The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

My lover gave me nine linked rings.
With my two hands I could not untangle them, I could not untangle them.
My lover, please untangle my nine linked rings, nine linked rings.
I will marry you and you will be my man.

— "Nine Linked Rings", Chinese folk song (before 1800)

Our life is frittered away by detail.... Simplify, simplify.

— Henry David Thoreau, *Walden* (1854)

Nothing is particularly hard if you divide it into small jobs.

— Henry Ford

Do the hard jobs first. The easy jobs will take care of themselves.

— Dale Carnegie

CHAPTER 1

Recursion

Status: Beta. Explain domain transformations?

1.1 Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine. Crucially, the correctness of the resulting algorithm cannot depend in any way on *how* the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply *none of our business*; they're somebody else's problem. It's often best to literally think of the black box as functioning purely by magic.

For example, the peasant multiplication algorithm described in the previous chapter reduces the problem of multiplying two arbitrary positive integers to three simpler problems: addition, mediation (halving), and parity-checking. The algorithm relies on an abstract "positive integer" data type that supports those three operations, but the correctness of the multiplication algorithm does not depend on the precise data representation (tally marks, clay tokens, Babylonian hexagesimal, quipu, counting rods, Roman numerals, abacus beads, finger positions, Arabic numerals, binary, negabinary, Gray code, balanced ternary, Fibonacci coding, . . .), or on the precise implementations of those operations. Of course, the *running time* of the multiplication algorithm depends on the *running time* of the addition, median, and parity operations, but that's a separate issue from *correctness*. Most importantly, we can create a more efficient multiplication algorithm just by switching to a more efficient number representation (from Roman numerals to Arabic numerals, for example).

Similarly, the Huntington-Hill algorithm reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations Insert and ExtractMax. The abstract data type "priority queue" is a black box; the correctness of the apportionment algorithm does not depend on any specific priority queue data structure. Of course, the *running time* of the apportionment algorithm depends on the *running time* of the Insert and ExtractMax algorithms, but that's a separate issue from the *correctness* of the algorithm. The beauty of the reduction is that we can create a more efficient apportionment algorithm by simply swapping in a new priority queue data structure. Moreover, the designer of that data structure does not need to know or care that it will be used to apportion Congress.

When we design algorithms, we may not know exactly how the basic building blocks we use are implemented, or how our algorithms might be used as building blocks to solve even bigger problems. That ignorance is uncomfortable for many beginners, but it is both unavoidable and extremely useful. Even when you do know precisely how your components work, it is often *extremely* helpful to pretend that you don't.

1.2 Simplify and Delegate

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem can be solved directly, just solve it directly.
- Otherwise, reduce the instance to one or more ***simpler instances of the same problem***.

If this self-reference is confusing, it's helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. I like to call that someone else the ***Recursion Fairy***. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will magically take care of all the simpler subproblems for you, using Methods That Are None Of Your Business So *Butt Out*.[1] Mathematically sophisticated

[1]When I was a student, I used to attribute recursion to "elves" instead of the Recursion Fairy, referring to the Brothers Grimm story about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves ("Wichtelmänner") have finished everything overnight. Someone

readers might recognize the Recursion Fairy by its more formal name: the ***Induction Hypothesis***.

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to simpler and simpler instances. Eventually, the recursive reductions must lead to an elementary ***base case*** that can be solved by some other method; otherwise, the recursive algorithm will loop forever. The most common way to satisfy this condition is to reduce to one or more ***smaller*** instances of the same problem. For example, if the original input is a skreeble with n glurps, the input to each recursive call should be a skreeble with strictly less than n glurps. Of course this is impossible if the skreeble has no glurps at all—You can't have negative glurps; that would be silly!—so in that case we must grindlebloff the skreeble using some other method.

We've already seen one instance of this pattern in the peasant multiplication algorithm, which is based directly on the following identity.

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

The same identity can be expressed algorithmically as follows:

```
MULTIPLY(x, y):
  if x = 0
      return 0
  else
      x′ ← ⌊x/2⌋
      y′ ← y + y
      prod ← MULTIPLY(x′, y′)    ⟨⟨Recurse!⟩⟩
      if x is odd
          prod ← prod + y
      return prod
```

A lazy Egyptian scribe could execute this algorithm by computing x' and y', *asking a more junior scribe to multiply* x' *and* y', and then possibly adding y to the junior scribe's response. The junior scribe's problem is simpler because $x' < x$, and repeatedly reducing a positive integer eventually leads to 0. How the junior scribe actually computes $x' \cdot y'$ is none of the senior scribe's business (and it's none of your business, either).

1.3 Tower of Hanoi

The Tower of Hanoi puzzle was first published—as an actual physical puzzle!—by the French recreational mathematician Éduoard Lucas in 1883, under the pseudonym

more entheogenically experienced than I might recognize them as Terence McKenna's "self-transforming machine elves".

"N. Claus (de Siam)" (an anagram of "Lucas d'Amiens").[2] The following year, Henri de Parville described the puzzle with the following remarkable story:[3]

> *In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.*

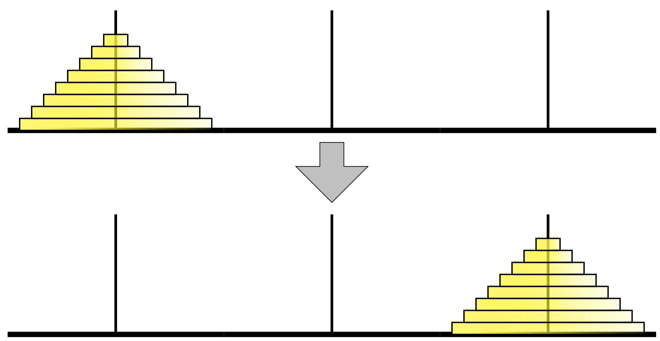


Figure 1.1. The Tower of Hanoi puzzle

Of course, as good computer scientists, our first instinct on reading this story is to substitute the variable n for the hardwired constant 64. And following standard practice (since most physical instances of the puzzle are made of wood instead of diamonds and gold), we will refer to the three possible locations for the disks as "pegs" instead of "needles". How can we move a tower of n disks from one peg to another, using a third peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?

As Claus de Siam pointed out in the pamphlet included with his puzzle, the secret to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle all at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are covering it; we have to move those $n-1$ disks to the third peg before we can move the largest disk. And then after we move the largest disk, we have to move those $n-1$ disks back on top of it.

So now all we have to figure out is how to—

[2]Lucas later claimed to have invented the puzzle in 1876.

[3]This English translation is from W. W. Rouse Ball and H. S. M. Coxeter's book *Mathematical Recreations and Essays*.

