) + n$, where $a > 0$ and $b > 1$. How sensitive is this recurrence to the initial conditions? More precisely, if $T_1(n)$ and $T_2(n)$ are two solutions corresponding to two initial conditions, what is the strongest relation you can infer between T_1 and T_2? ♢

Exercise 0.3: Consider recurrences of the form

$$T(n) = (T(n-1))^2 + g(n). \tag{4}$$

In this exercise, we restrict n to natural numbers and use explicit boundary conditions.
(a) Show that the number of binary trees of height at most n is given by this recurrence with $g(n) = 1$ and the boundary condition $T(1) = 1$. Show that this particular case of (4) has solution

$$T(n) = \left\lfloor k^{2^n} \right\rfloor. \tag{5}$$

(b) Show that the number of Boolean functions on n variables is given by (4) with $g(n) = 0$ and $T(1) = 2$. Solve this.
NOTE: Aho and Sloane (1973) investigate the recurrence (4). ♢

Exercise 0.4: Let T, T' be binary trees and $|T|$ denote the number of nodes in T. Define the relation $T \sim T'$ recursively as follows: (BASIS) If $|T| = 0$ or 1 then $|T| = |T'|$. (INDUCTION) If $|T| > 1$ then $|T'| > 1$ and either (i) $T_L \sim T'_L$ and $T_R \sim T'_R$, or (ii) $T_L \sim T'_R$ and $T_R \sim T'_L$. Here T_L and T_R denote the left and right subtrees of T.
(a) Use this to give a recursive algorithm for checking if $T \sim T'$.
(b) Give the recurrence satisfied by the running time $t(n)$ of your algorithm.
(c) Give asymptotic bounds on $t(n)$. ♢

End Exercises

§1. Simplification

[1] Non-separable recurrences looks like $G(n, T(n), T(n_1), \ldots, T(n_k)) = 0$, but these are rare.

[2] The reason behind this is that (1) is a homogeneous recurrence while (2) is non-homogeneous. For instance, $F(n) = F(n-1) + F(n-2) + 1$ would be non-homogeneous and its Θ-solution would not depend on the initial conditions.

In the real world, when faced with an actual recurrence to be solved, there are usually some simplifications steps to be taken.

- **Initial Condition.** In this book, we normally state recurrence without any initial conditions. This is deliberate: we expect the student to supply some specific initial conditions, based on the **Default Initial Condition** (DIC): the DIC says that there is some $n_1 > 0$ such that the recurrence holds for all $n \geq n_1$, and for $n < n_1$, $T(n)$ can be assigned arbitrary values. The intent is for the student to make convenient choices for n_1 and the initial values of $T(n)$. Normally, we make choices so that the resulting solution has a simple form. To use DIC, we need not specify n_1 and the initial values of $T(n)$ before hand. We just proceed to solve the recurrence, and at the appropriate moments, just specify these initial values.

strong assumptions!!

> If the DIC is too strong, we might consider the **weak Default Initial Condition** where we assume that there exists $0 < n_0 < n_1$, and constants $0 < C_0 \leq C_1$ such that
> $$(\forall\ n_0 \leq n < n_1)[C_0 \leq T(n) \leq C_1]. \tag{6}$$
> A solution under weak DIC will have to carry along the parameters C_1, C_2 in its solution.

What is the justification for this approach? It allows us to focus on the recurrence itself rather than the initial conditions. In many cases, this arbitrariness does not affect the asymptotic behavior of the solution.

- **Extension to Real Functions.** Even if the function $T(n)$ is originally defined for natural numbers n, we will now treat $T(n)$ as a real function (*i.e.*, n is viewed as a real variable), and defined for n sufficiently large. Under the Default Initial Condition (6), we assume $T(n)$ is defined for all $n > n_0$. See the Exercise for an alternative approach ("ample domain") that avoids extensions to real functions.

- **Conversion into a Recurrence Equation.** If we begin with a recurrence inequality such as $T(n) \leq G(n, T(n_1), \ldots, T(n_k))$, we simply treat it as an equality relation: $T(n) = G(T(n_1), \ldots, T(n_k))$. Our eventual solution for $T(n)$ is only an upper bound on the original function. Similarly, if we had started with $T(n) \geq G(n, T(n_1), \ldots, T(n_k))$, the eventual solution is only a lower bound.

¶1. Special Simplifications. Suppose the running time of an algorithm satisfies the following inequality:

$$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 6n + \lg n - 4, \tag{7}$$

for integer $n > 100$, with boundary condition

$$T(n) = 3n^2 - 4n + 2 \tag{8}$$

for $0 \leq n \leq 100$. Such a **recurrence inequation** may arises in some imagined implementation of Merge Sort, with special treatment for $n \leq 100$. Our general simplification steps tells us to (a) discard the specific boundary conditions (8) in favor of DIC, (b) treat $T(n)$ as a real function, and (c) write the recurrence as a equation.

What other simplifications might apply here? Let us convert (7) into the following

$$T(n) = 2T(n/2) + n. \tag{9}$$

This represents two additional simplifications: (i) We replaced the term "$+6n + \lg n - 4$" by some simple expression ("$+n$") with same Θ-order. (ii) We have removed the ceiling and floor functions.

Step (i) is justified because this does not affect the Θ-order (if this is not clear, then you can always come back to verify this claim). Step (ii) exploits the fact that we now treat $T(n)$ as a real function, so we need not worry about non-integral arguments when we remove the ceiling or floor functions. Also, it does not affect the asymptotic value of $T(n)$ here.

While these remarks may not be obvious, they should seem reasonable. Ultimately, one ought to return to such simplifications to justify them.

Exercises

Exercise 1.1: Show that our above simplifications of the the recurrence (7) (with its initial conditions) cannot affect the asymptotic order of the solution. [Show this for ANY choice of a Default Boundary Condition.] ♢

Exercise 1.2: We seek counterexamples to the claim that we can replace $\lceil n/2 \rceil$ by $n/2$ in a recurrence without changing the Θ-order of the solution.
(a) Construct a function $g(n)$ that provides a counter example for the following recurrence: $T(n) = T(\lceil n/2 \rceil) + g(n)$. HINT: make $g(n)$ depend on the parity of n.
(b) Construct a different counter example of the form $T(n) = h(n)T(\lceil \frac{n}{2} \rceil)$ for a suitable function $h(n)$.
♢

Exercise 1.3: Show examples where the choice of initial conditions can change the Θ-order of the solution $T(n)$. HINT: Choose $T(n)$ to grow exponentially fast. ♢

Exercise 1.4: Suppose x, n are positive numbers satisfying the following "recurrence" equation,

$$2^x = x^{2n}.$$

Solve for x as a function of n, showing

$$x(n) = [1 + o(1)]2n \log_2(2n).$$

HINT: take logarithms. This is an example of a bootstrapping argument where we use an approximation of $x(n)$ to derive yet a better approximation. See, e.g., Purdom and Brown [13]. ♢

Exercise 1.5: [Ample Domains] Our approach of considering real functions is non-standard. The standard approach to solving recurrences in the algorithms literature is the following. Consider the simplification of (7) to (9). Suppose, instead of assuming $T(n)$ to be a real function (so that (9) makes sense for all values of n), we continue to assume n is a natural number. It is easy to see that $T(n)$ is completely defined by (9) iff n is a power of 2. We say that (9) is closed over the set $D_0 := \{2^k : k \in \mathbb{N}\}$ of powers of 2. In general, we say a recurrence is "closed over a set $D \subseteq \mathbb{R}$" if for all $n \in D$, the recurrence for $T(n)$ depends only on smaller values n_i that also belong in D (unless n_i lies within the boundary condition).
(a) Let us call a set $D \subseteq \mathbb{R}$ an "ample set" if, for some $\alpha > 1$, the set $D \cap [n, \alpha \cdot n]$ is non-empty for all $n \in \mathbb{N}$. Here $[n, \alpha n]$ is closed real interval between n and αn. If the solution $T(n)$ is sufficiently "smooth", then knowing the values of $T(n)$ at an ample set D gives us a good approximation to values where $n \notin D$. In this question, our "smoothness assumption"

is simply: $T(n)$ *is monotonic non-decreasing.* Suppose that $T(n) = n^k$ for n ranging over an ample set D. What can you say about $T(n)$ for $n \notin D$? What if $T(n) = c^n$ over D? What if $T(n) = 2^{2^n}$ over D?
(b) Suppose $T(n)$ is recursively expressed in terms of $T(n_1)$ where $n_1 < n$ is the largest prime smaller than n. Is this recurrence defined over an ample set? ♢

Exercise 1.6: Consider inversions in a sequence of numbers.
(a) The sequence $S_0 = (1, 2, 3, 4)$ has no inversions, but sequence $S_1 = (2, 1, 4, 3)$ has two inversions, namely the pairs $\{1, 2\}$ and $\{3, 4\}$. Now, the sequence $S_2 = (2, 3, 1, 4)$ also has two inversions, namely the pairs $\{1, 2\}$ and $\{1, 3\}$. Let $I(S)$ be the number of inversions in S. Give an $O(n \lg n)$ algorithm to compute $I(S)$. Hint: this is a generalization of Mergesort.
(b) We next distinguish between the quality of the inversions of S_1 and S_2. The inversions $\{1, 2\}$ and $\{3, 4\}$ in S_1 are said to have weight of 1 each, so the **weighted inversion** of S_1 is $W(S_1) = 2 = 1 + 1$. But for S_2, the inversion $\{1, 2\}$ has weight 2 while inversion $\{1, 3\}$ has weight 1. So the weighted inversion is $W(S_2) = 3 = 2 + 1$. Thus the "weight" measures how far apart the two numbers are. In general, if $S = (a_1, \ldots, a_n)$ then a pair $\{a_i, a_j\}$ is an **inversion** if $i < j$ and $a_i > a_j$. The weight of this inversion is $j - i$. Let $W(S)$ be the sum of the weights of all inversions. Give an $O(n \lg n)$ algorithm for weighted inversions. ♢

End Exercises

§2. Divide-and-Conquer Algorithms

In this section, we see some other interesting recurrences that arise in a divide-and-conquer algorithms. First, we look at Karatsuba's classic algorithm for multiplying integers [7]. Then we consider a modern problem arising in searching for key words.

¶2. Example from Arithmetic. To motivate Karatsuba's algorithm, let us recall the classic "high-school algorithm" for multiplying integers. Given positive integers X, Y, we want to compute their product $Z = XY$. This algorithm assumes you know how to do single-digit multiplication and multi-digit additions ("pre-high school"). The algorithm multiples X by each digit of Y. If X and Y have n digits each, then we now have n products, each having at most $n + 1$ digits. After appropriate left-shifts of these n products, we add them all up. It is not hard to see that this algorithm takes $\Theta(n^2)$ time. Can we improve on this?

OK, you learned it in grade school

Usually we think of X, Y in decimal notation, but the algorithm works equally well in any base. We shall assume base 2 for simplicity. For instance, if $X = 19$ then in binary $X = 10011$. To avoid the ambiguity from different bases, we indicate[3] the base using a subscript, $X = (10011)_2$. The standard convention is that decimal base is assumed when no base is indicated. Thus a plain "100" without any base represents one hundred, and not four.

If we wanted four, we have to write "$(100)_2$"

Assume X and Y has length exactly n where n is a power of 2 (we can pad with 0's if necessary). Let us split up X into a high-order half X_1 and low-order half X_0. Thus

$$X = X_0 + 2^{n/2} X_1$$

[3]By the same token, we may write $X = (19)_{10}$ for base 10. But now the base "10" itself may be ambiguous – after all "10" in binary is equal to two. The convention is to write the base in decimal.

where X_0, X_1 are $n/2$-bit numbers. Similarly,

$$Y = Y_0 + 2^{n/2}Y_1.$$

Then

$$\begin{aligned} Z &= (X_0 + 2^{n/2}X_1)(Y_0 + 2^{n/2}Y_1) \\ &= X_0Y_0 + 2^{n/2}(X_1Y_0 + X_0Y_1) + 2^nX_1Y_1 \\ &= Z_0 + 2^{n/2}Z_1 + 2^nZ_2, \end{aligned}$$

where $Z_0 = X_0Y_0$, etc. Clearly, each of these Z_i's have at most $2n$ bits. Now, if we compute the 4 products

$$X_0Y_0, X_1Y_0, X_0Y_1, X_1Y_1$$

recursively, then we can put them together ("conquer step") in $\mathcal{O}(n)$ time. To see this, we must make an observation: in binary notation, multiplying any number X by 2^k (for any positive integer k) takes $\mathcal{O}(k)$ time, independent of X. We can view this as a matter of shifting left by k, or by appending a string of k zeros to X.

I bet you didn't learn this in high school

Hence, if $T(n)$ is the time to multiply two n-bit numbers, we obtain the recurrence

$$T(n) \le 4T(n/2) + Cn \tag{10}$$

for some $C > 1$. Given our simplification suggestions, we immediately rewrite this as

$$T(n) = 4T(n/2) + n.$$

As we will see, this recurrence has solution $T(n) = \Theta(n^2)$, so we have not really improved on the high-school method.

Karatsuba observed that we can proceed as follows: we can compute $Z_0 = X_0Y_0$ and $Z_2 = X_1Y_1$ first. Then we can compute Z_1 using the formula

$$Z_1 = (X_0 + X_1)(Y_0 + Y_1) - Z_0 - Z_2.$$

Thus Z_1 can be computed with one recursive multiplication plus some additional $\mathcal{O}(n)$ work. From Z_0, Z_1, Z_2, we can again obtain Z in $\mathcal{O}(n)$ time. This gives us the **Karatsuba recurrence**,

$$T(n) = 3T(n/2) + n. \tag{11}$$

We shall show that $T(n) = \Theta(n^\alpha)$ where $\alpha = \lg 3 = 1.58\cdots$. This is clearly an improvement of the high school method.

it is the first improvement in 2000 years

There is an even faster algorithm from Schönhage and Strassen (1971) that runs in $O(n \log n \log\log n)$ time. This has withstood improvements for almost 20 years, but in recent years, the $\log\log n$ factor has begun to be breached (they can be replaced by $\log^* n$). Many theoretical computer scientists believe that an $O(n \log n)$ algorithm should be possible. There is an increasing need for multiplication of arbitrarily large integers. In cryptography or computational number theory, for example. These are typically implemented in software in a "big integer" package. For instance, `Java` has a `BigInteger` class. A well-engineered big integer multiplication algorithm will typically implement the High-School algorithm for $n \le n_0$, and use Karatsuba for $n_0 < n \le n_1$, and use Schönhage-Strassen for $n > n_1$. Typical values for n_0, n_1 are $30, 200$.

¶3. A Google Problem. The Google Phenomenon is possible because of efficient algorithms: every files on the web can be searched and indexed. Searching is by keywords. Let us suppose that Google preprocesses every file in its database for keywords. However, a user may ask to search files for two or more keywords. We will reduce this this multi-keyword search to a precomputed single-keyword index.

Let F be a file, viewed as a sequence of words (ignoring punctuations, capitalization, etc). We first preprocess F for the occurrences of keywords. For each keyword w, we precompute a sorted sequence $P(w)$ of positions indicating where w occurs in F. E.g.,

$$P(divide) = (11, 16, 42, 101, 125, 767)$$

means that the keyword *divide* occurs 6 times in F, at positions $11, 16$, etc. Suppose we want to search the file using a conjunction of k keywords, $w_1, \ldots, w_k$. An interval $J = [s, t]$ is called a **cover** if each w_i occurs at least once within the positions in J. The size of a cover $[s, t]$ is just $t - s$. A cover is **minimal** if it is not contained in some larger cover; it is **minimum** if its size is smallest among all covers. Note that if $[s_i, t_i]$ are minimal covers for $i = 1, 2, \ldots$, and if $s_i < s_{i+1}$ then $t_i < t_{i+1}$. Our task is to compute a minimum cover.

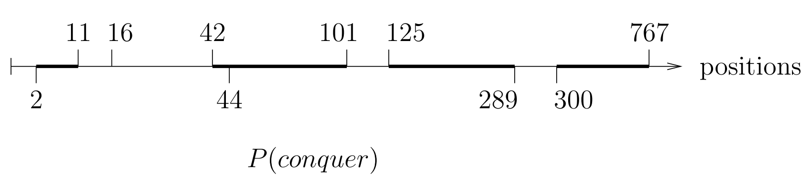

Figure 1: Minimal Covers

E.g., let $k = 2$ with $w_1 = divide$ and $w_2 = conquer$. With $P(divide)$ as before, let $P(conquer) = (2, 44, 289, 300)$. Then the minimal covers are $[2, 11], [42, 44], [44, 101], [125, 289], [289, 767]$. This is illustrated in Figure 1. The minimum cover is $[42, 44]$.

Let n_i be the length of list $P(w_i)$ $(i = 1, \ldots, k)$ and $n = n_1 + \cdots + n_k$. The case $k = 2$ is relatively straightforward, and we leave it for an exercise. Consider the case $k = 3$. First, merge $P(w_1), P(w_2), P(w_3)$ into the array $A[1..n]$. Recall that in Lecture I, we discussed the merging of sorted lists. Merging takes time $O(n_1 + n_2 + n_3) = O(n)$. To keep track of the origin of each number in A, we may also construct an array $B[1..n]$ such that $B[i] = j \in \{1, 2, 3\}$ iff $A[i]$ comes from the list $P(w_j)$.