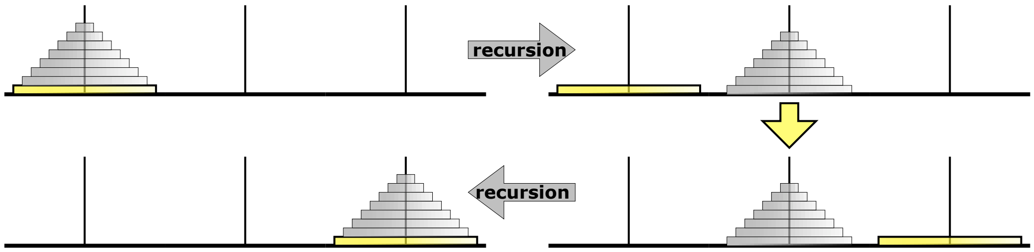


Figure 1.2. The Tower of Hanoi algorithm; ignore everything but the bottom disk.

STOP!! That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy—or to carry the original metaphor further, to the junior monks at the temple.

Our reduction does make one subtle but extremely important assumption: *There is a largest disk*. In other words, our recursive algorithm works for any $n \geq 1$, but it breaks down when $n = 0$. We must handle that base using a different method. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one peg to another in no time at all, by doing nothing.

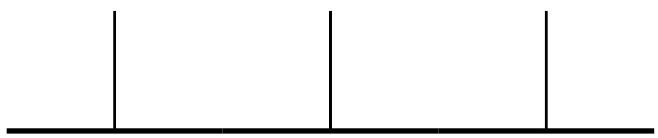


Figure 1.3. The vacuous base case for the Tower of Hanoi algorithm. There is no spoon.

While it's tempting to think about how all those smaller disks move around—or more generally, what happens when the recursion is unrolled—it's completely unnecessary. For even slightly more complicated algorithms, unrolling the recursion is far more confusing than illuminating. Our ***only*** task is to reduce the problem instance we're given to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves the top $n-1$ disks (more formally, the Inductive Hypothesis implies that our recursive algorithm correctly moves the top $n-1$ disks) so our algorithm is correct.

The recursive Hanoi algorithm is expressed in pseudocode in Figure 1.4. The algorithm moves a stack of n disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*) as a placeholder. Notice that the algorithm correctly does nothing at all when $n = 0$.

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1) + 1$ for any $n \geq 1$. By writing out the first several values of $T(n)$, we can easily guess that $\boldsymbol{T(n) = 2^n - 1}$; a straightforward

<u>HANOI(n, src, dst, tmp):</u>
  if $n > 0$
    HANOI($n-1, src, tmp, dst$) ⟨⟨*Recurse!*⟩⟩
    move disk n from *src* to *dst*
    HANOI($n-1, tmp, dst, src$) ⟨⟨*Recurse!*⟩⟩

Figure 1.4. A recursive algorithm to solve the Tower of Hanoi

induction proof implies that this guess is correct.In particular, moving a tower of 64 disks requires $2^{64} - 1 =$ 18,446,744,073,709,551,615 individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years ("plus de *cinq millards de siècles*") before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

1.4 Mergesort

Mergesort is one of the earliest algorithms proposed for sorting. According to Donald Knuth, it was proposed by John von Neumann as early as 1945.

1. Divide the input array into two subarrays of roughly equal size.
2. Recursively mergesort each of the subarrays.
3. Merge the newly-sorted subarrays into a single sorted array.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N |	G	E	X	A	M	P	L
Recurse:	I	N	O	S	R	T |	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.5. A mergesort example.

The first step is completely trivial—we only need to compute the median array index—and we can delegate the second step to the Recursion Fairy. All the real work is done in the final step; the two sorted subarrays can be merged using a simple linear-time algorithm. A complete description of the algorithm is given in Figure 1.6; to keep the recursive structure clear, I've extracted the merge step into an independent subroutine.

Correctness

To prove that this algorithm is correct, we apply our old friend induction twice, first to the MERGE subroutine then to the top-level MERGESORT algorithm.

Lemma 1.1. *MERGE correctly merges the subarrays $A[1..m]$ and $A[m+1..n]$, assuming those subarrays are sorted in the input.*

Proof: Let $A[1..n]$ be any array and m any integer such that the subarrays $A[1..m]$ and $A[m+1..n]$ are sorted. We prove that for all k from 0 to n, the last $n-k-1$ iterations

```
MergeSort(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MergeSort(A[1..m])
      MergeSort(A[m+1..n])
      Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

Figure 1.6. Mergesort

of the main loop correctly merge $A[i\,..\,m]$ and $A[j\,..\,n]$ into $B[k\,..\,n]$. The proof proceeds by induction on $n-k+1$, the number of elements remaining to be merged.

If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.) Otherwise, there are four cases to consider for the kth iteration of the main loop.

- If $j > n$, subarray $A[j\,..\,n]$ is empty, so $\min\big(A[i\,..\,m] \cup A[j\,..\,n]\big) = A[i]$.
- Otherwise, if $i > m$, subarray $A[i\,..\,m]$ is empty, so $\min\big(A[i\,..\,m] \cup A[j\,..\,n]\big) = A[j]$.
- Otherwise, if $A[i] < A[j]$, then $\min\big(A[i\,..\,m] \cup A[j\,..\,n]\big) = A[i]$.
- Otherwise, we must have $A[i] \geq A[j]$, and thus $\min\big(A[i\,..\,m] \cup A[j\,..\,n]\big) = A[j]$.

In all four cases, $B[k]$ is correctly assigned the smallest element of $A[i\,..\,m] \cup A[j\,..\,n]$. In the two cases with the assignment $B[k] \leftarrow A[i]$, the Recursion Fairy correctly merges—sorry, I mean the Induction Hypothesis implies that the last $n-k$ iterations of the main loop correctly merge $A[i+1\,..\,m]$ and $A[j\,..\,n]$ into $B[k+1\,..\,n]$. Similarly, in the other two cases, the Recursion Fairy correctly merges the rest of the subarrays. □

Theorem 1.2. *MergeSort correctly sorts any input array* $A[1\,..\,n]$.

Proof: We prove the theorem by induction on n. If $n \leq 1$, the algorithm correctly does nothing. Otherwise, the Recursion Fairy correctly sorts—sorry, I mean the induction hypothesis implies that our algorithm correctly sorts—the two smaller subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$, after which they are correctly Merged into a single sorted array (by Lemma 1.1). □

Analysis

What's the running time? Because the MergeSort algorithm is recursive, its running time is easily expressed by a recurrence. Merge clearly takes linear time, because it's a simple for-loop with constant work per iteration. We immediately obtain the following recurrence for MERGESORT:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

As in most divide-and-conquer recurrences, we can safely strip out the floors and ceilings using a domain transformation ⟨⟨**explain?**⟩⟩, giving us the simpler recurrence

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

The "all levels equal" case of the recursion tree method (described later in this chapter) immediately implies the closed-form solution $\boldsymbol{T(n) = O(n \log n)}$. Even if you are not (yet) familiar with recursion trees, you can verify the solution $T(n) = O(n \log n)$ by induction.

1.5 Quicksort

Quicksort is another recursive sorting algorithm, discovered by Tony Hoare in 1962. In this algorithm, the hard work is splitting the array into subsets so that merging the final result is trivial.

1. Choose a *pivot* element from the array.
2. Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.
3. Recursively quicksort the first and last subarray.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	O	E	I	N	L	M	P	T	X	S	R
Recursse:	A	E	G	I	L	M	N	O	P	R	S	T	X

Figure 1.7. A quicksort example.

A more detailed description of the algorithm is given in Figure 1.8. In the separate PARTITION subroutine, the input parameter p is index of the pivot element in the unsorted array; the subroutine partitions the array and returns the new index of the pivot.

Correctness

Just like mergesort, proving QUICKSORT is correct requires two separate induction proofs: one to prove that PARTITION correctly partitions the array, and the other to prove that QUICKSORT correctly sorts *assuming* PARTITION is correct. I'll leave the tedious details as an exercise for the reader.

```
QuickSort(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1..r−1])
      QuickSort(A[r+1..n])
```

```
Partition(A[1..n], p):
  swap A[p] ↔ A[n]
  i ← 0
  j ← n
  while (i < j)
      repeat i ← i+1 until (i ≥ j or A[i] ≥ A[n])
      repeat j ← j−1 until (i ≥ j or A[j] ≤ A[n])
      if (i < j)
          swap A[i] ↔ A[j]
  swap A[i] ↔ A[n]
  return i
```

Figure 1.8. Quicksort

Analysis

The analysis is also similar to mergesort. Partition runs in $O(n)$ time: $j-i=n$ at the beginning, $j-i=0$ at the end, and we do a constant amount of work each time we increment i or decrement j. For QuickSort, we get a recurrence that depends on r, the *rank* of the chosen pivot element:

$$T(n)=T(r-1)+T(n-r)+O(n)$$

If we could somehow choose the pivot to be the *median* element of the array A, we would have $r=\lceil n/2\rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n)=2T\big(\lceil n/2\rceil-1\big)+T\big(\lfloor n/2\rfloor\big)+O(n)\le 2T(n/2)+O(n),$$

and we'd have $T(n)=O(n\log n)$ by the recursion tree method.

In fact, as we will see shortly, we *can* locate the median element in an unsorted array in linear time. However, the algorithm is fairly complicated, and the hidden constant in the $O(\cdot)$ notation is large enough to make the resulting sorting algorithm impractical. In practice, most programmers settle for something simple, like choosing the first or last element of the array. In this case, r take any value between 1 and n, so we have

$$T(n)=\max_{1\le r\le n}\big(T(r-1)+T(n-r)+O(n)\big).$$

In the worst case, the two subproblems are completely unbalanced—either $r=1$ or $r=n$—and the recurrence becomes $T(n)\le T(n-1)+O(n)$. The solution is $\boldsymbol{T(n)=O(n^2)}$.

Another common heuristic is called "median of three"—choose three elements (usually at the beginning, middle, and end of the array), and take the median of those three elements the pivot. Although this heuristic is somewhat more efficient in practice than just choosing one element, especially when the array is already (nearly) sorted, we can still have $r=2$ or $r=n-1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n)\le T(1)+T(n-2)+O(n)$, whose solution is still $T(n)=O(n^2)$.

Intuitively, the pivot element should "usually" fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This observation suggests that the *average-case* running time should be $O(n \log n)$. Although this intuition is actually correct (at least under the right formal assumptions), we are still far from a *proof* that quicksort is usually efficient. We will formalize this intuition about average-case behavior in a later chapter.

1.6 The Pattern

Both mergesort and and quicksort follow a general three-step pattern shared by all divide and conquer algorithms:

1. **Divide** the given instance of the problem into several *independent smaller* instances.
2. **Delegate** each smaller instance to the Recursion Fairy.
3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

If the size of any subproblem falls below some constant threshold, the recursion bottoms out. Hopefully, at that point, the problem is trivial, but if not, we switch to a different algorithm instead.

Proving a divide-and-conquer algorithm correct almost always requires induction. Analyzing the running time requires setting up and solving a recurrence, which usually (but unfortunately not always!) can be solved using recursion trees, perhaps after a simple domain transformation.

1.7 Recursion Trees

So what are these recursion trees I keep talking about? Imagine a divide-and-conquer algorithm that spends $O(f(n))$ time on non-recursive work, and then makes r recursive calls, each on a problem of size n/c. The running time of such an algorithm would be governed by the recurrence

$$T(n) = r\,T(n/c) + f(n)$$

Recursion trees are a simple, general, pictorial method for solving recurrences in this form, and in even more general forms. The root of the recursion tree is a box containing the value $f(n)$; the root has a children, each of which is the root of a (recursively defined) recursion tree for the function $T(n/c)$. Equivalently, a recursion tree is a complete a-ary tree where each node at depth i contains the value $f(n/c^i)$. In practice, I recommend only drawing out the first two or three levels of the tree.

The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the precise base case doesn't matter; we can safely assume that $T(1) = O(1)$, or even that $T(n) = O(1)$ for all $n \le 10^{100}$. I'll also assume for simplicity that n is an integral power of b (although this really doesn't matter).

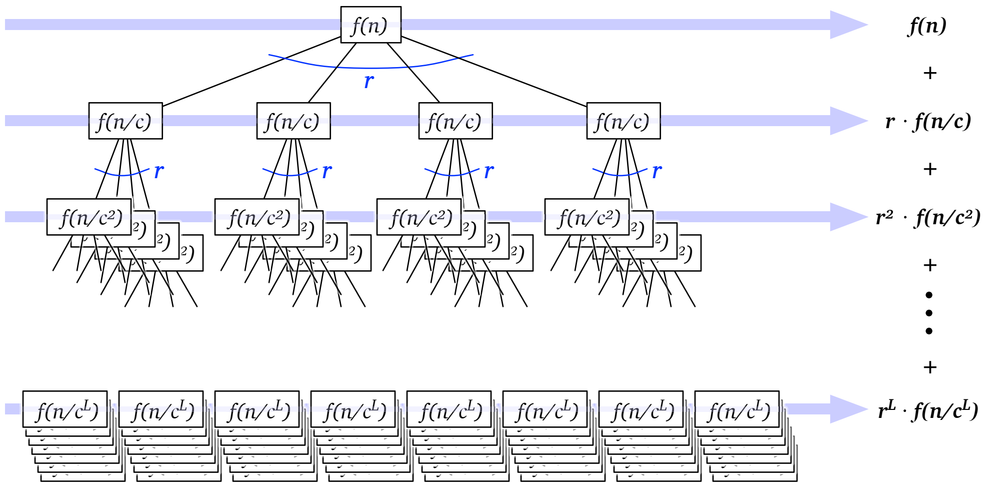


Figure 1.9. A recursion tree for the recurrence $T(n) = r\,T(n/c) + f(n)$

Now $T(n)$ is just the sum of all values stored in the recursion tree; we can evaluate this sum by considering the tree level-by-level. For each i, the ith level of the tree contains a^i nodes, each with value $f(n/b^i)$. Thus,

$$T(n) = \sum_{i=0}^{L} r^i f(n/c^i) \tag{Σ}$$

where L is the depth of the recursion tree. We easily see that $\boldsymbol{L = \log_c n}$, because $n/c^L = 1$. The base case $f(1) = \Theta(1)$ implies that the last non-zero term in the summation is $O(a^L) = O(a^{\log_c r}) = O(n^{\log_c r})$.

There are three common cases where the level-by-level sum (Σ) is easy to evaluate:

- **Decreasing:** If the sum is a *decreasing geometric series*—every term is a constant factor smaller than the previous term—then the sum is dominated by the value at the root of the recursion tree: $T(n) = O(f(n))$.
- **Equal:** If all terms in the sum are equal, we immediately have $T(n) = O(f(n) \cdot L) = O(f(n) \log n)$.
- **Increasing:** If the sum is an *increasing geometric series*—every term is a constant factor larger than the previous term—then the sum is dominated by the number of leaves in the recursion tree: $T(n) = O(n^{\log_c r})$.

In the first and third cases, only the largest term in the geometric series matters; all other terms are swallowed up by the $O(\cdot)$ notation. In the decreasing case, we don't even have to compute L; the asymptotic bound would still hold if the recursion tree were infinite!

For example, if we draw out the first few levels of the recursion tree for our mergesort recurrence $T(n) = 2T(n/2) + O(n)$, we discover that all levels are equal, which implies $T(n) = O(n \log n)$.

The recursion tree technique can also be used for algorithms where the recursive subproblems are not the same size. For example, the worst-case recurrence for quicksort $T(n) = T(n-1) + T(1) + O(n)$ gives us a completely unbalanced recursion tree, where one child of each internal node is a leaf. The level-by-level sum doesn't fall into any of our three default categories, but we can still derive the solution $T(n) = O(n^2)$ by observing that every level value is at most n and there are at most n levels. (Moreover, since $n/2$ levels each have value at least $n/2$, this analysis can be improved by at most a constant factor, which for our purposes means not at all.)

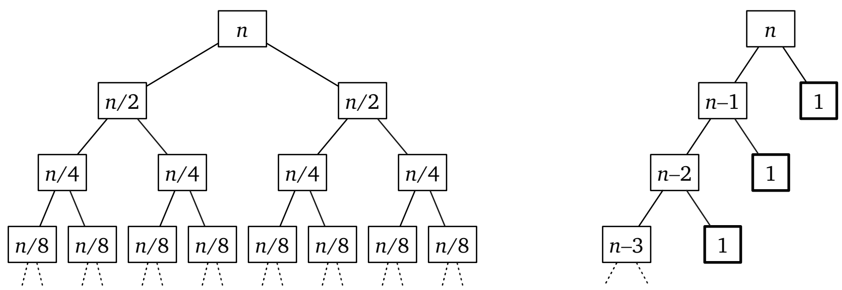


Figure 1.10. The recursion trees for mergesort and quicksort

♥1.8 Selection

So how *do* we find the median element of an array in linear time? The following algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth largest element in an n-element array, given the array and the integer k as input, using a variant of an algorithm called either "quickselect" or "one-armed quicksort". The basic quickselect algorithm chooses a pivot element, partitions the array using the PARTITION subroutine from QUICKSORT, and then recursively searches only *one* of the two subarrays.