We use a divide-and-conquer approach. Recursively, compute a minimum cover of $A[1..(n/2)]$ and $A[(n/2) + 1..n]$ (for simplicity, assume n is a power of 2). Let $C_{1,n/2}$ and $C_{(n/2)+1,n}$ be these minimum covers. We now need to find a minimal cover that straddles $A[(n/2)]$ and $A[(n/2) + 1]$. Let $C = [A[i], A[j]]$ be such a minimal cover, where $i \le (n/2)$ and $j \ge (n/2) + 1$. There are 6 cases. One case is when $C = C' \cup C''$, where $C' = [A[i], A[n/2]]$ is the rightmost cover for w_1 in $A[1..(n/2)]$, and $C'' = [A[(n/2) + 1], A[j]]$ is the leftmost cover for w_2, w_3 in $A[(n/2) + 1, n]$. We can find C' and C'' in $O(n)$ time. The remaining 5 cases can similarly be found in $O(n)$ time. Then C is the cover that has minimum size among these 6 cases. Hence, the overall complexity of the algorithm satisfies

$$T(n) = 2T(n/2) + n.$$

We have seen this recurrence before, as the Mergesort recurrence (2). The solution is $T(n) = \Theta(n \log n)$.

¶4. Master Recurrence and Divide-and-Conquer Algorithms. The recurrences (2) and (11) are instances of the **Master Recurrence** which has the form:

$$T(n) = aT(n/b) + f(n) \tag{12}$$

where $a > 0$ and $b > 1$ are constants and f is any function. We shall solve this recurrence under fairly general conditions.

The idea of solving a problem by reducing it to smaller subproblems is a very general one. In this chapter, we mainly focus on reductions from problems of size n to subproblems of size $\leq cn$ for some fixed $c < 1$. If there are a finite number of such subproblems, the running times can be bounded using solutions to the Master recurrence (72). In other problems, we reduce a problem of size n to several subproblems that of size $\leq n - c$ for some fixed $c \geq 1$. Such solutions would be exponential time without additional properties; we study these under the topic of dynamic programming (Chapter 7).

Exercises

Exercise 2.1: Carry out Karatsuba's algorithm for $X = 6 = (0110)_2$ and $Y = 11 = (1011)_2$. It is enough to display the recursion tree with the correct arguments for each recursive call, and the returned values. ♢

Exercise 2.2: Suppose an implementation of Karatsuba's algorithm achieves $T(n) \leq Cn^{1.58}$ where $C = 1000$. Moreover, the High School multiplication is $T(n) = 30n^2$. At what value of n does Karatsuba become competitive with the High School method? ♢

Exercise 2.3: Consider the recurrence $T(n) = 3T(n/2) + n$ and $T'(n) = 3T'(\lceil n/2 \rceil) + 2n$. Show that $T(n) = \Theta(T'(n))$. ♢

Exercise 2.4: The following is a programming exercise. It is best done using a programming language such as Java that has a readily available library of big integers.
(a) Implement Karatsuba's algorithm using such a programming language and using its big integer data structures and related facilities. The only restriction is that you must not use the multiplication, squaring, division or reciprocal facility of the library. But you are free to use its addition/subtraction operations, and any ability to perform left/right shifts (multiplication by powers of 2).
(b) Let us measure the running time of your implementation of Karatsuba's algorithm. For input numbers, use a random number generator to produce numbers of any desired bit length. If $T(n) \leq Cn^\alpha$ then $\lg T(n) \leq \lg C + \alpha \lg n$. The **exponent** α is thus the slope of the curve obtained by plotting $\lg T(n)$ against $\lg n$, we should get a slope of at most α. Plot the running time of your implementation to verify that its exponent is < 1.58.
(c) What is the exponent in Java's native implementation? Explain your data.
(d) My 1999 undergraduate class in algorithms did the preceding exercise, using the `java.math.BigInteger` package. One timing from this class is shown in Table 2. The "exponent" in this table is computing with a crude formula $\frac{\lg(avgTime) - avgTime_0}{\lg(numBits) - numBits_0}$ where $numBits_0 = 4000$ and $avgTime_0 = 4.358$ (the initial trial). This crude exponent hovers around 1.9. What would be the empirical exponent if you do a proper regression analysis? This data suggests that in 1999, the library only implemented the High School algorithm. By 2001, the situation appeared to have improved. ♢

NumBits	AvgTime	Exponent	NumBits	AvgTime	Exponent
4000	4.358	0.0	9600	23.034	1.9017905239616146
4200	4.696	1.531002145103799	9800	24.055	1.9064306092855452
4400	5.194	1.841260577604784	10000	24.986	1.905838802838669
4600	5.517	1.6873048110254347	10200	25.987	1.9074840762036238
4800	5.983	1.7381865504999572	10400	26.948	1.9067232067781992
5000	6.51	1.7985113947251763	10600	28.108	1.912700793571853
5200	6.988	1.7997159663026001	10800	29.111	1.9120055203582398
5400	7.509	1.812998128928515	11000	30.221	1.9143159996069712
5600	8.01	1.8089977665618309	11200	31.534	1.922120988851413
5800	8.684	1.85558837393382	11400	31.542	1.8898795547030012
6000	9.183	1.838236378924439	11600	32.67	1.8920105894497778
6200	9.769	1.8418523402197153	11800	33.703	1.8908891117429292
6400	10.365	1.8434357852847953	12000	34.67	1.8877101089855162
6600	11.088	1.864808884276074	12200	36.082	1.8955269064390694
6800	11.717	1.8638802969571109	12400	37.218	1.8956825843907563
7000	12.413	1.8704459319724756	12600	38.049	1.8884930574030907
7200	13.092	1.8714070696035303	12800	39.242	1.8894663931349043
7400	13.843	1.8787279477010768	13000	40.553	1.892493164635265
7600	14.532	1.8763458534440565	13200	41.696	1.8915733844170872
7800	15.297	1.8801860861195574	13400	42.951	1.8925738155123988
8000	16.054	1.8811947011507577	13600	44.159	1.8923271871808227
8200	16.905	1.8884383570994894	13800	45.533	1.8947617307075215
8400	17.644	1.8847717474449632	14000	46.816	1.8951803717241376
8600	18.498	1.8885827751677746	14200	48.1	1.8953182704475686
8800	19.283	1.8862283707110576	14400	49.401	1.8954588786790316
9000	20.225	1.8927722703240168	14600	50.873	1.8979435636574864
9200	21.17	1.8976522229154338	14800	52.364	1.9002856600816482
9400	22.063	1.8982439890258536	15000	53.537	1.8977482007273088

Figure 2: Timing as a function of number of bits

Exercise 2.5: Suppose the running time of an algorithm is an unknown function of the form $T(n) = An^a + Bn^b$ where $a > b$ and A, B are arbitrary positive constants. You want to discover the exponent a by measurement. How can you, by plotting the running time of the algorithm for various n, find a with an error of at most ϵ? Assume that you can do least squares line fitting. ♢

Exercise 2.6: Try to generalize Karatsuba's algorithm by breaking up each n-bit number into 3 parts. What recurrence can you achieve in your approach? Does your recurrence improve upon Karatsuba's exponent of $\lg 3 = 1.58\cdots$? ♢

Exercise 2.7: To generalize Karatsuba's algorithm, consider splitting an n-bit integer X into m equal parts (assuming m divides n). Let the parts be $X_0, X_1, \ldots, X_{m-1}$ where $X = \sum_{i=0}^{m-1} X_i 2^{in/m}$. Similarly, let $Y = \sum_{i=0}^{m-1} Y_i 2^{in/m}$. Let us define $Z_i = \sum_{j=0}^{i} X_j Y_{i-j}$ for $i = 0, 1, \ldots, 2m-2$. In the formula for Z_i, assume $X_\ell = Y_\ell = 0$ when $\ell \geq m$.
(i) Determine the Θ-order of $f(m,n)$, defined to be the time to compute the product $Z = XY$ when you are given $Z_0, Z_1, \ldots, Z_{2m-2}$. Remember that $f(m,n)$ is the number of bit operations.
(ii) It is known that we can compute $\{Z_0, Z_1, \ldots, Z_{2m-2}\}$ from the X_i's and Y_j's using $\mathcal{O}(m \log m)$ multiplications and $\mathcal{O}(m \log m)$ additions, all involving (n/m)-bit integers. Using this fact with part (i), give a recurrence relations for the time $T(n)$ to multiply two n-bit integers.
(iii) Conclude that for every $\varepsilon > 0$, there is an algorithm for multiplying any two n-bit integers in time $T(n) = \Theta(n^{1+\varepsilon})$. NOTE: part (iii) is best attempted after you have studied the Master Theorem in the subsequent sections. ♢

Exercise 2.8: Google[4] multi-keyword search.
(a) Solve the Google multi-keyword search for $k = 2$ in linear time.

[4] This problem was adapted from a Google interview question.

(b) Suppose $P(w_i) = (s_i, t_i)$ for $i = 1, \ldots, k$, i.e., each keyword has just two positions. Give an $O(k \log k)$ algorithm to find the minimum cover J for $w_1, \ldots, w_k$. $\diamondsuit$

Exercise 2.9: Write a program to solve the Google multi-keyword for the case $k = 3$ as described in the text. Use your favorite programming language (C or Java without any Object-Oriented fanfare is recommended). Initially, assume n is a power of 2. Indicate how to adapt your algorithm when n is not a power of 2. $\diamondsuit$

Exercise 2.10: Consider the following problem: we are given an array $A[1..n]$ of numbers, possibly with duplicates. Let $f(x)$ be the number of times ("frequency") a number x occurs. Given a number $k \geq 1$, we want to know whether there are k distinct numbers $x_1, \ldots, x_k$ such that $\sum_{i=1}^{k} f(x_i) > n/2$. Call $\{x_1, \ldots, x_k\}$ a **k-majority set**.
(a) Solve this decision problem for $k = 1$.
(b) Solve this decision problem for $k = 2$.
(c) Instead of the previous decision problem, we consider the optimization version: find the smallest k such that there are k numbers $x_1, \ldots, x_k$ with $\sum_{i=1}^{k} f(x_i) > n/2$. $\diamondsuit$

END EXERCISES

§3. Rote Method

We are going to introduce two "direct methods" for solving recurrences: rote method and induction. They are "direct" as opposed to other transformational methods which we will introduce later. Although fairly straightforward, these direct methods may call for some creativity (educated guesses). We begin with the rote method, as it appears to require somewhat less guess work.

¶5. Expand, Guess, Verify, Stop. The "rote method" is often thought of as the method of repeated expansion of a recurrence. Since such expansions can be done mechanically, this method has been characterized as rote. But in fact, expansions is only the first of 4 distinct stages. After several expansion steps, you guess the general term in the growing summation. Next, you verify your guess by natural induction. Finally, we must terminate the process by choosing a base of induction. The creative part of this process lies in the guessing step.

We will illustrate the method using the merge-sort recurrence (9):

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + n + n \\ &= 8T(n/8) + n + n + n \end{aligned} \tag{13}$$

This is the expansion step. At this point, we may guess that the $(i-1)$st step of this expansion yields

$$(G)_i : \quad T(n) = 2^i T(n/2^i) + in \tag{14}$$

for a general i. To verify our guess, we expand the guessed formula one more time,

$$\begin{aligned} T(n) &= 2^i[2T(n/2^{i+1}) + n/2^i] + in \\ &= 2^{i+1}T(n/2^{i+1}) + (i+1)n, \end{aligned} \tag{15}$$

which is just the formula $(G)_{i+1}$ in the sense of (14). Thus the formula (14) is verified for $i = 1, 2, 3, \ldots$. We must next choose a value of i at which to stop this expansion.

First consider the ideal situation: we simply choose $i = \lg n$. Then (14) yields $T(n) = 2^i T(n/2^i) + in = nT(1) + (\lg n)n$. Invoking DIC to make $T(1) = 0$, we obtain the solution $T(n) = n \lg n$. This is a beautiful solution, except for one problem: i must be an integer. It is meaningless, for instance, to expand the recurrence for $i = 2.3$ times. So we cannot use our old trick by pretending that i is a real variable (as we did for n).

So ideal case holds only when n is a power of 2, i.e., $n = 2^k$ for some integer $k \geq 0$. In general, we may choose an integer close to $\lg n$: $\lceil \lg n \rceil$ or $\lfloor \lg n \rfloor$ will do. Let us choose

$$i = \lfloor \lg n \rfloor \tag{16}$$

as our stopping value. With this choice, we obtain $1 \leq n/2^i < 2$. Under DIC, we can choose the initial condition to be

$$T(n) = 0, \qquad \text{for } n < 2. \tag{17}$$

This yields the *exact* solution that for $n \geq 2$,

$$T(n) = n \lfloor \lg n \rfloor. \tag{18}$$

To summarize, the rote method consists of

(E) Expansion steps as in (13),

(G) Guessing of a general formula as in (14),

(V) Verification of the formula as in (15),

(S) Stopping criteria choice as in (16).

But when the method works, it gives you the exact solution. How can this method fail? It is clear that you can always perform expansions, but you may be stuck at the next step while trying to guess a reasonable formula. For instance, try to expand the recurrence $T(n) = n + 2T(\lceil n/2 \rceil)$. In this case, we must give up exact solutions, and guess reasonable upper and/or lower bounds.

REMARKS:
I. The choice (17) is an application of DIC. But suppose you only use the weak DIC. Let us choose $n_0 = 1$ and $n_1 = 2$, so that for some C_0, C_1, we have

$$0 < C_0 \leq T(n) \leq C_1$$

for all $1 \leq n < 2$. In this case, we see that i must be chosen so that

$$\frac{n}{2^i} < 2 \leq \frac{n}{2^{i-1}}$$

which, after some manipulation, amounts to

$$i = 1 + \lfloor \lg(n/2) \rfloor.$$

Plugging into (14), we get that for $n \geq 2$,

$$\begin{aligned} T(n) &= 2^{1+\lfloor \lg(n/2) \rfloor} \Theta(1) + (1 + \lfloor \lg(n/2) \rfloor)n \\ &= n \lfloor \lg(n/2) \rfloor + \Theta(n). \end{aligned}$$

This is not as pretty as (18).
II. The appearance of the floor function in the solution (18) makes $T(n)$ non-continuous whenever n is a power of 2. We can make the solution continuous if we fully exploit our freedom in specifying boundary conditions. Let us now assume that $T(n) = n \lg n$ for $1 \le n < 2$. Then the above proof gives the solution

$$T(n) = n \lg n$$

for $n \ge 1$. This solution is the ultimate in simplicity for the recurrence (9).

Exercises

Exercise 3.1: No credit work: Rote is discredited word in pedagogy, so we would like a more dignified name for this method. We could call this the "4-Fold Path" or the the "EGVS Method". Suggest your own name for this method. In a humorous vein, what can EGVS stand for? ♢

Exercise 3.2: Solve the Karatsuba recurrence (11) using the Rote Method. ♢

Exercise 3.3: Use the Rote Method to solve the following recurrences
(a) $T(n) = n + 8T(n/2)$.
(b) $T(n) = n + 16T(n/4)$.

(c) Can you generalize your results in (a) and (b) to recurrences of the form $T(n) = n + aT(n/b)$ where a, b are in some special relation? ♢

Exercise 3.4: Give the exact solution for $T(n) = 2T(n/2)+n$ for $n \ge 1$ under the initial condition $T(n) = 0$ for $n < 1$. ♢

End Exercises

§4. Real Induction

The rote method, when it works, is a very sharp tool in the sense that as it gives us the exact solution to recurrences. Unfortunately, it does not work for many recurrences: while you can always expand, you may not be able to guess the general formula for the i-th expansion. We now introduce a more widely applicable method, based on the idea of "real induction".

To illustrate this idea, we use a simple example: consider the recurrence

$$T(x) = T(x/2) + T(x/3) + x. \tag{19}$$

The student is encouraged to attempt the rote method on this recurrence. Let us use real induction to prove an upper bound: suppose we guess that $T(x) \le Kx$ (ev.), for some $K > 1$. Then we verify it "inductively": Try rote first!

$$\begin{array}{rcll} T(x) & = & T(x/2) + T(x/3) + x & \text{(By definition)} \\ & \le & K\frac{x}{2} + K\frac{x}{3} + x & \text{(Inductive hypothesis)} \\ & = & Kx\left(\frac{1}{2} + K\frac{1}{3} + \frac{1}{K}\right) & \\ & = & Kx & \text{(Provided } K > 6/5) \end{array}$$

In the following, we will rigorously justify this method of proof.

How did we guess the upper bound $T(x) \le Kx$? What if we had guessed $T(x) \le Kx^2$? Well, we would have succeeded as well. In other words, this argument on confirms a particular guess; it does not tell us anything about the optimality of the guess. But actually, the proof can yield some hint on optimality. Finally, it is clear that we could also use real induction to confirm a guessed lower bound. The combined upper and lower bound can often lead to optimal bounds.