```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)

      if k < r
          return QUICKSELECT(A[1..r − 1], k)
      else if k > r
          return QUICKSELECT(A[r + 1..n], k − r)
      else
          return A[r]
```

The worst-case running time of QuickSelect obeys a recurrence similar to the QuickSort recurrence. We don't know the value of r or which subarray we'll recursively search, so we'll just assume the worst.

$$T(n) \;\le\; \max_{1\le r\le n} \Big(\max\{T(r-1), T(n-r)\} + O(n)\Big)$$

We can simplify the recurrence by using ℓ to denote the length of the recursive subproblem:

$$T(n) \;\le\; \max_{0\le \ell\le n-1} T(\ell) + O(n)$$

If the chosen pivot element is always either the smallest or largest element in the array, the recurrence simplifies to $T(n) = T(n-1) + O(n)$, which implies $T(n) = O(n^2)$. (The recursion tree for this recurrence is just a simple path.)

We could avoid this quadratic worst-case behavior if we could somehow magically choose a ***good*** pivot, meaning $\ell \le \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

$$T(n) \le T(\alpha n) + O(n).$$

This recurrence expands into a descending geometric series, which is dominated by its largest term, so $T(n) = O(n)$. (Again, the recursion tree is just a simple path. The constant in the $O(n)$ running time depends on the constant α.)

The Blum-Floyd-Pratt-Rivest-Tarjan algorithm chooses a good pivot for one-armed quicksort by ***recursively*** computing the median of a carefully-selected subset of the input array. Specifically, we divide the input array into $\lceil n/5\rceil$ *blocks*, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞s.) We compute the median of each block by brute force, collect those medians into a new array $M[1\,..\,\lceil n/5\rceil]$, and then recursively compute the median of this new array. Finally we use the median of medians (called "*mom*" in the following pseudocode) as the pivot in one-armed quicksort.

```
MomSelect(A[1..n], k):
  if n ≤ 25   ⟨⟨or whatever⟩⟩
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MedianOfFive(A[5i − 4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MomSelect(M[1..m], ⌊m/2⌋)          ⟨⟨Recursion!⟩⟩

      r ← Partition(A[1..n], mom)

      if k < r
          return MomSelect(A[1..r − 1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MomSelect(A[r + 1..n], k − r) ⟨⟨Recursion!⟩⟩
      else
          return mom
```

The first key insight is that the median of medians is in fact a good pivot. The median of medians is larger than $\lfloor \lceil n/5 \rceil / 2 \rfloor - 1 \approx n/10$ block medians, and each block median is larger than two other elements in its block. Thus, mom is larger than at least $3n/10$ elements in the input array; symmetrically, mom is smaller than at least $3n/10$ input elements. Thus, in the worst case, the last recursive call searches an array of size at most $7n/10$.

We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that *the algorithm does not actually do this!*) In this arrangement, the median-of-medians is the element closest to the center of the grid.

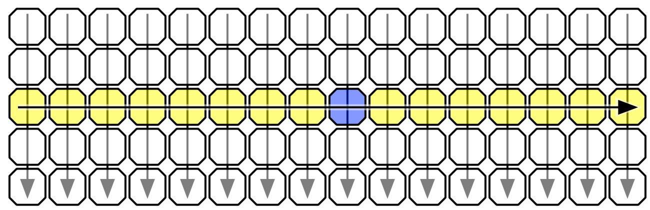


Figure 1.11. Visualizing the median of medians

The left half of the first three rows of the grid contains $3n/10$ elements, each of which is smaller than the median-of-medians. If the element we're looking for is larger than the median-of-medians, our algorithm will throw away *everything* smaller than the median-of-median, including those $3n/10$ elements, before recursing. Thus, the input to the recursive subproblem contains at most $7n/10$ elements. A symmetric argument applies when our target element is smaller than the median-of-medians.

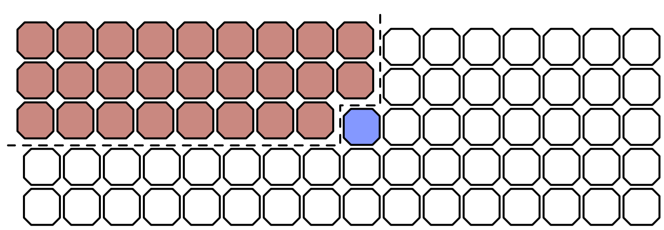


Figure 1.12. Discarding approximately 3/10 of the array

Okay, so mom is a good pivot, but now the algorithm is making *two* recursive calls instead of just one; how do we know the resulting running time is still linear? The second key insight is that the *total* size of the two recursive subproblems is a constant factor smaller than the size of the original input array. The worst-case running time of the algorithm obeys the recurrence

$$T(n) \le O(n) + T(n/5) + T(7n/10).$$

If we draw out the recursion tree for this recursion, we observe that the total work at each level of the recursion tree is at most $9/10$ the total work at the previous level. Thus, the level-by-level sum is a descending geometric series, giving us the solution $T(n) = O(n)$.

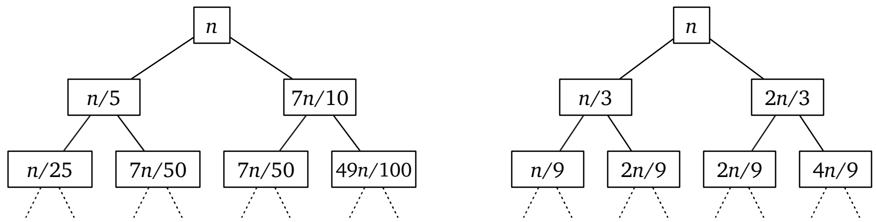


Figure 1.13. The recursion trees for MomSelect and for a similar selection algorithm with blocks of size 3

Where did the magic constant 5 come from? That's the smallest odd block size that gives us a descending geometric series in the running time! (Even block sizes introduce additional complications.) If we had used blocks of 3 elements instead, the running-time recurrence would have been

$$T(n) \le O(n) + T(n/3) + T(2n/3).$$

In this case, every level of the recursion tree has the same value n. The leaves of the recursion tree are not all at the same level, but for purposes of deriving an upper bound, it suffices to observe that the *deepest* leaf is at level $\log_{3/2} n$, so the total work in the tree is *at most* $O(n \log_{3/2} n) = O(n \log n)$. So this algorithm is no faster than sorting!

Finer analysis reveals that the constant hidden by the $O()$ notation is quite large, even if we count only comparisons. Selecting the median of 5 elements requires at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. Naïvely partitioning the array after the recursive call would require $n-1$ comparisons, but we already know $3n/10$ elements larger than the pivot and $3n/10$ elements smaller than the pivot, so partitioning actually requires only $2n/5$ additional comparisons. Thus, a more precise recurrence for the worst-case number of comparisons is

$$T(n) \le 8n/5 + T(n/5) + T(7n/10).$$

The recursion tree method implies the upper bound

$$T(n) \le \frac{8n}{5} \sum_{i \ge 0} \left(\frac{9}{10}\right)^i = \frac{8n}{5} \cdot 10 = 16n.$$

In practice, this algorithm isn't as horrible as this worst-case analysis predicts—getting a worst-case partition at every level of recursion is incredibly unlikely—but it is slower than sorting for even moderately large arrays.

1.9 Multiplication

In Chapter 0, we saw two different algorithms for multiplying two n-digit numbers in $O(n^2)$ time: the grade-school lattice algorithm and the Egyptian peasant algorithm.

Perhaps we can get a more efficient algorithm by splitting the numbers in half, and exploiting the following identity:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

Here is a divide-and-conquer algorithm that computes the product of two n-digit numbers x and y, based on this formula. Each of the four sub-products e, f, g, h is computed recursively. The last line does not involve any multiplications, however; to multiply by a power of ten, we just shift the digits and fill in the right number of zeros.

```
MULTIPLY(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← MULTIPLY(a, c, m)
      f ← MULTIPLY(b, d, m)
      g ← MULTIPLY(b, c, m)
      h ← MULTIPLY(a, d, m)
      return 10^{2m} e + 10^m (g + h) + f
```

Correctness of this algorithm follows easily by induction. The running time for this algorithm is given by the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + O(n).$$

(We can safely ignore the ceiling in the recursive argument.) The recursion tree method transforms this recurrence into an *increasing* geometric series, which implies $T(n) = O(n^{\log_2 4}) = O(n^2)$. Hmm... I guess this didn't help after all.

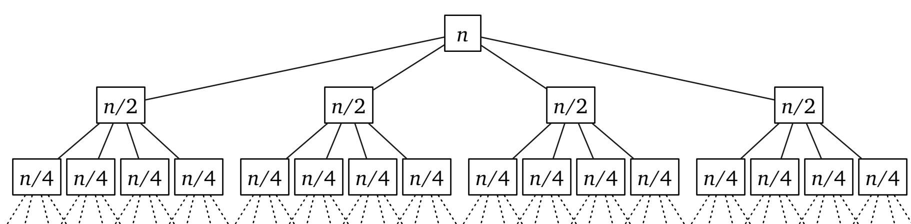


Figure 1.14. The recursion tree for naive divide-and-conquer multiplication

In the mid-1950s, Andrei Kolmogorov, one of the giants of 20th century mathematics, publicly conjectured that there is *no* algorithm to multiply two n-digit numbers in $o(n^2)$

time. Kolmogorov organized a seminar at Moscow University in 1960, where he restated his "n^2 conjecture" and posed several related problems that he planned to discuss at future meetings. Almost exactly one week later, 23-year-old student Anatoliĭ Karatsuba presented Kolmogorov with a remarkable counterexample. According to Karastuba himself,

> After the seminar I told Kolmogorov about the new algorithm and about the disproof of the n^2 conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at that point the seminar was terminated.

Karastuba observed that the middle coefficient $bc + ad$ can be computed from the other two coefficients ac and bd using only *one* more recursive multiplication, via the following algebraic identity:

$$ac + bd - (a - b)(c - d) = bc + ad$$

This trick lets us replace the four recursive calls in the previous algorithm with just three recursive calls, as shown below:

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m        ⟨⟨x = 10^m a + b⟩⟩
      c ← ⌊y/10^m⌋;  d ← y mod 10^m        ⟨⟨y = 10^m c + d⟩⟩
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^{2m} e + 10^m (e + f − g) + f
```