¶6. Natural Induction. Real induction is less familiar, so let us begin by recalling the related but well-known method of **natural induction**. The latter is a proof method based on induction over natural numbers. In brief, suppose $P(\cdot)$ is a natural number predicate, i.e., for each $n \in \mathbb{N}$, $P(n)$ is a proposition.

For example, $P(n)$ might be "There is a prime number between n and $n+10$ inclusive". A proposition is either true or false. Thus, we may verify[5] that $P(100)$ is true because 101 is prime, but $P(200)$ is false because 211 is the smallest prime larger than 200. A similar predicate is $P(n) \equiv$ "there is prime between n and $2n-1''$, called Bertrand's Postulate (1845).

We simply write "$P(n)$" or, for emphasis, "$P(n)$ holds" when we want to assert that "proposition $P(n)$ is true". Natural induction is aimed at proving propositions of the form

$$(\forall n \in \mathbb{N})[P(n) \text{ holds}]. \tag{20}$$

When (20) holds, we say the predicate $P(\cdot)$ is **valid**. For instance, Chebyshev proved in 1850 that Bertrand's Postulate $P(n)$ is valid. A "proof by natural induction" has three steps:
(i) [Natural Basis Step] Show that $P(0)$ holds.
(ii) [Natural Induction Step] Show that if $n \ge 1$ and $P(n-1)$ holds then $P(n)$ holds:

$$(n \ge 1) \wedge \ P(n-1) \Rightarrow P(n). \tag{21}$$

(iii) [Principle of Natural Induction] Invoke the principle of natural induction, which simply says that (i) and (ii) imply the validity of $P(\cdot)$, i.e., (20).

Since step (iii) is independent of the predicate $P(\cdot)$, we only need to show the first two steps. A variation of natural induction is the following: for any natural number predicate $P(\cdot)$, introduce a new predicate (the "star version of P") denoted $P^*(\cdot)$, defined via

$$P^*(n) : (\forall m \in \mathbb{N})[m < n \Rightarrow P(m)]. \tag{22}$$

The "Strong Natural Induction Step" replaces (21) in step (ii) by

$$(n \ge 1) \wedge \ P^*(n) \Rightarrow P(n). \tag{23}$$

It is easy to see that if we carry out the Natural Basis Step and the Strong Natural Induction Step, we have shown the validity of $P^*(n)$. Moreover, $P^*(\cdot)$ is valid iff $P(\cdot)$ is valid. Hence, a proof of the validity of $P^*(\cdot)$ is called a **strong natural induction proof** of the validity of $P(\cdot)$.

¶7. Real Induction. Now we introduce the real analogue of strong natural induction. Unlike natural induction, real induction is rarely discussed in standard mathematical literature, except possibly as a form of transfinite induction. Nevertheless, this topic holds interest in areas such as

[5] The smallest n such that $P(n)$ is false is $n = 114$.

program verification [2], timed logic [10], and real computational models [3]. We regard it is an important technique for analysis of algorithms.

Real induction is applicable to **real predicates**, *i.e.*, a predicate $P(\cdot)$ such that for each $x \in \mathbb{R}$, we have a proposition denoted $P(x)$. For example, suppose $T(x)$ is a total complexity function that satisfies the Karatsuba recurrence (11) subject to the initial condition $T(x) = 1$ for $x \leq 10$. Let us define the real predicate

$$P(x) : [x \geq 10 \Rightarrow T(x) \leq x^2]. \tag{24}$$

As in (20), we want to prove the **validity** of the real predicate $P(\cdot)$, i.e.,

$$(\forall x \in \mathbb{R})[P(x) \text{ holds}]. \tag{25}$$

In analogy to (22), we transform $P(\cdot)$ into a "star-version of P", defined as follows:

$$P^*_\delta(x) : (\forall y \in \mathbb{R})[y \leq x - \delta \Rightarrow P(y)] \tag{26}$$

where δ is any positive real number. The predicate $P^*_\delta(x)$ is called the **Real Induction Hypothesis**. When δ is understood, we may simply write $P^*(x)$ instead of $P^*_\delta(x)$.

Theorem 1 (Principle of Real Induction). *Let $P(x)$ be a real predicate. Suppose there exist real numbers $\delta > 0$ and x_1 such that*

(I) [Real Basis Step] *For all $x < x_1$, $P(x)$ holds.*

(II) [Real Induction Step] *For all $x \geq x_1$, $P^*_\delta(x) \Rightarrow P(x)$.*

Then $P(x)$ is valid: for all $x \in \mathbb{R}$, $P(x)$ holds.

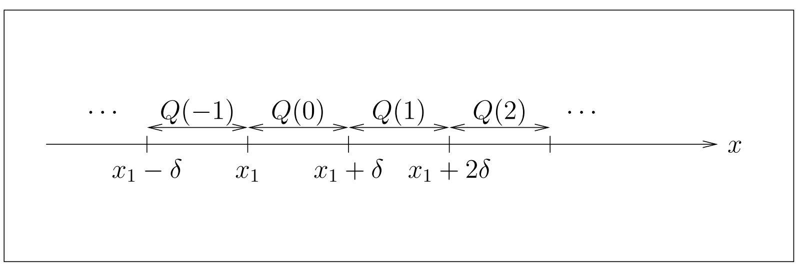

Figure 3: Discrete steps in real induction

Proof. The idea is to divide the real line into discrete intervals of length δ starting at x_1, using the function

$$t(x) := \left\lfloor \frac{x - x_1}{\delta} \right\rfloor.$$

Thus $t(x) < 0$ iff $x < x_1$. Also let the *integer* predicate $Q(\cdot)$ be given by

$$Q(n) : (\forall x \in \mathbb{R})[t(x) = n \Rightarrow P(x)].$$

Here n ranges over the integers, not just natural numbers. We then introduce

$$Q^*(n) : (\forall m \in \mathbb{Z})[m < n \Rightarrow Q(m)].$$

Note that $Q^*(0)$ is equivalent to the Real Basis Step. We claim that for all $n \in \mathbb{N}$,

$$Q^*(n) \Rightarrow Q(n). \tag{27}$$

To show this, fix any $n \in \mathbb{N}$, and any x satisfying $t(x) = n$. It suffices to show that $P(x)$ holds, assuming $Q^*(n)$. Note that for all $y \le x - \delta$,

$$t(y) = \left\lfloor \frac{y - x_1}{\delta} \right\rfloor \le \left\lfloor \frac{x - \delta - x_1}{\delta} \right\rfloor = t(x) - 1 = n - 1.$$

But $t(y) \le n-1$ and $Q^*(n)$ implies $P(y)$. Thus, we have established the Real Induction Hypothesis, $P^*_\delta(x)$. Also, $n = t(x) \ge 0$ means $x \ge x_1$. Hence the Real Induction Step (II) tells us that $P(x)$ holds. This proves our claim.

Now, (27) is equivalent to

$$Q^*(n) \Rightarrow Q^*(n+1). \tag{28}$$

If we view $Q^*(n)$ as a natural number predicate, then (28) is just the Natural Induction Step for the predicate $Q^*(\cdot)$. Then by the Principle of Natural Induction, we conclude that $Q^*(\cdot)$ is valid. The validity of $Q^*(\cdot)$ is equivalent to the validity of the real predicate $P(\cdot)$. **Q.E.D.**

Let us apply real induction to real recurrences. Note that its application requires the existence of two constants, x_1 and δ, making it somewhat harder to use than natural induction.

¶8. Example. Suppose $T(x)$ satisfies the recurrence

$$T(x) = x^5 + T(x/a) + T(x/b) \tag{29}$$

where $a \ge b > 1$. Given $x_0 \ge 1$ and $K > 0$, let $P(x)$ be the proposition

$$x \ge x_0 \Rightarrow T(x) \le Kx^5. \tag{30}$$

Define the constant $k_0 = a^{-5} + b^{-5}$. CLAIM: If $k_0 < 1$ then for all $x_0 \ge 1$, there is a $K > 0$ such that $P(x)$ is valid.

Proof: Now for any x_1, if $x_1 > x_0$ then our Default Initial Condition says that there is a $C > 0$ such that

$$T(x) \le C$$

for all $x_0 \le x < x_1$. If we choose K such that $K \ge C/x_0^5$ then for all $x_0 \le x < x_1$, we have $T(x) \le C \le Kx_0^5 \le Kx^5$ (since $x \ge x_0 \ge 1$). Hence $P(x)$ holds. This establishes the Real Basis Step (I) for $P(x)$ relative to x_1.

To establish the Real Induction Step (II), we need more properties for x_1 and must choose a suitable δ. First choose

$$x_1 = ax_0. \tag{31}$$

Thus for $x \ge x_1$, we have $x_0 \le x/a \le x/b$. Next choose

$$\delta = x_1 - (x_1/b) = x_1 \frac{b-1}{b}. \tag{32}$$

This ensures that for $x \ge x_1$, we have $x/a \le x/b \le x - \delta$. The Real Induction Hypothesis $P^*_\delta(x)$ says that for all $y \le x - \delta$, $P(y)$ holds, i.e., $y \ge x_0 \Rightarrow P(y)$. Suppose $x \ge x_1$ and $P^*_\delta(x)$ holds. We need to show that $P(x)$ holds:

$$\begin{aligned} T(x) &= x^5 + T(x/a) + T(x/b) \\ &\le x^5 + K\cdot(x/a)^5 + K\cdot(x/b)^5, \qquad \text{(by } P^*_\delta(x) \text{ and } x_0 \le x/a \le x/b \le x-\delta) \quad (33) \\ &= x^5(1 + K\cdot k_0) \\ &\le Kx^5 \quad (34) \end{aligned}$$

where the last inequality is true provided our choice of K above further satisfies $1 + K\cdot k_0 \le K$ or $K \ge 1/(1-k_0)$. This proves the Real Induction Step (II). Invoking the Principle of Real Induction, we conclude that $P(\cdot)$ is valid.

■

In a similar vein, we can use real induction to prove a lower bound: there is a constant $k > 0$ such that $T(x) \ge kx^5$ (ev.). Hence, we have shown $T(x) = \Theta(n^5)$ for the recurrence (29).

¶9. Automatic Real Induction. The last example shows that the direct application of the Principle of Real Induction is tedious, as we have to keep track of the constants such as δ, x_1 and K. Our goal is to prove a theorem which makes most of this process automatic. The property of the complexity functions used in the above derivation is captured by the following definition:

A real function $f : \mathbb{R}^k \to \mathbb{R}$ is said to be a **growth function** if f is eventually total, eventually non-decreasing and is unbounded in each of its variables. For instance, $f(x) = x^2 - 3x$ and $f(x, y) = x^y + x/\log x$ are growth functions, but $f(x) = -x$ and $f(x, y, z) = xy/z$ are not.

Theorem 2. *Assume $T(x)$ satisfies the real recurrence*

$$T(x) = G(x, T(g_1(x)), \ldots, T(g_k(x)))$$

and

- *$G(x, t_1, \ldots, t_k)$ and each $g_i(x)$ $(i = 1, \ldots, k)$ are growth functions.*
- *There is a constant $\delta > 0$ such that each $g_i(x) \le x - \delta$* (ev. x).

Suppose $f(x)$ is a growth function such that

$$G(x, Kf(g_1(x)), \ldots, Kf(g_k(x))) \le Kf(x)) \text{ (ev. } K, x). \quad (35)$$

Under the Default Initial Condition, we conclude

$$T(x) = \mathcal{O}(f(x)).$$

Proof. Pick $x_0 > 0$ and $K > 0$ large enough so that all the "eventual premises" of the theorem are satisfied. In particular, $f(x), G(x, t_1, \ldots, t_k)$ and $g_i(x)$ are all defined, non-decreasing and positive when their arguments are $\ge x_0$. Also, $g_i(x_0) \le x_0 - \delta$ for each i. Let $P(x)$ be the predicate

$$P(x) : \; x \ge x_0 \Rightarrow T(x) \le Kf(x).$$

Pick

$$x_1 = \max\{g_i^{-1}(x_0) : i = 1, \ldots, k\}. \quad (36)$$

The inverse g_i^{-1} of g_i is undefined at x_0 if there does not exist y_i such that $g_i(y_i) = x_0$, or if there exists more than one such y_i. In this case, take $g_i^{-1}(x_0)$ in (36) to be any y_i such that $g_i(y_i) \geq x_0$. We then conclude that for all $x \geq x_1$,

$$x_0 \leq g_i(x) \leq x - \delta.$$

By the Default Initial Condition (DIC), we conclude that for all $x \in [x_0, x_1]$, $P(x)$ holds. Thus, the Real Basis Step is verified. We now verify the Real Induction Step. Assume $x \geq x_1$ and $P_\delta^*(x)$. Then,

$$\begin{array}{rcll} T(x) & = & G(x, T(g_1(x)), \ldots, T(g_k(x))) & \\ & \leq & G(x, Kf(g_1(x), \ldots, Kf(g_1(x))) & (\text{by } P_\delta^*(x)) \\ & \leq & Kf(x) & (\text{by } (35)). \end{array}$$

Thus $P(x)$ holds. By the Principle of Real Induction, $P(x)$ is valid. This implies $T(x)\mathcal{O}(f(x))$.

Q.E.D.

To apply this theorem, the main property to verify is the inequality (35), since the other properties are usually routine to check. Let us see this in action on the example (29). We basically need to verify that

1. $f(x) = x^5$, $G(x, t_1, t_2) = x^5 + t_1 + t_2$, $g_1(x) = x/a$ and $g_2(x) = x/b$ are growth functions
2. $g_1(x) \leq x - 1$ and $g_2(x) \leq x - 1$ when x is large enough.
3. The inequality (35) holds when $K \geq 1/(1 - k_0)$. This is just the derivation of (34) from (33).

From theorem 2 we conclude that $T(x) = \mathcal{O}(f(x))$. The step (35) is the most interesting step of this derivation.

It is clear that we can give an analogous theorem which can be used to easily establish lower bounds on $T(x)$. We leave this as an Exercise.

REMARKS:

I. One phenomenon that arises is that one often has to introduce a stronger induction hypothesis than the actual result aimed for. For instance, to prove that $T(x) = \mathcal{O}(x \log x)$, we may need to guess that $T(x) = Cx \log x + Dx$ for some $C, D > 0$. See the Exercises below.

II. The **Archimedean Property** of real numbers says that for all $\delta > 0$ and $x > 0$, there exists $n \in \mathbb{N}$ such that $n\delta > x$. This is the property that allowed us to reduce Real Induction to Natural Induction.

III. A real predicate P can be identified with a subset S_P of $\mathbb{R}$ comprising those x such that $P(x)$ holds. The statement $P(x)$ can be generically viewed as asserting membership of x in S_P, *viz.*, "$x \in S_P$". Then a principle of real induction is just one that gives necessary conditions for a set S_P to be equal to $\mathbb{R}$. Similarly, a natural number predicate is just a subset of $\mathbb{N}$.

In the rest of this chapter, we indicate other systematic pathways; similar ideas are in lecture notes of Mishra and Siegel [11], the books of Knuth [8], Greene and Knuth [5]. See also Purdom and Brown [13] and the survey of Lueker [9].

Exercises

Exercise 4.1: Give another proof of theorem 1, by using contradiction. ♢

Exercise 4.2: Suppose $T(x) = 3T(x/2) + x$. Show by real induction that $T(x) = \Theta(x^{\lg 3})$. ♢

Exercise 4.3: Consider equation (9), $T(n) = 2T(n) + n$. Fix any $k > 1$. Show by induction that $T(n) = \mathcal{O}(n^k)$. Which part of your argument suggests to you that this solution is not tight? ♢

Exercise 4.4: Consider the recurrence $T(n) = n + 10T(n/3)$. Suppose we want to show $T(n) = \mathcal{O}(n^3)$.
(a) Attempting to prove by real induction, students often begin with a statement such as "Using the Default Initial Condition, we may assume that there is some $C > 0$ and $n_0 > 0$ such that $T(n) \le Cn^3$ for all $n < n_0$". What is wrong with this statement?
(b) Give a correct proof by real induction.
(c) Suppose $T(n) = n + 10T((n + K)/2)$ for some constant K. How does your proof in (b) change? ♢

Exercise 4.5: Let $T(n) = 2T(\frac{n}{2} + c) + n$ for some $c > 0$.
(a) By choosing suitable initial conditions, prove the following bounds on $T(n)$ by induction, and *not* by any other method:
(a.1) $T(n) \le D(n - 2c)\lg(n - 2c)$ for some $D > 1$. Is there a smallest D that depends only on c? Explain. Similarly, show $T(n) \ge D'(n - 2c)\lg(n - 2c)$ for some $D' > 0$.
(a.2) $T(n) = n \lg n - o(n)$.
(a.3) $T(n) = n \lg n + \Theta(n)$.
(b) Obtain the exact solution to $T(n)$.
(c) Use your solution to (b) to explain your answers to (a). ♢

Exercise 4.6: Generalize our principle of real induction so that the constant δ is replaced by a real function $\delta : \mathbb{R} \to \mathbb{R}_{>0}$. ♢

Exercise 4.7: (Gilles Dowek, "Preliminary Investigations on Induction over Real Numbers", manuscript 2002).
(a) A set $S \subseteq \mathbb{R}$ is closed if every limit point of S belongs to S. Let $P(x)$ be a real predicate $P(x)$. Assume $\{x \in \mathbb{R} : P(x)\text{holds}\}$ is a closed set. Suppose

$$P(a). \wedge .(\forall c \ge a)[P(c). \Rightarrow .(\exists\varepsilon)(\forall y)[c \le y \le c + \varepsilon \Rightarrow P(y)]]$$

Conclude that $(\forall x \ge a)P(x)$.
(b) Let $a, b \in \mathbb{R}$ and $\alpha, \beta : \mathbb{R} \to \mathbb{R}$ such that for all x, $\alpha(x) \ge 0$ and $\alpha(x) > 0$. Suppose f is a differentiable function satisfying

$$f(a) = bf'(x) = -\alpha(x)f(x) + \beta(x)$$

then for all $x \ge a$, $f(x) > 0$. Intuition: If $f(x)$ is the height of an object at time x, then the object will never reach the ground, *i.e.*, $f(x) > 0$. ♢

End Exercises

§5. Basic Sums

Consider the recurrence $T(n) = T(n-1) + n$. By rote method, this has the "solution"

$$T(n) = \sum_{i=1}^{n} i,$$

assuming $T(0) = 0$. But the RHS of this equation involves an **open sum**, meaning that the number of summands is unbounded as a function of n. We do not accept this "answer" even though it is correct.