The running time of Karatsuba's FastMultiply algorithm is given by the recurrence

$$T(n) \le 3T(\lceil n/2 \rceil) + O(n)$$

Again, we can safely ignore the ceiling in the recursive argument, and the recursion tree method transforms the recurrence into an increasing geometric series, but the new solution is only $T(n) = O(n^{\log_2 3}) = \boldsymbol{O(n^{1.58496})}$, a significant improvement over our earlier quadratic-time algorithm.[4] Karatsuba's algorithm arguably launched the design and analysis of algorithms as a formal field of study.

Of course, in practice, all this is done in binary (or perhaps base 256 or 65536) instead of decimal.

[4] My presentation simplifies the actual history slightly. In fact, Karatsuba proposed an algorithm based on the formula $(a + c)(b + d) - ac - bd = bc + ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is slightly messier: $a - b$ and $c - d$ are still m-digit numbers, but $a + b$ and $c + d$ might each have $m + 1$ digits. The simplification presented here is due to Donald Knuth.

We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to obtain even faster multiplication algorithms. Andrei Toom and Stephen Cook discovered an infinite family of algorithms that split any integer into k parts, each with n/k digits, and then compute the product using only $2k-1$ recursive multiplications. For any fixed k, the resulting algorithm runs in $O(n^{1+1/(\lg k)})$ time, where the hidden constant in the $O(\cdot)$ notation depends on k.

Ultimately, this divide-and-conquer strategy led Gauss (yes, really) to the discovery of the ***Fast Fourier transform***, which is described in detail in a later chapter. The fastest multiplication algorithm known, published by Martin Fürer in 2007 and based on FFTs, runs in $O(n \log n 2^{O(\log^* n)})$ time. Here, $\log^* n$ denotes the slowly growing *iterated logarithm* of n, which is the number of times one must take the logarithm of n before the value is less than 1:

$$\lg^* n = \begin{cases} 1 & \text{if } n \leq 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

For all practical purposes, $\log^* n \leq 6$. It is widely conjectured that the best possible algorithm for multiply two n-digit numbers runs in $\Theta(n \log n)$ time.

1.10 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that does $n-1$ multiplications by a:

```
SlowPower(a, n):
  x ← a
  for i ← 2 to n
      x ← x · a
  return x
```

This iterative algorithm requires n multiplications.

Notice that the input a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). Since we don't know what kind of things we're multiplying, we can't know how long a multiplication takes, so we're forced analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, using the following simple recursive formula:

$$a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}.$$

What makes this approach more efficient is that once we compute the first factor $a^{\lfloor n/2 \rfloor}$, we can compute the second factor $a^{\lceil n/2 \rceil}$ using at most one more multiplication.

```
FastPower(a, n):
  if n = 1
      return a
  else
      x ← FastPower(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

The total number of multiplications satisfies the recurrence $T(n) \leq T(n/2) + 2$. The recursion-tree method immediately give us the solution $T(n) = O(\log n)$.

Incidentally, this algorithm is asymptotically optimal—any algorithm for computing a^n must perform at least $\Omega(\log n)$ multiplications. In fact, when n is a power of two, this algorithm is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, our divide-and-conquer algorithm computes a^{15} in six multiplications ($a^{15} = a^7 \cdot a^7 \cdot a$; $a^7 = a^3 \cdot a^3 \cdot a$; $a^3 = a \cdot a \cdot a$), but only five multiplications are necessary ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). It is an open question whether the absolute minimum number of multiplications for a given exponent n can be computed efficiently.

Exercises

Tower of Hanoi

1. Prove that the original recursive Tower of Hanoi algorithm performs *exactly* the same sequence of moves—the same disks, to and from the same pegs, in the same order—as each of the following non-recursive algorithms. The pegs are labeled 0, 1, and 2, and our problem is to move a stack of n disks from peg 0 to peg 2 (as shown on page 4).

   (a) If n is even, swap pegs 1 and 2. At the ith step, make the only legal move that avoids peg $i \bmod 3$. If there is no legal move, then all disks are on peg $i \bmod 3$, and the puzzle is solved. Homework

   (b) For the first move, move disk 1 to peg 1 if n is even and to peg 2 if n is odd. Then repeatedly make the only legal move that does not undo the previous move. If no such move exists, the puzzle is solved. Homework

   (c) Pretend that disks $n+1$, $n+2$, and $n+3$ are at the bottom of pegs 0, 1, and 2, respectively. Repeatedly make the only legal move that satisfies the following constraints, until no such move is possible. Homework

   - Do not place an odd disk directly on top of another odd disk.
   - Do not place an even disk directly on top of another even disk.
   - Do not undo the previous move.

   (d) Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. Homework