¶10. What Does It Mean to Solve a Recurrence? The student may have noticed that the above open sum is well-known and is equal to

$$\binom{n+1}{2} = \frac{n(n+1)}{2} = \Theta(n^2).$$

Indeed, we would be perfectly happy with the answer "$T(n) = \Theta(n^2)$". In theory, one can always express a separable recurrence equation of $T(n)$ as an open sum, by rote expansion. We do not regard this as acceptable because we are really only interested in solutions which can be written as a **closed sum** or **product**, meaning that the number of summands (or factors in case of product) is independent of n. Moreover, each summand or factor must be a "familiar" function.

Do we know what we want?

¶11. Familiar Functions. So we conclude that "solving a recurrence" is relative to the form of solution we allow. This we interpret to mean a closed sum of "familiar" functions. For our purposes, the functions considered familiar include

polynomials $f(n) = n^k$, logarithms $f(n) = \log n$, and exponentials $f(n) = c^n$ $(c > 0)$.

I see! Solving means to relate to known functions

Functions such as factorials $n!$, binomial coefficients $\binom{n}{k}$ and harmonic numbers H_n (see below) are tightly bounded by familiar functions, and are therefore considered familiar. Finally, we have a rule saying that *the sum, product and functional composition of familiar functions are considered familiar*. Thus $\log^k n$, $\log\log n$, $n + 2\log n$ and $n^n \log n$ are familiar. For instance, let $f(n)$ be the number of ways an integer n can be written as the sum of two integers. Number theorists have shown that $f(n)$ is $(\log n)^{O(\log n)}$, which is familiar by our definition.

In addition to the above functions, two very slow growing functions arise naturally in algorithmic analysis. These are the log-star function $\log^* x$ and the inverse Ackermann function $\alpha(n)$ (see Lecture XII). We will consider them familiar, although functional compositions involving them are only familiar in our technical sense!

We refer the reader to the Appendix A in this lecture for basic properties of the exponential and logarithm function. In this section, we present some basic closed form summations.

Here are some basic facts that you should know of basic functions:

Lemma 3.
(i) For all $k < k'$, $n^k = \mathcal{O}(n^{k'})$ and $n^k \neq \Omega(n^{k'})$.
(ii) For all $k > 0$, $\lg n = \mathcal{O}(n^k)$ and $\lg n \neq \Omega(n^k)$.
(iii) For all k and all $c > 1$, $n^k = \mathcal{O}(c^n)$ and $n^k \neq \Omega(c^k)$.

We ask you to prove these in the exercises.

¶12. Arithmetic series. The basic arithmetic series is

$$\begin{aligned} S_n &:= \sum_{i=1}^{n} i \\ &= \binom{n+1}{2}. \end{aligned} \tag{37}$$

In proof,

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

More generally, for fixed $k \geq 1$, we have the "arithmetic series of order k",

$$S_n^k := \sum_{i=1}^{n} i^k = \Theta(n^{k+1}). \tag{38}$$

In proof, we have

$$n^{k+1} > S_n^k > \sum_{i=\lceil n/2 \rceil}^{n} (n/2)^k \geq (n/2)^{k+1}.$$

For more precise bounds, we bound S_n^k by integrals,

$$\frac{n^{k+1}}{k+1} = \int_0^n x^k dx < S_n^k < \int_1^{n+1} x^k dx = \frac{(n+1)^{k+1}-1}{k+1},$$

yielding

$$S_n^k = \frac{n^{k+1}}{k+1} + \mathcal{O}_k(n^k). \tag{39}$$

¶13. Geometric series. For $x \neq 1$ and $n \geq 1$,

$$\begin{aligned} S_n(x) &:= \sum_{i=0}^{n-1} x^i \\ &= \frac{x^n-1}{x-1}. \end{aligned} \tag{40}$$

In proof, note that $xS_n(x) - S_n(x) = x^n - 1$. Next, letting $n \to \infty$, we get the series

$$\begin{aligned} S_\infty(x) &:= \sum_{i=0}^{\infty} x^i \\ &= \begin{cases} \infty & \text{if } x \geq 1 \\ \uparrow \text{ (undefined)} & \text{if } x \leq -1 \\ \frac{1}{1-x} & \text{if } |x| < 1. \end{cases} \end{aligned}$$

Why is $S_\infty(-1)$ (say) considered undefined? For instance, writing

$$\begin{aligned} S_\infty(-1) &= 1-1+1-1+1-1+\cdots \\ &= (1-1)+(1-1)+(1-1)+\cdots \\ &= 0+0+0+\cdots, \end{aligned}$$

we conclude $S_\infty(-1) = 0$. But writing

$$\begin{aligned} S_\infty(-1) &= 1-1+1-1+1-\cdots \\ &= 1-(1-1)+(1-1)-\cdots \\ &= 1+0+0+\cdots, \end{aligned}$$

we conclude $S_\infty(-1) = 1$. So that we must consider this sum as having no definite value, i.e., undefined. Again,

$$\begin{aligned} S_\infty(-1) &= 1 - 1 + 1 - 1 + 1 - \cdots \\ &= 1 - S_\infty(-1), \end{aligned}$$

and we conclude that $S_\infty(-1) = 1/2$. In fact, there are infinitely many possible solutions for $S_\infty(-1)$.

Viewing x as a formal[6] variable, the simplest infinite series is $S_\infty(x) = \sum_{i=0}^{\infty} x^i$. It has a very simple closed form solution,

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}. \tag{41}$$

Viewed numerically, we may regard this solution as a special case of (40) when $n \to \infty$; but avoiding numerical arguments, it can be directly derived from the formal identity $S_\infty(x) = 1 + xS_\infty(x)$. We suggest calling $\sum_{i=0}^{\infty} x^i$ the "mother of series" because, from the formal solution to this series, we can derive solutions for many related series, including finite series. In fact, for $|x| < 1$, we can derive equation (40) by plugging equation (41) into

The one series to know!

$$S_n(x) = S_\infty(x) - x^n S_\infty(x) = (1 - x^n)S_\infty(x).$$

By differentiating both sides of the mother series with respect to x, we get:

$$\begin{aligned} \frac{1}{(1-x)^2} &= \sum_{i=1}^{\infty} i x^{i-1} \\ \frac{x}{(1-x)^2} &= \sum_{i=1}^{\infty} i x^i \end{aligned} \tag{42}$$

This process can be repeated to yield formulas for $\sum_{i=0}^{\infty} i^k x^i$, for any integer $k \geq 2$. Differentiating both sides of equation (40), we obtain the finite summation analogue:

$$\begin{aligned} \sum_{i=1}^{n-1} i x^{i-1} &= \frac{(n-1)x^n - nx^{n-1} + 1}{(x-1)^2}, \\ \sum_{i=1}^{n-1} i x^i &= \frac{(n-1)x^{n+1} - nx^n + x}{(x-1)^2}, \end{aligned} \tag{43}$$

(44)

Combining the infinite and finite summation formulas, equations (42) and (43), we also obtain

$$\sum_{i=n}^{\infty} i x^i = \frac{nx^n - (n-1)x^{n+1}}{(1-x)^2}. \tag{45}$$

We may verify by induction that these formulas actually hold for all $x \neq 1$ when the series are finite. In general, for any $k \geq 0$, we obtain formulas for the **geometric series of order** k:

$$\sum_{i=1}^{n-1} i^k x^i. \tag{46}$$

The infinite series have finite values only when $|x| < 1$.

[6] I.e., as an uninterpreted symbol rather than as a numerical value. Thereby, we avoid questions about the sum converging to some unique numerical value.

¶14. Harmonic series. This is defined for $n \geq 1$ as

$$H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}. \tag{47}$$

This is easy to see using calculus,

$$H_n < 1 + \int_1^n \frac{dx}{x} < 1 + H_n.$$

But $\int_1^n \frac{dx}{x} = \ln n$. This proves that

$$H_n = \ln n + g(n), \qquad \text{where } 0 < g(n) < 1. \tag{48}$$

Note that ln is the natural logarithm (appendix A). A generalization of (48) is this: for $0 < x \leq y$,

$$\sum_{x \leq n \leq y} \frac{1}{n} \leq \ln(y/x) + \min\{1, 1/x\}$$

where the summation is over all integers n in the interval $[x, y]$. There are more precise estimates for $g(n)$:

$$g(n) = \gamma + (2n)^{-1} + \mathcal{O}(n^{-2}) \tag{49}$$

where $\gamma = 0.577...$ is **Euler's constant**.

We can also deduce asymptotic properties of H_n without calculus: if $n = 2^N$, then

$$H_n = \sum_1 + \sum_2 + \cdots + \sum_N$$

where $\sum_k$ is defined as $\sum_{i=2^{k-1}}^{2^k-1} \frac{1}{i}$. Clearly,

$$1/2 = 2^{k-1}\frac{1}{2^k} < \sum_k \leq 2^{k-1}\frac{1}{2^{k-1}} = 1.$$

This proves that

$$(1/2)\lg(n) \leq H_n \leq \lg(n)$$

for n a power of 2. Extrapolating to all values of n, we conclude that $H_n = \Theta(\log n)$. This result also shows that H_n and $\lg(n)$ are unbounded.

For any real $\alpha \geq 1$, we can define the sum

$$H_n^{(\alpha)} := \sum_{i=0}^{n} \frac{1}{i^\alpha}.$$

Thus $H_n^{(1)}$ is just H_n. If we let $n = \infty$, the sum $H_\infty^{(\alpha)}$ is bounded for $\alpha > 1$; it is clearly unbounded for $\alpha = 1$ since $\ln n$ is unbounded. The sum is just the value of the Riemann zeta function at α. For instance, $H_\infty^{(2)} = \pi^2/6$. An exercise below estimates the sum $H_n^{(2)}$, and we see that a constant analogous to Euler's γ arises. Again, without calculus, we can prove that $H_n^{(\alpha)}$ is unbounded iff $\alpha \leq 1$.

Consistent with our policy of converting an integer recurrence into a real recurrence (see §1), it is useful to define the Harmonic numbers H_x where $x \geq 1$ is real. Sometimes, this is necessary after transformations of an integer recurrence. E.g., to solve the integer recurrence $T(n) = 2T(n/2) + (n/\lg n)$, we convert it to the standard form

$$t(N) = t(N-1) + 1/N \tag{50}$$

using the transformation $t(N) = T(2^N)/2^N$. We would like to say that the solution is H_N although $N = \lg n$ is not necessarily integer and so (50) is a real recurrence.

We define the **generalized Harmonic number**. For any real $x \geq 1$, define

$$\begin{aligned} H_x &:= \frac{1}{x} + \frac{1}{x-1} + \cdots + \frac{1}{x - \lfloor x \rfloor + 1} \\ &= \sum_{i=0}^{\lfloor x \rfloor - 1} \frac{1}{x-i}. \end{aligned}$$

Notice that we use $\lfloor x \rfloor$ instead of $\lceil x \rceil$ in the definition; this prevents the last term from blowing up. Of course, H_x agrees with the usual H_n when x is integer. Thus,

$$H_{\lfloor x \rfloor} \geq H_x \geq H_{\lceil x \rceil} - 1$$

and hence $|H_x - \ln x| < 1$. Clearly $H_x < H_{x+1}$. However, H_x is not monotonic increasing but rather has a "saw-tooth" shape.

Returning to recurrence (50), its solution is therefore $t(N) = H_N$, assuming that $t(x) = 0$ for all $x < 1$. Back solving, $T(n) = nH_{\lg n} = n(\ln \lg n + \mathcal{O}(1))$.

¶15. Stirling's Approximation. So far, we have treated open sums. If we have an open product such as the factorial function $n!$, we can convert it into an open sum by taking logarithms. This method of estimating an open product may not give as tight a bound as we wish (why?). For the factorial function, there is a family of more direct bounds that are collectively called **Stirling's approximation**. The following Stirling approximation is from Robbins (1955) and it may be committed to memory:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\, e^{\alpha_n}$$

where

$$\frac{1}{12n+1} < \alpha_n < \frac{1}{12n}.$$

Sometimes, the bound $\alpha_n > (12n)^{-1} - (360n^3)^{-1}$ is useful [4]. Up to Θ-order, we may prefer to simplify the above bound to

$$n! = \Theta\left(\left(\frac{n}{e}\right)^{n+\frac{1}{2}}\right).$$

¶16. Binomial theorem.

$$\begin{aligned} (1+x)^n &= 1 + nx + \frac{n(n-1)}{2}x^2 + \cdots + x^n \\ &= \sum_{i=0}^{n} \binom{n}{i} x^i. \end{aligned}$$

In general, the binomial function $\binom{x}{i}$ is defined for all real x and integer i:

$$\binom{x}{i} = \begin{cases} 0 & \text{if } i < 0 \\ 1 & \text{if } i = 0 \\ \frac{x(x-1)\cdots(x-i+1)}{i(i-1)\cdots 2\cdots 1} & \text{if } i > 0. \end{cases}$$

The binomial theorem can be viewed as an application of Taylor's expansion for a function $f(x)$ at $x = a$:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + \cdots$$

where $f^{(n)}(x) = \frac{d^n f}{dx^n}$. This expansion is defined provided all derivatives of f exist and the series converges. Applied to $f(x) = (1+x)^p$ for any real p at $x = 0$, we get

$$\begin{aligned}(1+x)^p &= 1 + px + \frac{p(p-1)}{2!}x^2 + \frac{p(p-1)(p-2)}{3!}x^3 + \cdots \\ &= \sum_{i\ge 0} \binom{p}{i} x^i.\end{aligned}$$

See [8, p. 56] for Abel's generalization of the binomial theorem.