```
HANOI(n):
  i ← 1
  while ρ(i) ≤ n
      if n − i is even
          move disk ρ(i) forward      ⟨⟨0 → 1 → 2 → 0⟩⟩
      else
          move disk ρ(i) backward     ⟨⟨0 → 2 → 1 → 0⟩⟩
      i ← i + 1
```

For example, $\rho(42) = 2$, because $42/2^1$ is an integer but $42/2^2$ is not. (Equivalently, $\rho(n)$ is one more than the position of the least significant 1 in the binary representation of n.) Because its behavior resembles the marks on a ruler, $\rho(n)$ is sometimes called the *ruler function*.

Homework

2. The Tower of Hanoi is a relatively recent descendant of a much older mechanical puzzle known as the Chinese linked rings, Baguenaudier (a French word meaning "to wander about aimlessly"), Meleda, Patience, Tiring Irons, Prisoner's Lock, Spin-Out, and many other names. This puzzle was already well known in both China and Europe by the 16th century. The Italian mathematician Luca Pacioli described the 7-ring puzzle and its solution in his unpublished treatise *De Viribus Quantitatis*, written between 1498 and 1506;[5] only a few years later, the Ming-dynasty poet Yang Shen described the 9-ring puzzle as "a toy for women and children". The puzzle is apocryphally attributed to a 2nd-century Chinese general, who gave the puzzle to his wife to occupy her time while he was away at war.

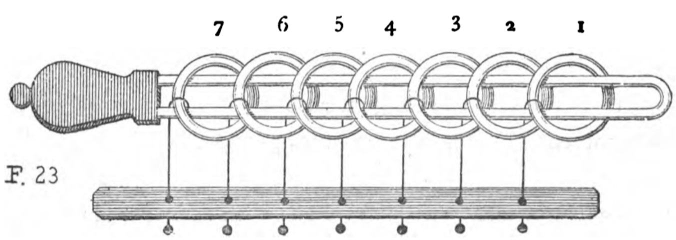


Figure 1.15. The 7-ring Baguenaudier, from *Récréations Mathématiques* by Édouard Lucas (1891)

The Baguenaudier puzzle has many physical forms, but it typically consists of a long metal loop and several rings, which are connected to a solid base by movable rods. The loop is initially threaded through the rings as shown in the figure above; the goal of the puzzle is to remove the loop.

More abstractly, we can model the puzzle as a sequence of bits, one for each ring, where the ith bit is `1` if the loop passes through the ith ring and `0` otherwise. (Here

[5] *De Viribus Quantitatis* [*On the Powers of Numbers*] is an important early work on recreational mathematics and perhaps the oldest surviving treatise on magic. Pacioli is better known for *Summa de Aritmetica*, a near-complete encyclopedia of late 15th-century mathematics, which included the first description of double-entry bookkeeping.

we index the rings from right to left, as shown in the figure.) The puzzle allows two legal moves:

- You can always flip the 1st (= rightmost) bit.
- If the bit string ends with exactly i `0`s, you can flip the $(i+2)$th bit.

The goal of the puzzle is to transform a string of n `1`s into a string of n `0`s. For example, the following sequence of 21 moves solves the 5-ring puzzle:

$$\texttt{11111} \xrightarrow{1} \texttt{11110} \xrightarrow{3} \texttt{11010} \xrightarrow{1} \texttt{11011} \xrightarrow{2} \texttt{11001} \xrightarrow{1} \texttt{11000} \xrightarrow{5} \texttt{01000}$$
$$\xrightarrow{1} \texttt{01001} \xrightarrow{2} \texttt{01011} \xrightarrow{1} \texttt{01010} \xrightarrow{3} \texttt{01110} \xrightarrow{1} \texttt{01111} \xrightarrow{2} \texttt{01101} \xrightarrow{1} \texttt{01100} \xrightarrow{4} \texttt{00100}$$
$$\xrightarrow{1} \texttt{00101} \xrightarrow{2} \texttt{00111} \xrightarrow{1} \texttt{00110} \xrightarrow{3} \texttt{00010} \xrightarrow{1} \texttt{00011} \xrightarrow{2} \texttt{00001} \xrightarrow{1} \texttt{00000}$$

♦(a) Call a sequence of moves *reduced* if no move is the inverse of the previous move. Prove that for any non-negative integer n, there is *exactly one* reduced sequence of moves that solves the n-ring Baguenaudier puzzle. *[Hint: This problem is much easier if you're already familiar with graphs.]*

(b) Describe an algorithm to solve the Baguenaudier puzzle. Your input is the number of rings n; your algorithm should print a reduced sequence of moves that solves the puzzle. For example, given the integer 5 as input, your algorithm should print the sequence $1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1$.

(c) *Exactly* how many moves does your algorithm perform, as a function of n? Prove your answer is correct.

3. A less familiar chapter in the Tower of Hanoi's history is its brief relocation of the temple from Benares to Pisa in the early 13th century. The relocation was organized by the wealthy merchant-mathematician Leonardo Fibonacci, at the request of the Holy Roman Emperor Frederick II, who had heard reports of the temple from soldiers returning from the Crusades. The Towers of Pisa and their attendant monks became famous, helping to establish Pisa as a dominant trading center on the Italian peninsula. Homework

Unfortunately, almost as soon as the temple was moved, one of the diamond needles began to lean to one side. To avoid the possibility of the leaning tower falling over from too much use, Fibonacci convinced the priests to adopt a more relaxed rule: ***Any number of disks on the leaning needle can be moved together to another needle in a single move.*** It was still forbidden to place a larger disk on top of a smaller disk, and disks had to be moved one at a time *onto* the leaning needle or between the two vertical needles.

Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa then they could than could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly

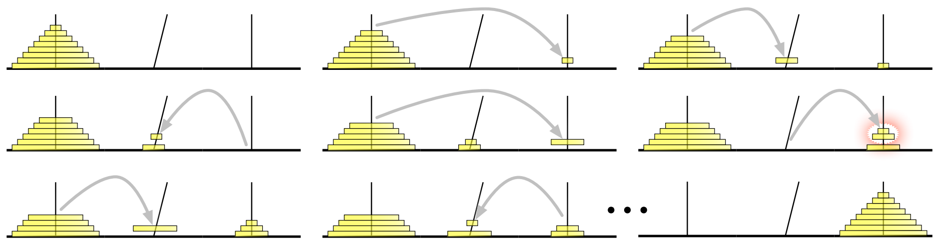


Figure 1.16. The Towers of Pisa. In the fifth move, two disks are taken off the leaning needle.

crowned Pope Gregory IX excommunicated Frederick II, making the local priests less sympathetic to hosting foreign heretics with strange mathematical habits. Soon afterward, a bell tower was erected on the spot where the temple once stood; it too began to lean almost immediately.

Describe an algorithm to transfer a stack of n disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform?

4. Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the pegs are numbered 0, 1, and 2, as in problem 1, and your task is to move a stack of n disks from peg 1 to peg 2.

Exam

(a) Suppose you are forbidden to move any disk directly between peg 1 and peg 2; *every* move must involve peg 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

Homework

♥(b) Suppose you are only allowed to move disks from peg 0 to peg 2, from peg 2 to peg 1, or from peg 1 to peg 0. Equivalently, suppose the pegs are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make? *[Hint: See the chapter on solving recurrences in the appendix.]*

Fun Homework

♣♥(c) Finally, suppose your only restriction is that you may never move a disk directly from peg 1 to peg 2. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make? *[Hint: This variant is considerably harder to analyze than the other two.]*

5. A German mathematician developed a new variant of the Towers of Hanoi puzzle, known in the US literature as the "Liberty Towers" puzzle.[6] In this variant, there is a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the

[6]No it isn't.

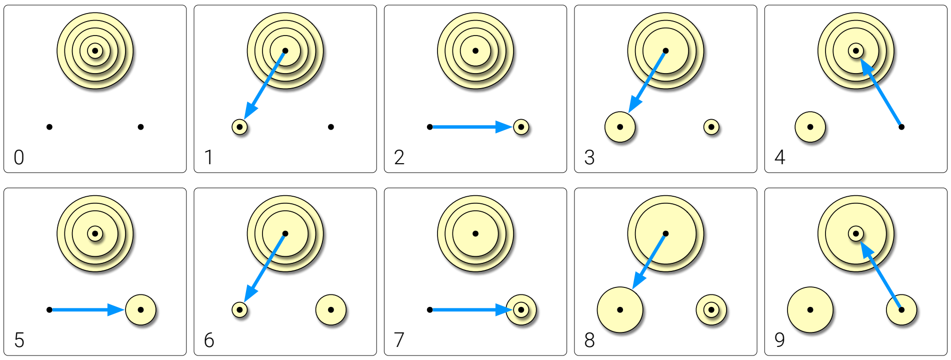


Figure 1.17. The first several moves in a counterclockwise Towers of Hanoi solution.

smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

(a) Describe a recursive algorithm for the case $k=3$. *Exactly* how many moves does your algorithm make? (This is the same as part (a) of the previous question.) Exam

(b) Describe a recursive algorithm for the case $k=n+1$ that requires at most $O(n^3)$ moves. *[Hint: Use part (a).]* Homework

(c) Describe a recursive algorithm for the case $k=n+1$ that requires at most $O(n^2)$ moves. *[Hint: Don't use part (a).]* Homework

(d) Describe a recursive algorithm for the case $k=\sqrt{n}$ that requires at most a polynomial number of moves. (What polynomial??) Fun Homework

(e) Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n? Research

Recursion Trees

6. Use recursion trees to solve each of the following recurrences.

$$\begin{array}{lll} A(n)=2A(n/4)+\sqrt{n} & B(n)=2B(n/4)+n & C(n)=2C(n/4)+n^2 \\ D(n)=3D(n/3)+\sqrt{n} & E(n)=3E(n/3)+n & F(n)=3F(n/3)+n^2 \\ G(n)=4G(n/2)+\sqrt{n} & H(n)=4H(n/2)+n & I(n)=4I(n/2)+n^2 \end{array}$$

7. Use recursion trees to solve each of the following recurrences.

(j) $J(n)=J(n/2)+J(n/3)+J(n/6)+n$

(k) $K(n)=K(n/2)+2K(n/3)+3K(n/4)+n^2$

(l) $L(n)=L(n/15)+L(n/10)+2L(n/6)+\sqrt{n}$

♥8. Use recursion trees to solve each of the following recurrences.

(m) $M(n) = 2M(n/2) + O(n \log n)$

(n) $N(n) = 2N(n/2) + O(n/\log n)$

(p) $P(n) = \sqrt{n}\,P(\sqrt{n}) + n$

(q) $Q(n) = \sqrt{2n}\,Q(\sqrt{2n}) + \sqrt{n}$

Sorting

Exam, easy

9. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.

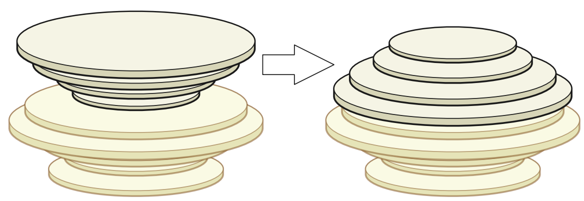


Figure 1.18. Flipping the top four pancakes.

(a) Describe an algorithm to sort an arbitrary stack of n pancakes using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

(b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

*[Hint: This problem has **nothing** to do with the Tower of Hanoi!]*

Homework

10. Prove that quicksort with the median-of-three heuristic requires $\Omega(n^2)$ time to sort an array of size n in the worst case. Specifically, for any integer n, describe a permutation of the integers 1 through n, such that in every recursive call to median-of-three-quicksort, the pivot is always the second smallest element of the array. Designing this permutation requires intimate knowledge of the PARTITION subroutine.

(a) As a warm-up exercise, assume that the PARTITION subroutine is *stable*, meaning it preserves the existing order of all elements smaller than the pivot, and it preserves the existing order of all elements smaller than the pivot.

(b) Assume that the PARTITION subroutine uses the specific algorithm listed on page 9, which is *not* stable.

11. (a) Prove that the following algorithm actually sorts its input. Homework, google