Exercises

Exercise 5.1: Show Lemma 3. For logarithms, we want you to use direct inequalities (no calculus). ♢

Exercise 5.2: Solve the recurrence $T(x) = \frac{1}{x} + T(x-1)$ for all $x > 1$. ♢

Exercise 5.3: Let $c > 0$ be any real constant.
(a) Prove that $H_n = o(n^c)$. HINT: first let $c = 1$ and sum the first $\sqrt{n}$ terms of H_n/n.
(b) Show that $\ln(n+c) - \ln n = \mathcal{O}(c/n)$.
(c) Show that $|H_{x+c} - H_x| = \mathcal{O}(c/n)$ where H_x is the generalized Harmonic function.
(d) Bound the sum $\sum_{i=1+\lfloor c\rfloor}^{n} \frac{1}{i(i-c)}$. ♢

Exercise 5.4: Consider $S_\infty(x)$ as a numerical sum.
(a) Prove that there is a unique value for $S_\infty(x)$ when $|x| < 1$.
(b) Prove that there are infinitely many possible values for $S_\infty(x)$ when $x \le -1$.
(c) Are all real values possible as a solution to $S_\infty(-1)$? ♢

Exercise 5.5: Show the following useful estimate:

$$\ln(n) - (2/n) < \ln(n-1) < (\ln n) - (1/n).$$

♢

Exercise 5.6:
(a) Give the exact value of $\sum_{i=2}^{n} \frac{1}{i(i-1)}$. HINT: use partial fraction decomposition of $\frac{1}{i(i-1)}$.
(b) Conclude that $H_\infty^{(2)} \le 2$. ♢

Exercise 5.7: The goal is to give tight bounds for $H_n^{(2)} := \sum_{i=1}^{n} \frac{1}{i^2}$ (cf. previous exercise).
(a) Let $S(n) = \sum_{i=2}^{n} \frac{1}{(i-1)(i+1)}$. Find the exact bound for $S(n)$.
(b) Let $G(n) = S(n) - H_n^{(2)} + 1$. Now $\gamma' = G(\infty)$ is a real constant,

$$\gamma' = \frac{1}{1\cdot 3\cdot 4} + \frac{1}{2\cdot 4\cdot 9} + \frac{1}{3\cdot 5\cdot 16} + \cdots + \frac{1}{(i-1)\cdot(i+1)\cdot i^2} + \cdots.$$

Show that $G(n) = \gamma' - \theta(n^{-3})$.
(c) Give an approximate expression for $H_n^{(2)}$ (involving γ') that is accurate to $\mathcal{O}(n^{-3})$. Note that γ' plays a role similar to Euler's constant γ for harmonic numbers.
(d) What can you say about γ', given that $H_\infty^{(2)} = \pi^2/6$? Use a calculator (and a suitable approximation for π) to compute γ' to 6 significant digits. ♢

Exercise 5.8: Solve the recurrence $T(n) = 5T(n-1) + n$. ♢

Exercise 5.9: Solve exactly (choose your own initial conditions):
(a) $T(n) = 1 + \frac{n+1}{n}T(n-1)$.
(b) $T(n) = 1 + \frac{n+2}{n}T(n-1)$. ♢

Exercise 5.10: Show that $\sum_{i=1}^n H_i = (n+1)H_n - n$. More generally,

$$\sum_{i=1}^n \binom{i}{m} H_i = \binom{n+1}{m+1}\left[H_{n+1} - \frac{1}{m+1}\right].$$

♢

Exercise 5.11: Give a recurrence for S_n^k (see (38)) in terms of S_n^i, for $i < k$. Solve exactly for S_n^4. ♢

Exercise 5.12: Derive the formula for the "geometric series of order 2", $k = 2$ in (46). ♢

Exercise 5.13: (a) Use Stirling's approximation to give an estimate of the exponent E in the expression $2^E = \binom{2n}{n}$.
(b) (Feller) Show $\binom{2n}{n} = \sum_{k=0}^n \binom{n}{k}^2$. ♢

END EXERCISES

§6. Standard Form and Summation Techniques

We try to reduce all recurrences to the following **standard form**:

$$t(n) = t(n-1) + f(n). \tag{51}$$

Let us assume that the recurrence is valid for integers $n \geq 1$. Thus

$$t(i) - t(i-1) = f(i), \quad (i = 1, \ldots, n).$$

Adding these n equations together, all but two terms on the left-hand side cancel, leaving us $t(n) - t(0) = \sum_{i=1}^n f(i)$. (We say the left-hand side is a "telescoping sum", and this trick is known as "telescopy".) Choosing the convenient initial condition $t(n) = 0$ for $n < 0$, we obtain

$$t(n) = \sum_{i=0}^n f(i). \tag{52}$$

If this open sum has the form of one of the basic sums in the previous section, we are done! For instance, in bubble sort, we obtain a standard form recurrence:

$$t(n) = t(n-1) + n.$$

Choosing the initial condition $t(0) = 0$, we obtain the exact solution $t(n) = \sum_{i=1}^{n} i = \binom{n+1}{2}$.

In general, n is a real variable, and we must modify (52) as follows. Repeated expansions of the standard form recurrence throw out of the successive terms $f(n), f(n-1), f(n-2), \ldots, f(n-i), \ldots$ where i is an natural number. Where do we stop? A natural stopping place is when $i = \lfloor n \rfloor$. There is a notation[7] for the **fractional part** of n, defined as

$$\{n\} := n - \lfloor n \rfloor .$$

We have $0 \le \{n\} < 1$. Thus[8]

$$t(n) - t(\{n\} - 1) = \sum_{x=\{n\}}^{n} f(x). \tag{53}$$

Invoking DIC with $t(x) = 0$ for $x < 0$, we finally obtain

$$t(n) - t(\{n\} - 1) = \sum_{x=\{n\}}^{n} f(x). \tag{54}$$

¶17. Two Special Cases. Let us consider what is to be done if the open sum (52) does not easily reduce to one of the basic sums we have discussed. Note that integration is the continuous analogue of summation, so we can often estimate discrete sums by calculus. But we generally forbid the use of calculus. There are two common situations which we can solve. In the following, assume

No calculus please!

$$f : \mathbb{R} \to \mathbb{R}_{\ge 0}.$$

We want to estimate the sum in (52).

Polynomial Type: The function f is **polynomial-type** if f is non-decreasing increasing and

$$f(i) = \mathcal{O}(f(i/2)).$$

E.g.,

$$\sum_{i=1}^{n} i^3, \qquad \sum_{i=1}^{n} i \log i, \qquad \sum_{i=1}^{n} \log i \quad . \tag{55}$$

Exponential Type: The function f is **exponential-type** if it grows exponentially large or grows exponentially-small:

(a) f grows exponentially large if there exists $C > 1$ such that

$$f(i) \ge C \cdot f(i-1) \text{ (ev.)}.$$

E.g.,

$$\sum_{i=1}^{n} 2^i, \qquad \sum_{i=1}^{n} i^{-5} 2^{2^i}, \qquad \sum_{i=1}^{n} i! \quad . \tag{56}$$

[7] Alternatively, $\{n\}$ may be written as $n \,\mathbf{mod}\, 1$. Here **mod** is the operator corresponding to the relational $a \equiv b($ mod $m)$.

[8] Thus we generalize the summation notation to allow real limits: for real a, b we define $\sum_{x=a}^{b} f(x)$ to be the sum over all $f(x)$ where $x \in \{a + i : i \in \mathbb{N}, a + i \le b\}$.

(b) f grows exponentially small if there exists $0 < c < 1$ such that

$$f(i) \le c \cdot f(i-1) \text{ (ev.)}.$$

E.g.,

$$\sum_{i=1}^{n} 2^{-i}, \qquad \sum_{i=1}^{n} i^2 i^{-i}, \qquad \sum_{i=1}^{n} i^{-i} \quad . \tag{57}$$

According to the above classification of f, we call a summation $S_n = \sum_{i=1}^{n} f(i)$ a **polynomial-type** or an **exponential-type** sum.

Theorem 4 (Summation Rules). *Let* $S_n = \sum_{i=1}^{n} f(n)$.

1. *If* S_n *is a polynomial-type sum, replace* each term *by its largest term* $f(n)$ *to get a summation that has the same* Θ*-order. Hence* $S_n = \Theta(nf(n))$.
2. *If* S_n *is an exponential-type sum, replace the* entire sum *by its largest term to get a summation that has the same* Θ*-order:*
   (a) If $f(i)$ *grows exponentially large, the largest term is* $f(n)$*, and hence* $S_n = \Theta(f(n))$.
   (b) If $f(i)$ *grows exponentially small, the largest term is* $f(1)$*, and hence* $S_n = \Theta(f(1))$.

Proof. For a polynomial type sum, we get an upper bound

$$S_n \le \sum_{i=1}^{n} f(n) = nf(n).$$

Similarly, we get a lower bound

$$S_n \ge \sum_{i=\lfloor n/2 \rfloor}^{n} \Omega_1(f(\lfloor n/2 \rfloor)) = \Omega_1(nf(\lfloor n/2 \rfloor)).$$

The result follows since $f(\lfloor n/2 \rfloor) = \Theta(f(n))$. For an exponentially large sum, there is some $C > 1$ such that

$$f(n) \le S_n = f(n) + f(n-1) + f(n-2) + \cdots \le f(n)\left[1 + \frac{1}{C} + \frac{1}{C^2} + \cdots\right] < \frac{C}{C-1} f(n).$$

Similarly for an exponentially small sum, there is a $c < 1$ such that

$$f(1) \le S_n = f(1) + f(2) + f(3) + \cdots \le f(1)\left[1 + c + c^2 + \cdots\right] < f(1)\frac{1}{1-c}.$$

Q.E.D.

Let us illustrate this theorem on the examples given earlier:
Polynomial Sums. For $k > 0$,

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1}), \qquad \sum_{i=1}^{n} i \log i = \Theta(n^2 \log n), \qquad \sum_{i=1}^{n} \log i = \Theta(n \log n) \quad . \tag{58}$$

Exponentially Large Sums.

$$\sum_{i=1}^{n} 2^i = \Theta(2^n), \qquad \sum_{i=1}^{n} i^{-5} 2^{2^i} = \Theta(n^{-5} 2^{2^n}), \qquad \sum_{i=1}^{n} i! = \Theta(n!) \quad . \tag{59}$$

Exponentially Small Sums.

$$\sum_{i=1}^{n} 2^{-i} = \Theta(1), \qquad \sum_{i=1}^{n} i^2 i^{-i} = \Theta(1), \qquad \sum_{i=1}^{n} i^{-i} = \Theta(1) \quad . \tag{60}$$

Summation that does not fit the framework of Theorem 4 can sometimes be reduced to one that does. A trivial case is where summation we are interested in does not begin with $i = 1$. As another example, consider

$$S := \sum_{i=1}^{n} \frac{i!}{\lg^i n}, \tag{61}$$

which has terms depending on i as well as on the limit n. Write $S = \sum_{i=1}^{n} f(i,n)$ where

$$f(i,n) = \frac{i!}{\lg^i n}.$$

We note that $f(i,n)$ is growing exponentially for $i \ge 2\lg n$ (ev. n), since $f(i,n) = \frac{i}{\lg n} f(i-1,n) \ge 2f(i-1,n)$. Hence we may split the summation into two parts, $S = A + B$ where A comprise the terms for which $i \ge 2\lg n$ and B comprising the rest. Since B is an exponential sum, we have $B = \Theta(f(n,n))$. We can easily use Stirling's estimate for A to see that $A = \mathcal{O}(\log^{3/2} n) = \mathcal{O}(f(n,n))$. Thus $S = \Theta(f(n,n))$.

Another useful fact is the following:

Lemma 5. *Polynomial-type functions are closed under addition, multiplication, taking of logarithms, and raising to any constant power.*

Proof. If $f(n) \le Cf(n/2)$ and $g(n) \le Cg(n/2)$ then $f(n) + g(n) \le C(f(n/2) + g(n/2))$, $f(n)g(n) \le C^2 f(n/2)g(n/2)$, and $f(n)^k \le C^k f(n/2)^k$. Also $\log(f(n)) \le (\log C) + \log(f(n/2)) \le 2\log(f(n/2))$ (ev.). **Q.E.D.**

¶18. Grouping: Breaking Up into Small and Large Parts. The above example (61) illustrates the technique of breaking up a sum into two parts, one containing the "small terms" and the other containing the "big terms". This is motivated by the wish to apply different summation techniques for the 2 parts, and this in turn determines the cutoff point between small and big terms. Suppose we want to show

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = o(n).$$

It is sufficient to show that

$$S_n := H_n/n = \sum_{i=1}^{n} \frac{1}{i \cdot n}$$

goes to 0 as $n \to \infty$. Write $S_n = A_n + B_n$ where

$$A_n = \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} \frac{1}{i \cdot n}.$$

Thus, we choose $i = \sqrt{n}$ as the cutoff between the two parts. Then

$$A_n \le \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} \frac{1}{n} \le \frac{1}{\sqrt{n}}$$

and
$$B_n = \sum_{i=\lfloor\sqrt{n}\rfloor+1}^{n} \frac{1}{i\cdot n} \le \sum_{i=1}^{n} \frac{1}{\sqrt{n}\cdot n} = \frac{1}{\sqrt{n}}.$$
Thus $S_n \le \frac{2}{\sqrt{n}} \to 0$ as $n \to \infty$.

Exercises

Exercise 6.1: (a) Verify that the examples in (55), (56) and (57) are polynomial type or exponential type, as claimed.
(b) Is the summation $\sum_{i=1}^{n} i^{\lg i}$ an exponential type or polynomial type? Give bounds for the summation. ♢

Exercise 6.2: Let T_n be a complete binary tree with $n \ge 1$ nodes. So $n = 2^{h+1} - 1$ where h is the height of T_n. Suppose an algorithm has to visit all the nodes of T_n and at each node of height $i \ge 0$, expend $(i+1)^2$ units of work. Let $T(n)$ denote the total work expended by the algorithm at all the nodes. Give a tight upper and lower bounds on $T(n)$. ♢

Exercise 6.3: (a) Show that the summation $\sum_{i=2}^{n} (\lg n)^{\lg n}$ is neither polynomial-type nor exponential type.
(b) Estimate this sum. ♢

Exercise 6.4: For this problem, please use arguments from first principles. Do not use calculus, properties of $\log x$ such as $x/\log x \to \infty$, etc. (a) Show that $H_n = o(n^\alpha)$ for any $\alpha > 0$. HINT: Generalize the argument in the text.
(b) Likewise, show that $H_n \to \infty$ as $n \to \infty$. ♢

Exercise 6.5: Let $n = 2^k$. Show that $H_n = \theta(k)$ by grouping. Conclude that for all n, $H_n = \Theta(\lg n)$. ♢

Exercise 6.6: Use the method of grouping to show that $S(n) = \sum_{i=1}^{n} \frac{\lg i}{i}$ is $\Omega(\lg^2 n)$. ♢

Exercise 6.7: Give the Θ-order of the following sums:
(a) $S = \sum_{i=1}^{n} \sqrt{i}$.
(b) $S = \sum_{i=1}^{n} \lg(n/i)$. ♢

Exercise 6.8: Let $f(i) = f_n(i) = \frac{i-1}{n-i+1}$. The sum $F(n) = \sum_{i=1}^{n} f_n(i)$ is neither polynomial-type nor exponential-type. Give a Θ-order bound on $F(n)$. HINT: transform this into something familiar. ♢

Exercise 6.9: Can our summation rules for $S(n) = \sum_{i=1}^{n} f(i)$ be extended to the case where $f(i)$ is "decreasing polynomially", suitably defined? NOTE: such a definition must somehow distinguish between $f(i) = 1/i$ and $f(i) = 1/(i^2)$, since in one case $S(n)$ diverges and in the other it converges as $n \to \infty$. ♢

End Exercises

§7. Domain Transformation

So our goal for a general recurrence is to transform it into the standard form. You may think of change of domain as a "change of scale". Transforming the domain of a recurrence equation may sometimes bring it into standard form. Consider

$$T(N) = T(N/2) + N. \tag{62}$$

We define

$$t(n) := T(2^n), \quad N = 2^n.$$

This transforms the original N-domain into the n-domain. The new recurrence is now in standard form,

$$t(n) = t(n-1) + 2^n.$$

Choosing the boundary condition $t(0) = 1$, we get $t(n) = \sum_{i=0}^{n} 2^i$. This is a geometric series which we know how to sum, $t(n) = 2^{n+1} - 1$; hence, $T(N) = 2N - 1$.

Omit in a first reading

¶19. Logarithmic transform. More generally, consider the recurrence

$$T(N) = T\left(\frac{N}{c} - d\right) + F(N), \quad c > 1, \tag{63}$$

and d is an arbitrary constant. It is instructive to begin with the case $d = 0$. Then it is easy to see that the "logarithmic transformation" of the argument N to the new argument $n := \log_c(N)$ converts this to the new recurrence

$$t(n) = t(n-1) + F(c^n)$$

where we define

$$t(n) := T(c^n) = T(N).$$

There is possible confusion in such manipulations where we have used some implicit conventions. So let us state the connection between t and T more explicit. Let τ denote the **domain transformation function**,

$$\tau(N) = \log_c(N)$$

(so "n" is only a short-hand for "$\tau(N)$"). Then $t(\tau(N))$ is defined to be $T(N)$, valid for large enough N. In order for this to be well-defined, we need τ to have an inverse for large enough n. Then we can write

$$t(n) := T(\tau^{-1}(n)).$$

We now return to the general case where d is an arbitrary constant. Note that if $d < 0$ then we must assume that N is sufficiently large (how large?) so that the recurrence (63) is meaningful (*i.e.*, $(N/c) - d < N$). The following transformation

$$n := \tau(N) = \log_c(N + \frac{cd}{c-1})$$

will reduce the recurrence to standard from. To see this, note that the "inverse transformation" is

$$\begin{aligned} N &:= c^n - \frac{cd}{c-1} \\ &= \tau^{-1}(n) \\ (N/c) - d &= c^{n-1} - \frac{cd}{c-1} \\ &= \tau^{-1}(n-1). \end{aligned}$$

Writing $t(n)$ for $T(\tau^{-1}(n))$ and $f(n)$ for $F(\tau^{-1}(n))$, we convert equation (63) to

$$\begin{aligned} t(n) &= t(n-1) + F\left(c^n - \frac{cd}{c-1}\right) \\ &= t(n-1) + f(n) \\ &= \sum_{i=1}^{n} f(i). \end{aligned}$$

To finally "solve" for $t(n)$ we need to know more about the function $F(N)$. For example, if $F(N)$ is a polynomially bounded function, then $f(n) = F(c^n + \frac{cd}{c-1})$ would be $\Theta(F(c^n))$. This is the justification for ignoring the additive term "d" in the equation (63).

¶20. Division transform. Notice that the logarithmic transform case does not quite capture the following closely related recurrence

$$T(N) = T(N-d) + F(N), d > 0. \tag{64}$$

It is easy to concoct the necessary domain transformation: replace N by $n = N/d$ and substituting

$$t(n) = T(dn)$$

will transform it to the standard form,

$$t(n) = t(n-1) + F(dn).$$

Again, to be formal, we can explicitly introduce the transform function $\tau(N) = N/d$, etc. This may be called the "division transform".

¶21. General Pattern. In general, we consider $T(N) = T(r(N)) + F(N)$ where $r(N) < N$ is some function. We want a domain transform $n = \tau(N)$ so that

$$\tau(r(N)) = \tau(N) - 1. \tag{65}$$

For instance, if $r(N) = \sqrt{N}$ we may choose

$$\tau(N) = \lg\lg(N). \tag{66}$$

Then we see that

$$\tau(\sqrt{N}) = \lg(\lg(\sqrt{N})) = \lg(\lg(N)/2) = \lg\lg N - 1 = \tau(N) - 1.$$

Applying this transformation to the recurrence

$$T(N) = T(\sqrt{N}) + N, \tag{67}$$

we may define $t(n) := T(\tau^{-1}(n)) = T(2^{2^n}) = T(N)$, thereby transforming the recurrence (67) to to $t(n) = t(n-1) + 2^{2^n}$.

REMARKS:
1. The transformation (66) may be regarded as two applications of the logarithmic transform.
2. Domain transformation can be confusing because of the difficulty of keeping straight the similar-looking symbols, 'n' versus 'N' and 't' versus 'T'. Of course, these symbols are mnemonically chosen. When properly used, these conventions reduce clutter in our formulas. But if they are confusing, you can always fall back to the use of the explicit transformation functions such as τ.

Exercises

Exercise 7.1: Justify the simplification step (iv) in §1 (where we replace $\lceil n/2 \rceil$ by $n/2$). ♢

Exercise 7.2: Solve recurrence (63) in these cases:
(a) $F(N) = N^k$.
(b) $F(N) = \log N$. ♢

Exercise 7.3: Construct examples where you need to compose two or more of the above domain transformations. ♢

End Exercises

§8. Range Transformation

A transformation of the range is sometimes called for. For instance, consider

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

To put this into standard form, we could define

$$t(n) := \frac{T(n)}{2^n}$$

and get the standard form recurrence

$$t(n) = t(n-1) + \frac{n}{2^n}.$$

Telescoping gives us a series of the type in equation (42), which we know how to sum. Specifically, $t(n) = \sum_{\{n\}}^{n} \frac{n}{2^n} = \Theta(1)$. Hence $T(n) = \Theta(2^n)$.

We have transformed the range of $T(n)$ by introducing a multiplicative factor 2^n: this factor is called the **summation factor**. The reader familiar with linear differential equations will see an analogy with "integrating factor". (In the same spirit, the previous trick of domain transformation is simply a "change of variable".)

In general, a range transformation converts a recurrence of the form

$$T(n) = c_n T(n-1) + F(n) \tag{68}$$

into standard form. Here c_n is a constant depending on n. Let us discover which summation factor will work. If $C(n)$ is the summation factor, we get

$$t(n) := \frac{T(n)}{C(n)},$$

and hence

$$\begin{aligned}
t(n) &= \frac{T(n)}{C(n)} \\
&= \frac{c_n}{C(n)} T(n-1) + \frac{F(n)}{C(n)} \\
&= \frac{T(n-1)}{C(n-1)} + \frac{F(n)}{C(n)}, \qquad \text{(provided } C(n) = c_n C(n-1)\text{)} \\
&= t(n-1) + \frac{F(n)}{C(n)}.
\end{aligned}$$

Thus we need $C(n) = c_n C(n-1)$ which expands into

$$C(n) = c_n c_{n-1} \cdots c_1.$$

Exercises

Exercise 8.1: Solve the recurrence (68) in the case where $c_n = 1/n$ and $F(n) = 1$. ♢

Exercise 8.2: (a) Reduce the following recurrence

$$T(n) = 4T(n/2) + \frac{n^2}{\lg n}$$

to standard form. Then solve it exactly when n is a power of 2.
(b) Extend the solution of part(a) to general n using our generalized Harmonic numbers H_x for real $x \geq 2$ (see §2). You may choose any suitable initial conditions, but please state it explicitly.
(c) Solve the variations

$$T(n) = 4T(n/2) + \frac{n^2}{\lg^2 n}$$

and

$$T(n) = 4T(n/2) + \frac{n^2}{\sqrt{\lg n}}.$$

♢

Exercise 8.3: Repeat the previous question with the following recurrences:
(a) $T(n) = 4T(n/2) + \frac{n^2}{\lg^2 n}$
(b) $T(n) = 4T(n/2) + \frac{n^2}{\sqrt{\lg n}}$. ♢

End Exercises

§9. Differencing and QuickSort

Summation is the discrete analogue of integration. Extending this analogy, we introduce the **differencing** as the discrete analogue of differentiation. As expected, differencing is the inverse of summation. The differencing operation ∇ applied to any complexity function $T(n)$ yields another function ∇T defined by

$$(\nabla T)(n) = T(n) - T(n-1).$$

Differentiation often simplifies an equation: thus, $f(x) = x^2$ is simplified to the linear equation $(Df)(x) = 2x$, using the differential operator D. Similarly, differencing a recurrence equation for $T(n)$ may lead to a simpler recurrence for $(\nabla T)(n)$.

Indeed, the "standard form" (51) can be rewritten as

$$\nabla t(n) = f(n).$$

This is just an equation involving a difference operator – the discrete analogue of a differential equation.

For example, consider the recurrence

$$T(n) = n + \sum_{i=1}^{n-1} T(i).$$

This recurrence does not immediately yield to the previous techniques. But note that

$$(\nabla T)(n) = 1 + T(n-1).$$

Hence $T(n) - T(n-1) = 1 + T(n-1)$ and $T(n) = 2T(n-1) + 1$, which can be solved by the method of range transformation. (Solve it!)

¶22. QuickSort. A well-known application of differencing is the analysis of the QuickSort algorithm of Hoare. In QuickSort, we randomly pick a "pivot" element p. If p is the ith largest element, this subdivides the n input elements into $i-1$ elements less than p and $n-i$ elements greater than p. Then we recursively sort the subsets of size $i-1$ and $n-i$. The recurrence is

$$T(n) \quad = \quad n + \frac{1}{n}\sum_{i=0}^{n-1}(T(i-1) + T(n-i)), \tag{69}$$

since for each i, the probability that the two recursive subproblems in QuickSort are of sizes i and $n-i$ is $1/n$. The additive factor of "n" indicates the cost (up to a constant factor) to subdivide the subproblems, and there is no cost in "merging" the solutions to the subproblems. Simplifying (69),

$$\begin{array}{lcll}
T(n) & = & n + \frac{2}{n}\sum_{i=0}^{n-1} T(i) & \\
nT(n) & = & n^2 + 2\sum_{i=0}^{n-1} T(i) & \text{[Multiply by } n\text{]} \\
(n-1)T(n-1) & = & (n-1)^2 + 2\sum_{i=0}^{n-2} T(i) & \text{[Substitute } n \text{ by } n-1\text{]} \\
nT(n) - (n-1)T(n-1) & = & 2n - 1 + 2T(n-1) & \text{[Differencing operator for } nT(n)\text{]} \\
nT(n) & = & 2n - 1 + (n+1)T(n-1) & \text{[Simplify]} \\
\frac{T(n)}{n+1} & = & \frac{2}{n+1} - \frac{1}{n(n+1)} + \frac{T(n-1)}{n} & \text{[Divide by } n(n+1) \text{ (range transform)]} \\
t(n) & = & \frac{2}{n+1} - \frac{1}{n(n+1)} + t(n-1) & \text{[Define } t(n) = T(n)/(n+1)\text{]} \\
 & = & 2(H_{n+1} - 1) - \sum_{i=1}^{n} \frac{1}{i(i+1)} + t(0) & \text{[Telescoping a standard form]}
\end{array}$$

Thus we see that $t(n) \le 2H_{n+1}$ (assuming $t(0) = 0$) and hence we conclude

$$T(n) = 2n \ln n + \mathcal{O}(n).$$

It is also easy to get the exact solution for $t(n)$, by evaluating the sum $\sum_{i=1}^{n} \frac{1}{i(i+1)}$ (in a previous Exercise).

¶23. QuickSelect. The following recurrence is a variant of the QuickSort recurrence, and arises in the average case analysis of the QuickSelect algorithm:

$$T(n) = n + \frac{T(1) + T(2) + \cdots + T(n-1)}{n} \tag{70}$$

In the selection problem we need to "select the kth largest" where k is given (This problem is studied in more detail in Lecture XXX). Recursively, after splitting the input set into subsets of sizes $i-1$ and $n-i$ (as in QuickSort), we only need to continue one one of the two subsets (unless the pivot element is already the kth largest that we seek). This explains why, compared to (), the only change in (70) is to replace the constant factor of 2 to 1. To solve this, let us first multiply the equation by n (a range transform!). Then, on differencing, we obtain

$$\begin{aligned} nT(n) - (n-1)T(n-1) &= 2n - 1 + T(n-1) \\ nT(n) - nT(n-1) &= 2n - 1 \\ T(n) - T(n-1) &= 2 - \frac{1}{n} \\ T(n) &= 2n - \ln n + \Theta(1). \end{aligned}$$

Again, note that we essentially obtain an exact solution.

¶24. Improved QuickSort. We further improve the constants in QuickSort by first randomly choosing three elements, and picking the median of these three to be our pivot. The resulting recurrence is slightly more involved:

$$T(n) = n + \sum_{i=2}^{n-1} p_i [T(i-1) + T(n-i)] \tag{71}$$

where

$$p_i = \frac{(i-1)(n-i)}{\binom{n}{3}}$$

is the probability that the pivot element gives rise to subproblems of sizes $i-1$ and $n-i$.

EXERCISES

Exercise 9.1: Solve the following recurrences to Θ-order:

$$T(n) = n + \frac{2}{n} \sum_{i=\lfloor n/2 \rfloor}^{n-1} T(i).$$

HINT: Because of the upper bound $\lfloor n/2 \rfloor$, the function $\nabla T(n)$ has different behavior depending on whether n is even or odd. Simple differencing does not seem to work well here. Instead, we suggest the guess and verify-by-induction approach. ♢

Exercise 9.2: Generalize the previous question. Consider the recurrence

$$T(n) = n + \frac{c}{n} \sum_{i=1+\lfloor \alpha n \rfloor}^{n-1} T(i)$$

where $c > 0$ and $0 \le \alpha < 1$ are constants.
(a) Solve the recurrence for $c = 2$.
(b) Solve $T(n)$ when $c = 4$ and $\alpha = 0$.
(c) Fix $c = 4$. Determine the range of α such that $T(n) = \Theta(n)$. You need to argue why $T(n)$ is not $\Theta(n)$ for α outside this range.
(d) Determine the solution of this recurrence for general c, α. ♢

Exercise 9.3:
(a) Show that every polynomial $p(X)$ of degree d can be written as a sum of binomial coefficients with suitable coefficients c_i:

$$p(X) = c_d \binom{X}{d} + c_{d-1} \binom{X}{d-1} + \cdots + c_1 \binom{X}{1} + c_0.$$

(b) Assume the above form for $p(X)$, express $(\nabla p)(X)$ as a sum of binomial coefficients. HINT: what is $\nabla \binom{m}{n}$? ♢

End Exercises

§10. Examples

There is a wide variety of recurrences. This section looks at some recurrences, some of which falling outside our transformation techniques.

§10.1. Recurrences with Max

A class of recurrences that arises frequently in computer science involves the max operation. Fredman has investigated the solution of a class of recurrences involving max.

Consider the following variant of QuickSort: each time after we partition the problem into two subproblems, we will solve the subproblem that has the smaller size first (if their sizes are equal, it does not matter which order is used). We want to analyze the depth of the recursion stack. If a problem of size n is split into two subproblems of sizes n_1, n_2 then $n_1 + n_2 = n - 1$. Without loss of generality, let $n_1 \le n_2$. So $0 \le n_1 \le \lfloor (n-1)/2 \rfloor$. If the stack contains problems of sizes $(n_1 \ge n_2 \ge \cdots \ge n_k \ge 1)$ where n_k is the problem size at the top of the stack, then we have

$$n_{i-1} \ge n_i + n_{i+1}.$$

Since $n_1 \le n$, this easily implies $n_{2i+1} \le n/2^i$ or $k \le 2 \lg n$. A tighter bound is $k \le \log_\phi n$ where $\phi = 1.618\ldots$ is the golden ratio. This is not tight either.

The depth of recursion satisfies

$$D(n) = \max_{n_1=0}^{\lfloor (n-1)/2 \rfloor} [\max\{1 + D(n_1), D(n_2)\}]$$

This recurrence involving max is actually easy to solve. Assuming $D(n) \le D(m)$ for all $n \le m$, and for any real x, $D(x) = D(\lfloor x \rfloor)$, it is easy to see that $D(n) = 1 + D(n/2)$. Using the fact that $D(1) = 0$, we obtain $D(n) \le \lg n$. [Note: $D(1) = 0$ means that all problems on the stack has size ≥ 2.

§10.2. The Master Theorem

We first look at a recurrence that does fall under our transformation techniques: the **master recurrence** is

$$T(n) = aT(n/b) + f(n) \tag{72}$$

where $a > 0, b > 1$ are constants and $f(n)$ is some function.

We have already seen several instances of this recurrence. Another well-known one is Strassen's algorithm for multiplying two $n \times n$ matrices in subcubic time. Strassen's recurrence is $T(n) = 7T(n/2) + n^2$. Evidently, the Master recurrence is the recurrence to solve if we manage to solve a problem of size n by breaking it up into a subproblems each of size n/b, and merging these a sub-solutions in time $f(n)$. The recurrence was systematically studied by Bentley, Haken and Saxe [1]. Solving it requires a combination of domain and range transformation.

First apply a domain transformation by defining a new function $t(k)$ from $T(n)$: let

$$t(k) := T(b^k) \quad \text{(for all } k \in \mathbb{R}\text{)}.$$

Then (72) transforms into

$$t(k) = a\,t(k-1) + f(b^k).$$

Next, transform the range by using the summation factor $1/a^k$. This defines the function $s(k)$ from $t(k)$:

$$s(k) := t(k)/a^k.$$

Now $s(k)$ satisfies a recurrence in standard form:

$$\begin{aligned} s(k) &= \frac{t(k)}{a^k} \\ &= \frac{t(k-1)}{a^{k-1}} + \frac{f(b^k)}{a^k} \\ &= s(k-1) + \frac{f(b^k)}{a^k} \end{aligned}$$

Telescoping, we get

$$s(k) - s(\{k\}) = \sum_{i=\{k\}+1}^{k} \frac{f(b^i)}{a^i},$$

where $\{k\}$ is the fractional part of k (recall that k is real). Using the DIC, we chose the boundary condition $s(x) = 0$ for $x < 0$. Thus

$$s(k) = \sum_{i=\{k\}}^{k} \frac{f(b^i)}{a^i}.$$

Now, we cannot proceed any further without knowing the nature of the function f.

Let us call the function

$$W(n) = n^{\log_b a} \tag{73}$$

the **watershed function** for our recurrence, and $\log_b a$ the **watershed exponent**. The Master Theorem considers three cases for f. These cases are obtained by comparing f to $W(n)$. The easiest case is where f and W have the same Θ-order (CASE (0). The other two cases are where f grows "polynomially slower" (CASE (-1)) or "polynomially faster" (CASE $(+1)$) than the watershed function.

CASE (0) This is when $f(n)$ satisfies

$$f(n) = \Theta(n^{\log_b a}). \tag{74}$$

Then $f(b^i) = \Theta(a^i)$ and hence

$$s(k) = \sum_{i=1}^{k} f(b^i)/a^i = \Theta(k). \tag{75}$$

CASE (-1) This is when $f(n)$ **grows polynomially slower** than the watershed function:

$$f(n) = \mathcal{O}(n^{-\epsilon+\log_b a}), \tag{76}$$

for some $\epsilon > 0$. Then $f(b^i) = \mathcal{O}(b^{i(\log_b a - \epsilon)})$. Let $f(b^i) = \mathcal{O}_1(a^i b^{-i\epsilon})$ (using the subscripting notation for $\mathcal{O}$). So $s(k) = \sum_{i=1}^{k} f(b^i)/a^i = \sum \mathcal{O}_1(b^{-i\epsilon}) = \mathcal{O}_2(1)$, since $b > 1$ implies $b^{-\epsilon} < 1$. Hence

$$s(k) = \Theta(1). \tag{77}$$

CASE $(+1)$ This is when $f(n)$ satisfies the **regularity condition**

$$af(n/b) \le cf(n) \text{ (ev.)} \tag{78}$$

for some $c < 1$. Expanding this,

$$\begin{aligned} f(n) &\ge \frac{a}{c} f\left(\frac{n}{b}\right) \\ &\ge \left(\frac{a}{c}\right)^{\log_b n} f(1) \\ &= \Omega(n^{\epsilon+\log_b a}), \end{aligned}$$

where $\epsilon = -\log_b c > 0$. Thus the regularity condition implies that $f(n)$ **grows polynomially faster** than the watershed function,

$$f(n) = \Omega(n^{\epsilon+\log_b a}). \tag{79}$$

It follows from (78) that $f(b^{k-i}) \le (c/a)^i f(b^k)$. So

$$\begin{aligned} s(k) &= \sum_{i=1}^{k} f(b^i)/a^i \\ &= \sum_{i=0}^{k-1} f(b^{k-i})/a^{k-i} \\ &\le \sum_{i=0}^{k-1} (c/a)^i f(b^k)/a^{k-i} \\ &= f(b^k)/a^k \left(\sum_{i=0}^{k-1} c^{k-i}\right) \\ &= \mathcal{O}\left(\frac{f(b^k)}{a^k}\right), \end{aligned}$$

since $c < 1$. But clearly, $s(k) \geq f(b^k)/a^k$. Hence we have

$$s(k) = \Theta(f(b^k)/a^k). \tag{80}$$

Summarizing,

$$s(k) = \begin{cases} \Theta(1), & \text{CASE } (-1), \quad \text{see } (77), \\ \Theta(k), & \text{CASE } (0), \quad \text{see } (75), \\ \Theta(f(b^k)/a^k), & \text{CASE } (+1), \quad \text{see } (80). \end{cases}$$

Back substituting,

$$t(k) = a^k s(k) = \begin{cases} \Theta(a^k), & \text{CASE } (-1) \\ \Theta(a^k k), & \text{CASE } (0) \\ \Theta(f(b^k)), & \text{CASE } (+1). \end{cases}$$

Since $T(n) = t(\log_b n)$, we conclude:

THEOREM 6 (Master Theorem). *The master recurrence (72) has solution:*

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{if } f(n) = \mathcal{O}(n^{-\epsilon + \log_b a}), \text{ for some } \epsilon > 0, \\ \Theta(n^{\log_b a} \log n), & \text{if } f(n) = \Theta(n^{\log_b a}), \\ \Theta(f(n)), & \text{if } af(n/b) \leq cf(n) \text{ for some } c < 1. \end{cases}$$

In applications of the Master Theorem for case (+), we often first to verify equation (79) mentally, before checking the stronger regularity condition (78). The Master Theorem is powerful but unfortunately, there are gaps between its 3 cases. For instance, $f(n) = n^{\log_b a} \log n$ grows faster than the watershed function, but not polynomially faster. Thus the Master Theorem is inapplicable for this $f(n)$. Yet it is just as easy to solve this case using the transformation techniques (see Exercise).

In practice, the polynomial version of the theorem is most useful:

COROLLARY 7. *Let $a > 0, b > 1$ and k be constants. The solution to $T(n) = aT(n/b) + n^k$ is given by*

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{if } \log_b a > k \\ \Theta(n^k), & \text{if } \log_b a < k \\ \Theta(n^k \lg n), & \text{if } \log_b a = k \end{cases}$$

What if the values a, b in the master recurrence are not constants but depends on n? For instance, attempting to apply this theorem to the recurrence

$$T(n) = 2^n T(n/2) + n^n$$

(with $a = 2^n$ and $b = 2$), we obtain the false conclusion that $T(n) = \Theta(n^n \log n)$. See Exercises. The paper [15] treats the case $T(n) = a(n)T(b(n)) + f(n)$. For other generalizations of the master recurrence, see [14].

¶25. Graphic Interpretation of the Master Recurrence. We imagine a recursion tree with branching factor of a at each node, and every leaf of the tree is at level $\log_b a$. We further associate a "size" of n/b^i and "cost" of $f(n/b^i)$ to each node at level i (root is at level $i = 0$). Then $T(n)$ is just the sum of the costs at all the nodes. The Master Theorem says this: In case (0), the total cost associated with nodes at any level is $\Theta(n^{\log_b a})$ and there are $\log_b n$ levels giving an overall cost of $\Theta(n^{\log_b a} \log n)$. In case (+1), the cost associated with the root is $\Theta(T(n))$. In case (−1), the total cost associated with the leaves is $\Theta(T(n))$. Of course, this "recursion tree" is not realizable unless a and $\log_b a$ are integers. Hence the student should view this as a heuristic aid to remembering how the Master Theorem works.

Exercises

Exercise 10.1: Which is the faster growing function: $T_1(n)$ or $T_2(n)$ where

$$T_1(n) = 6T_1(n/2) + n^3, \qquad T_2(n) = 8T_2(n/2) + n^2.$$

◇

Exercise 10.2: State the solution, up to Θ-order of the following recurrences:

$$\begin{aligned} T(n) &= 10T(n/10) + \log^{10} n. \\ T(n) &= 100T(n/10) + n^{10}. \\ T(n) &= 10T(n/100) + (\log n)^{\log\log n}. \\ T(n) &= 16T(n/4) + 4^{\lg n}. \end{aligned}$$

◇

Exercise 10.3: Solve the following using the Master's theorem.
(a) $T(n) = 3T(n/25) + \log^3 n$
(b) $T(n) = 25T(n/3) + (n/\log n)^3$
(c) $T(n) = T(\sqrt{n}) + n$.
HINT: in the third problem, the Master theorem is applicable after a simple transformation.
◇

Exercise 10.4: Sometimes the Master Theorem is not applicable directly. But it can still be used to yield useful information. Use the Master Theorem to give as tight an upper and lower bound you can for the following recurrences:
(a) $T(n) = n^3 \log^3 n + 8T(n/2)$
(b) $T(n) = n^2/\log\log n + 9T(n/3)$
(c) $T(n) = 4T(n/2) + 3T(n/3) + n$. ◇

Exercise 10.5: Suppose $T(n) = n + 3T(n/2) + 2T(n/3)$. Joe claims that $T(n) = O(n)$, Jane claims that $T(n) = O(n^2)$, John claims that $T(n) = O(n^3)$. Who is closest to the truth? ◇

Exercise 10.6: We want to improve on Karatsuba's multiplication algorithm. We managed to subdivide a problem of size n into $a \geq 2$ subproblems of size $n/4$. After solving these a subproblems, we could combine their solutions in $O(n)$ time to get the solution to the original problem of size n. To beat Karatsuba, what is the maximum value a can have? ◇

Exercise 10.7: Suppose algorithm A_1 has running time satisfying the recurrence

$$T_1(n) = aT(n/2) + n$$

and algorithm A_2 has running time satisfying the recurrence

$$T_2(n) = 2aT(n/4) + n.$$

Here, $a > 0$ is a parameter which the designer of the algorithm can choose. Compare these two running times for various values of a. ♢

Exercise 10.8: Suppose

$$T_0(n) = 18T_0(n/6) + n^{1.5}$$

and

$$T_1(n) = 32T_1(n/8) + n^{1.5}.$$

Which is the correct relation: $T_0(n) = \Omega(T_1(n))$ or $T_0(n) = \mathcal{O}(T_1(n))$? We want you to do this exercise without using a calculator or its equivalent; instead, use inequalities such as $\log_8(x) < \log_6(x)$ (for $x > 1$) and $\log_6(2) < 1/2$. ♢

Exercise 10.9: How is the regularity condition on $f(n)$ and the condition that $f(n)$ increase polynomially related? What can you say about the sum $\sum_{i=1}^{n} f(i)$ when f satisfies the regularity condition for some a, b, c? ♢

Exercise 10.10: Solve the master recurrence when $f(n) = n^{\log_b a} \log^k n$, for any $k \geq 1$. ♢

Exercise 10.11: Show that the master theorem applies to the following variation of the master recurrence:

$$T(n) = a \cdot T(\frac{n+c}{b}) + f(n)$$

where $a > 0, b > 1$ and c is arbitrary. ♢

Exercise 10.12:
(a) Solve $T(n) = 2^n T(n/2) + n^n$ by direct expansion.
(b) To what extent can you generalize the Master theorem to handle some cases of $T(n) = a_n T(n/b_n) + f(n)$ where a_n, b_n are both functions of n? ♢

Exercise 10.13: Let $W(n)$ be the watershed function of the master recurrence. In what sense is the "watershed function" of the next order equal to $W(n)/\ln n$? ♢

Exercise 10.14:
(a) Let

$$s(n) = \sum_{i=1}^{n} \frac{\lg i}{i}$$

Prove that $s(n) = \Theta(\lg^2 n)$. For the lower bound, we want you to use real induction, and the fact that for $n \geq 2$, we have

$$\ln(n) - (2/n) < \ln(n-1) < (\ln n) - (1/n).$$

(b) Using the domain/range transformations to solve the following recurrence:

$$T(n) = 2T(n/2) + n\frac{\lg \lg n}{\lg n}.$$

♢

Exercise 10.15: Consider the recurrence $T(n) = aT(n/b) + \frac{n^4}{\log n}$ where $a > 0$ and $b > 1$. Describe the set S of all pairs (a, b) for which the Master Theorem gives a solution for this recurrence. Do not describe the solutions. You must describe the set S in the simplest possible terms. ♢

Exercise 10.16: The following recurrences arises in the analysis of a parallel algorithm for hidden-surface removal (Reif and Sen, Proc. ACM Symp. on Comp. Geometry, 1988):

$$T(n) = T(2n/3) + \lg n \lg\lg n$$

Another version of the algorithm [15] leads to

$$T(n) = T(2n/3) + (\lg n)/\lg\lg n.$$

Solve for $T(n)$ in both cases. ♢

End Exercises

§10.3. The Multiterm Master Theorem

The Master recurrence (72) can be generalized to the following **multiterm master recurrence**:

$$T(n) = f(n) + \sum_{i=1}^{k} a_i T\left(\frac{n}{b_i}\right) \tag{81}$$

where $k \geq 1$, $a_i > 0$ (for all $i = 1, \ldots, k$) and $b_1 > b_2 > \cdots > b_k > 1$. We give two examples:

$$\begin{aligned} T(n) &= T(c_1 n) + T(c_2 n) + n, \qquad (c_1 + c_2 < 1), && (82)\\ T(n) &= T(n/2) + T(n/4) + \log^7 n. && (83) \end{aligned}$$

The first recurrence (82) arise in linear time selection algorithms (see Chapter XI). There are many versions of this algorithm with different choices for the constants c_1, c_2. E.g., $c_1 = 7/10, c_2 = 1/5$. The second recurrence (83) arise in the so-called conjugate search trees in computational geometry (see Exercise 8.7).

Before we give the multiterm analogue of the Master Theorem, we generalize two concepts from the Master Theorem:
(a) Associated with the recurrence (81) is the **watershed constant**, a real number α such that

$$\sum_{i=1}^{k} \frac{a_i}{b_i^{\alpha}} = 1. \tag{84}$$

Clearly α exists and is unique since the summation tends to 0 as $\alpha \to \infty$, and tends to ∞ as $\alpha \to -\infty$. As usual, let $W(n) = n^\alpha$ denote the watershed function.
(b) The recurrence (81) gives rise to a **generalized regularity condition** on the driving function $f(n)$, namely,

$$\sum_{i=1}^{k} a_i f(n/b_i) \leq c f(n) \tag{85}$$

for some $0 < c < 1$.

Theorem 8 (Multiterm Master Theorem).

$$T(n) = \begin{cases} \Theta(n^\alpha \log n) & \text{if } f(n) = \Theta(n^\alpha) \\ \Theta(n^\alpha) & \text{if } f(n) = \mathcal{O}(n^{\alpha-\varepsilon}), \text{for some } \varepsilon > 0, \\ \Theta(f(n)) & \text{if } f \text{ satisfies the regularity condition (85).} \end{cases}$$

Proof. The proof uses real induction.

CASE (0): Assume that $f(n) = \Theta_1(W(n))$. We will show that $T(n) = \Theta_2(W(n)\log n)$. We have

$$\begin{aligned} T(n) &= f(n) + \textstyle\sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \\ &= \Theta_1(n^\alpha) + \textstyle\sum_{i=1}^k a_i \Theta_2\left(\left(\frac{n}{b_i}\right)^\alpha \log\left(\frac{n}{b_i}\right)\right) && \text{(by induction)} \\ &= \Theta_1(n^\alpha) + \Theta_2(n^\alpha)\left[\textstyle\sum_{i=1}^k \frac{a_i}{b_i^\alpha}\log\left(\frac{n}{b_i}\right)\right] \\ &= \Theta_1(n^\alpha) + \Theta_2(n^\alpha)\left[\log n - D\right], && \text{(where } D = \textstyle\sum_{i=1}^k \frac{a_i}{b_i^\alpha}\log(b_i) \text{ and using (84))} \\ &= \Theta_2(n^\alpha \log n). \end{aligned}$$

Let us elaborate on the last equality. Suppose $f(n) = \Theta_1(n^\alpha)$ amounts to the inequalities $c_1 W(n) \le f(n) \le C_1 W(n)$ (ev.). We must choose c_2, C_2 such that $c_2 W(n)\log n \le T(n) \le C_2 W(n)\log n$ (ev.). The following choice suffices:

$$C_2 = C_1/D, \qquad c_2 = c_1/D.$$

CASE (-1): Assume $0 \le f(n) \le D_1 n^{\alpha-\varepsilon}$ for some $\varepsilon > 0$. The lower bound is easy: assume $T(n/b_i) \ge c_1(n/b_i)^\alpha$ (ev.) for each i. Then[9]

$$\begin{aligned} T(n) &= f(n) + \textstyle\sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \\ &\ge \textstyle\sum_{i=1}^k a_i c_1 (\frac{n}{b_i})^\alpha && \text{(since } f(n) \ge 0 \text{ and by induction)} \\ &= c_1 n^\alpha. \end{aligned}$$

The upper bound needs a slightly stronger hypothesis: assume $T(n/b_i) \le C_1 n^\alpha(1 - n^{-\varepsilon})$ (ev.). Then

$$\begin{aligned} T(n) &= f(n) + \textstyle\sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \\ &\le D_1 n^{\alpha-\varepsilon} + \textstyle\sum_{i=1}^k a_i C_1 \left(\frac{n}{b_i}\right)^\alpha \left[1 - \left(\frac{n}{b_i}\right)^{-\varepsilon}\right] && \text{(by induction)} \\ &= C_1 n^\alpha - C_1 n^{\alpha-\varepsilon}\left[\textstyle\sum_{i=1}^k \frac{a_i}{b_i^{\alpha-\varepsilon}} - D_1/C_1\right] \\ &\le C_1 n^\alpha - C_1 n^{\alpha-\varepsilon} \end{aligned}$$

provided $\sum_{i=1}^k a_i/b_i^{\alpha-\varepsilon} \ge 1 + (D_1/C_1)$. Since $\sum_{i=1}^k a_i/b_i^{\alpha-\varepsilon} > 1$, we can certainly choose a large enough C_1 to satisfy this.

CASE $(+1)$: The lower bound $T(n) = \Omega(f(n))$ is trivial. As for upper bound, assuming $T(m) \le D_1 f(m)$ (ev.) whenever $m = n/b_i$,

$$\begin{aligned} T(n) &= f(n) + \textstyle\sum_{i=1}^k a_i T\left(\frac{n}{b_i}\right) \\ &\le f(n) + \textstyle\sum_{i=1}^k a_i D_1 f(n/b_i) && \text{(by induction)} \\ &= f(n) + D_1 c f(n) && \text{(by regularity)} \\ &\le D_1 f(n) && \text{(if } D_1 \ge 1/(1-c)\text{)} \end{aligned}$$

Q.E.D.

[9] The fact $f(n) \ge 0$ (ev.) is a consequence of "$f \in \mathcal{O}(n^{\alpha-\varepsilon})$" and the definition of the big-Oh notation.

The use of real induction appears to be necessary in this proof: unlike the master recurrence, the multiterm version does not yield to transformations. Again, the generalized regularity condition implies that $f(n) = \Omega(n^{\alpha+\varepsilon})$ for some $\varepsilon > 0$. This is shown by induction:

$$\begin{array}{rcll} f(n) & \geq & \frac{1}{c}\sum_{i=1}^{k} a_i f(n/b_i) & \\ & \geq & \frac{1}{c}\sum_{i=1}^{k} a_i D(n/b_i)^{\alpha+\varepsilon} & \text{(by induction, for some } D > 0) \\ & = & \frac{D}{c} n^{\alpha+\varepsilon} \sum_{i=1}^{k} \frac{a_i}{b_i^{\alpha+\varepsilon}} & \\ & = & Dn^{\alpha+\varepsilon} & \text{(if we choose } c = \sum_{i=1}^{k} \frac{a_i}{b_i^{\alpha+\varepsilon}}) \end{array}$$

Since $\sum_{i=1}^{k} \frac{a_i}{b_i^{\alpha}} = 1$, we should be able to choose a $\varepsilon > 0$ to satisfy the last condition. Note that this derivation imposes no condition on D, and so D can be determined based on the initial conditions.

Let us apply the theorem to the recurrence for $T_1(n)$ in the selection problem (82) and $T_2(n)$ in the conjugate tree problem (83). For (82), we see that $\alpha < 1$ and since the regularity condition holds for the function $f(n) = n$, we conclude that $T_1(n) = \Theta(n)$. For (83), we may use a calculator to verify that the watershed value is $\alpha = 0.694\ldots$. Since $f(n) = \mathcal{O}(n^{\alpha-\varepsilon})$, we conclude that $T_2(n) = \Theta(n^{0.694\cdots})$.

Exercises

Exercise 10.17: The following recurrence arises in the analysis of the running time of the "conjugation tree" in computational geometry:

$$T(n) = T(n/2) + T(n/4) + \lg^7 n.$$

Solve for $T(n)$. ♢

End Exercises

§11. Orders of Growth

The reader should first review the basic properties of the exponential and logarithm functions in the appendix.

Learning to judge the growth rates of complexity functions is a fundamental skill in algorithmics. This section is a practical one, designed to help students develop this skill.

Most complexity functions in practice are the so-called **logarithmico-exponential functions** (for short, L-functions): such functions $f(x)$ are real and defined for all $x \geq x_0$ for some x_0 depending of f. An L-function is either the identity function x or a constant $c \in \mathbb{R}$, or else obtained as a finite composition with the functions

$$A(x), \qquad \ln(x), \qquad e^x$$

where $A(x)$ denotes a real branch of an algebraical function. For instance, $A(x) = \sqrt{x}$ is the function that picks the real square-root of x. The reader may have noticed that all the common complexity functions are totally ordered in the sense that for any f, g, either $f \preceq g$ or $g \preceq f$. A theorem[10] of Hardy [6] confirms this: *if f and g are L-functions then* $f \le g$ (ev.) *or* $g \le f$ (ev.). In particular, each L-function f is eventually non-negative, $0 \le f$ (ev.), or non-positive, $f \le 0$ (ev.).

The following are the common categories of functions you will encounter:

CATEGORY	SYMBOL	EXAMPLES
vanishing term	$o(1)$	$\frac{1}{n}$, 2^{-n}
constants	$\Theta(1)$	1, $2-\frac{1}{n}$
polylogs	$\log^k n$ (for any $k > 0$)	H_n, $\log^2 n$
polynomials	n^k (for any $k > 0$)	n^3, $\sqrt{n}$
non-polynomials	$n^{\Omega(1)}$	$n!$, 2^n, $n^{\log\log n}$

Note that $n!$ and H_n are not L-functions, but they can be closely approximated by L-functions. The last category forms a grab-bag of anything growing faster than a polynomial. These 5 categories form a hierarchy of increasingly larger Θ-order.

¶26. Rules for comparing functions. We are interested in comparing functions up to their Θ-order. The trick of comparing two functions by taking their logarithms is this: *if* $\log f \preceq \log g$ *then clearly* $f \preceq g$. But students often think the converse is also true.

We list some simple rules. Most comparisons of interest to us can be reduced to repeated applications of these rules:

Sum: In a direct comparison involving a sum $f(n) + g(n)$, ignore the smaller term in this sum. E.g., given $n^2 + n\log n + 5$, you should ignore the "$n \log n + 5$" term. However, beware that if the sum appears in an exponent, the neglected part may turn out be decisive when the dominant terms are identical.

Product: If $0 \preceq f \preceq f'$ and $0 \preceq g \preceq g'$ then $fg \preceq f'g'$. (If, in addition, $f \prec f'$ then we have $fg \prec f'g'$.)
E.g., this rule implies $n^b \prec n^c$ when $b < c$ (since $1 \prec n^{c-b}$, by the logarithm rule next).

Logarithm: $1 \prec \log^{(k+1)} n \prec (\log^{(k)} n)^c$ for any integer $k \ge 0$ and real $c > 0$. Here $\log^{(k)} n$ refers to the k-fold application of the logarithm function and $\log^{(0)} n = n$.

Exponentiation: If $1 \le f \le g$ (ev.) then $d^f \preceq d^g$ for any constant $d > 1$. If $1 \le f \le cg$ (ev.) for some $c < 1$ then $d^f \prec d^g$.

¶27. Example. Suppose we want to compare $n^{\log n}$ versus $(\log n)^n$. By the rule of exponentiation, $n^{\log n} \prec (\log n)^n$ follows if we take logs and show that $\log^2 n \le 0.5 n \log\log n$ (ev.). In fact, we show the stronger $\log^2 n \prec n \log\log n$. Taking logs again, and by the rule of sum, it is sufficient to show $2\log\log n \prec \log n$. Taking logs again, and by the rule of sum again, it is suffices to show $\log^{(3)} n \prec \log^{(2)} n$. But the latter follows from the rule of logarithms.

Exercises

[10] In the literature on L-functions, the notation "$f \preceq g$" actually means $f \le g$ (ev.). There is a deep theory involving such functions, with connection to Nevanlinna theory.

Exercise 11.1: (i) Simplify the following expressions: (a) $n^{1/\lg n}$, (b) $2^{2^{\lg\lg n-1}}$, (c) $\sum_{i=0}^{k-1} 2^i$, (d) $2^{(\lg n)^2}$, (e) $4^{\lg n}$, (f) $(\sqrt{2})^{\lg n}$.
(ii) Re-do the above, replacing each occurrence of "2" (explicit or otherwise) in the previous expressions by some constant $c > 2$. ♢

Exercise 11.2: Order these in increasing big-Oh order:

$$n\lg n, \quad n^{-1}, \quad \lg n, \quad n^{\lg n}, \quad 10n + n^{3/2}, \quad \pi^n, \quad 2^n, \quad 2^{\lg n}.$$

♢

Exercise 11.3: Order the following 5 functions in order of increasing Θ-order: (a) $\log^2 n$, (b) $n/\log^4 n$, (c) $\sqrt{n}$, (d) $n2^{-n}$, (e) $\log\log n$. ♢

Exercise 11.4: Order the following functions (be sure to parse these nested exponentiations correctly): (a) $n^{(\lg n)^{\lg n}}$, (b) $(\lg n)^{n^{\lg n}}$, (c) $(\lg n)^{(\lg n)^n}$, (d) $(n/\lg n)^{n^{n/(\lg n)}}$. (e) $n^{n^{(\lg n)/n}}$. ♢

Exercise 11.5: Order the following set of 36 functions in non-increasing order of growth. Between consecutive pairs of functions, insert the appropriate ordering relationship: $\preceq$, $\asymp$, $\le$ (ev.), $=$.

	a	b	c	d	e	f
1.	$\lg\lg n$	$(\lg n)^{\lg n}$	2^n	$2^{\lg n}$	$2^{\lg^* n}$	$2^{2^{n+1}}$
2.	$(1/3)^n$	$n2^n$	$n^{\lg\lg n}$	e^n	$n^{1/\lg n}$	$\lceil\lg n\rceil!$
3.	$2^{\sqrt{2\lg n}}$	$(3/2)^n$	2	$\lg(n!)$	n	$\sqrt{\lg n}$
4.	$2^{(\lg n)^2}$	2^{2^n}	n^2	$n\lg n$	$(n+1)!$	$4^{\lg n}$
5.	$\lg(\lg^* n)$	$\lg^2 n$	$(1+\frac{1}{n})^n$	$n^{\lg n}$	$n!$	$2^{(\lg n)/n}$
6.	$(\sqrt{2})^{\lg n}$	$\lg^* n$	$(n/\lg n)^2$	$\sqrt{n})$	$\lg^*(\lg n)$	$1/n$

NOTE: to organize of this large list of functions, we ask that you first order each row. Then the rows are merged in pairs. Finally, perform a 3-way merge of the 3 lists. Show the intermediate lists of your computation (it allows us to visually verify your work). ♢

Exercise 11.6: Order the following functions:

$$n, \quad \lceil\lg n\rceil!, \quad \lceil\lg\lg n\rceil!, \quad n^{\lceil\lg\lg n\rceil!}, \quad 2^{\lg^* n}, \quad \lg^*(2^n), \quad \lg^*(\lg n), \quad \lg(\lg^* n).$$

♢

Exercise 11.7: (Purdom-Brown)
(a) Show that $\sum_{i=1}^n i! = n![1 + \mathcal{O}(1/n)]$. NOTE: The summation rule gives only a Θ-order so this is more precise.
(b) $\sum_{i=1}^n 2^i \ln i = 2^{n+1}[\ln n - (1/n) + \mathcal{O}(n^{-2})]$. HINT: use $\ln i = \ln n - (i/n) + \mathcal{O}(i^2/n^2)$ for $i = 1, \ldots, n$. ♢

Exercise 11.8: (Knuth) What is the asymptotic behavior of $n^{1/n}$? of $n(n^{1/n} - 1)$?
HINT: take logs. Alternatively, expand $\prod_{i=1}^n e^{1/(in)}$. ♢

Exercise 11.9: Estimate the growth behavior of the solution to this recurrence: $T(n) = T(n/2)^2 + 1$. ♢

End Exercises

§A. APPENDIX: Exponential and Logarithm Functions

Next to the polynomials, the two most important functions in algorithmics are the **exponential function** and its inverse, the **logarithm function**. Many of our asymptotic results depend on their basic properties. For the student who wants to understand these properties, the following will guide them through some exercises. We define the **natural exponential function** to be

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

for all real x. This definition is also good for complex x, but we do not need this here. The **base of the natural logarithm** is defined to be the number

$$e := \exp(1) = \sum_{i=0}^{\infty} \frac{1}{i!} = 2.71828...$$

The next exercise derives some asymptotic properties of the exponential function.

Exercise A.1: Show that

(a) $\exp(x)$ is continuous,
(b) $\frac{d\exp(x)}{dx} = \exp(x)$ and hence $\exp(x)$ has all derivatives,
(c) $\exp(x)$ is positive, strictly increasing,
(d) $\exp(x) \to 0$ as $x \to -\infty$, $\exp(x) \to \infty$ as $x \to \infty$,
(e) $\exp(x+y) = \exp(x)\exp(y)$,

$\diamondsuit$

We often need explicit bounds on exponential functions (not just asymptotic behavior). Derive the following bounds:

Exercise A.2:

(a) $\exp(x) \ge 1 + x$ for all $x \ge 0$ with equality iff $x = 0$.
(b) $\exp(x) > \frac{x^{n+1}}{(n+1)!}$ for $x > 0$. Hence $\exp(x)$ grow faster than any polynomial in x.
(c) For all real $n \ge 0$,

$$\left(1 + \frac{x}{n}\right)^n \le e^x \le \left(1 + \frac{x}{n}\right)^{n+(x/2)}.$$

It follows that an alternative definition of e^x is

$$e^x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n.$$

(d) $\exp(x)\left(1 - \frac{x^2}{n}\right) \le \left(1 + \frac{x}{n}\right)^n$ for all $x, n \in \mathbb{R}$, $n \ge 1$ and $|x| \le n$. See [12]. $\diamondsuit$

The **natural logarithm** function $\ln(x)$ is the inverse of $\exp(x)$: $\ln(x)$ is defined[11] to be the real number y such that $\exp(y) = x$. Note that this is a partial function because it is defined for all and only positive x.

[11] This real value y is called the principal value of the logarithm. That is because if we view $\exp(\cdot)$ as a complex function, then $\ln(x)$ is a multivalued function that takes all values of the form $y + 2n\pi$, $n \in \mathbb{Z}$.

Exercise A.3: Show that
(a) $\frac{d\ln(x)}{dx} = \frac{1}{x}$,
(b) $\ln(xy) = \ln(x) + \ln(y)$,
(c) $\ln(x)$ increases monotonically from $-\infty$ to $+\infty$ as x increases from 0 to $+\infty$. ♢

These two functions now allow us to define exponentiation to any base: for any positive b and any real α, we define

$$\exp_b(\alpha) := \exp(\alpha \ln(b)).$$

Usually, we write $\exp_b(\alpha)$ as b^α. Note that if $b = e$ then we obtain e^α, a familiar notation for $\exp(\alpha)$.

Again, the logarithm function $\log_b(x)$ to an arbitrary base $b > 0$ is defined to be the inverse of the function $f(y) = b^y$. However, $\log_b(x)$ is highly degenerate for $b = 1$ (being defined only when $x = 1$).

Exercise A.4: We show some familiar properties: the base b is omitted if it does not affect the stated property.
(a)

$$\begin{array}{l} \log 1 = 0, \quad \log_b b = 1, \quad \log_b x = (\log_c x)/(\log_c b), \\ y = x^{\log_x y}, \quad \log(x^y) = y \log x, \quad \log(ab) = (\log a) + (\log b). \end{array}$$

(b) $\log(1/x) = -\log x, \quad \log_b x = 1/(\log_x b), \quad a^{\log b} = b^{\log a}$.
(c) $\frac{dx}{dx}(x^\alpha) = \alpha x^{\alpha-1}$.
(d) For $b > 1$, the function $\log_b(x)$ increases monotonically from $-\infty$ to $+\infty$ as x increases from 0 to ∞. At the same time, for $0 < b < 1$, $\log_b(x)$ decreases monotonically from $+\infty$ to $-\infty$. ♢

Notations for Logarithms. Logarithms to base 2 is important in computer science and we will write "lg" for $\log_2$. Of course, ln is the natural logarithm. Some authors use *Log* for $\log_{10}$. Our default assumption is that the base of logarithms is some $b > 1$, so that $\log_b x$ is a monotonically increasing function. When the actual value of b is immaterial (except that $b > 1$), we simple write 'log' without specifying the base. We also write $\log^{(k)} n$ for the k-fold application of the logarithm function to n. Thus $\log^{(2)} n = \log\log n$, and by definition, $\log^{(0)} n = n$. This is to be distinguished from "$\log^k n$" which equals $(\log n)^k$. On the black board, we might sometimes write $\ell\log n$, $\ell\ell\log n$ for $\log\log n$, $\log\log\log n$, etc.

¶28. Bounds on logarithms. For approximations involving logarithms, it is useful to recall a fundamental series for logarithms:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots = -\sum_{i=1}^{\infty} \frac{(-x)^i}{i}$$

valid for $|x| < 1$. We easily see that $x - x^2/2 < \ln(1+x) < x$. To see that $\ln(1+x) < x$ we must show that $R = \sum_{i=2}^{\infty}(-x)^i/i > 0$. This follows because if we pair up the terms in R we obtain

$$R = (x^2/2 - x^3/3) + (x^4/4 - x^5/5) + \cdots,$$

which is clearly a sum of positive terms. A similar argument shows $\ln(1+x) > x - x^2/2$.

How do we evaluate $\ln(y)$ for a general $y > 2$? Assume that we have (a good approximation) to $\ln(2)$. Then we can write $y = 2^n(1+x)$ and thus evaluate $\ln(y)$ as $n\ln(2)+\ln(1+x)$. Alternatively, we can write $y = n(1+x)$ where $n \in \mathbb{N}$ and write $\ln(y) = \ln(n) + \ln(1+x)$. To evaluate $\ln(n)$ we use the fact $\ln(n) = H_n - \gamma - (2n)^{-1} - \mathcal{O}(n^{-2})$ (see §5).

¶29. Log-star function. We define the **log-star function**: $\log^* x$ is the maximum non-negative integer n such that $\lg^{(n)}(x)$ is defined. Thus $\log^*(x) = 0, 1, 2$ iff $x \le 0$, $0 < x \le 1$, $1 < x \le 2$ (respectively). So log-star is integer-valued. Although we have used base 2 in its definition, it could be defined generally for any $b > 1$.

References

[1] J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980.

[2] G. Dowek. Preliminary investigations on induction over real numbers, 2003. Manuscript, http://www.lix.polytechnique.fr/ dowek/publi.html.

[3] M. H. Escardó and T. Streicher. Induction and recursion on the partial real line with applications to Real PCF. *Theoretical Computer Science*, 210(1):121–157, 1999.

[4] W. Feller. *An introduction to Probability Theory and its Applications*. Wiley, New York, 2nd edition edition, 1957. (Volumes 1 and 2).

[5] D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*. Birkhäuser, 2nd edition, 1982.

[6] G. H. Hardy. *Orders of Infinity*. Cambridge Tracts in Mathematics and Mathematical Physics, No. 12. Reprinted by Hafner Pub. Co., New York. Cambridge University Press, 1910.

[7] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk SSSR*, 145:293–294, 1962.

[8] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Boston, 2nd edition edition, 1975.

[9] G. S. Lueker. Some techniques for solving recurrences. *Computing Surveys*, 12(4), 1980.

[10] B. P. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In *Proc. 6th Australian Software Engineering Conf. (ASWEC91)*, pages 257–270. Australian Comp. Soc., 1991.

[11] B. Mishra and A. Siegel. (Class Lecture Notes) Analysis of Algorithms, January 28, 1991.

[12] D. S. Mitrinović. *Analytic Inequalities*. Springer-Verlag, New York, 1970.

[13] J. Paul Walton Purdom and C. A. Brown. *The Analysis of Algorithms*. Holt, Rinehart and Winston, New York, 1985.

[14] R. M. Verma. A general method and a master theorem for divide-and-conquer recurrences with applications. *J. of Algorithms*, 16:67–79, 1994.

[15] X. Wang and Q. Fu. A frame for general divide-and-conquer recurrences. *Info. Processing Letters*, 59:45–51, 1996.