```
StoogeSort(A[0 .. n − 1]) :
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      StoogeSort(A[0 .. m − 1])
      StoogeSort(A[n − m .. n − 1])
      StoogeSort(A[0 .. m − 1])
```

(b) Would StoogeSort still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

(c) State a recurrence (including the base case(s)) for the number of comparisons executed by StoogeSort.

(d) Solve the recurrence, and prove that your solution is correct. *[Hint: Ignore the ceiling.]*

(e) Prove that the number of *swaps* executed by StoogeSort is at most $\binom{n}{2}$.

12. Consider the following cruel and unusual sorting algorithm, due to Gary Miller. Homework

```
Cruel(A[1 .. n]):
  if n > 1
      Cruel(A[1 .. n/2])
      Cruel(A[n/2 + 1 .. n])
      Unusual(A[1 .. n])
```

```
Unusual(A[1 .. n]):
  if n = 2
      if A[1] > A[2]                      ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                    ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i + n/4] ↔ A[i + n/2]
      Unusual(A[1 .. n/2])                ⟨⟨recurse on left half⟩⟩
      Unusual(A[n/2 + 1 .. n])            ⟨⟨recurse on right half⟩⟩
      Unusual(A[n/4 + 1 .. 3n/4])         ⟨⟨recurse on middle half⟩⟩
```

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove by induction that Cruel correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(b) Prove that Cruel would *not* correctly sort if we removed the for-loop from Unusual.

(c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

(d) What is the running time of UNUSUAL? Justify your answer.

(e) What is the running time of CRUEL? Justify your answer.

Homework, google

13. An ***inversion*** in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. *[Hint: Modify mergesort.]*

Long homework

14. (a) Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. *[Hint: See the previous problem.]*

(b) Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to part (a).]*

(c) Prove that your algorithm from part (b) actually runs in $O(n \log n)$ time.

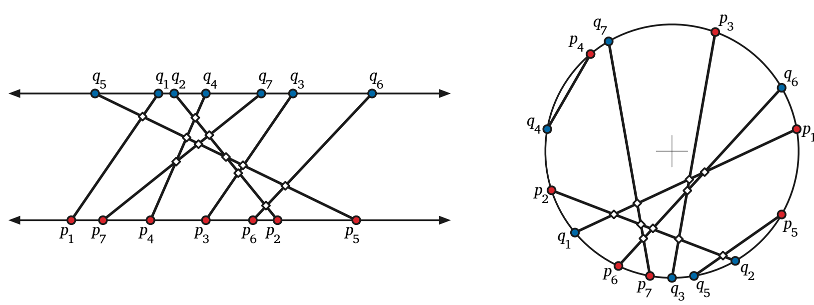


Figure 1.19. Eleven intersecting pairs of segments with endpoints on parallel lines, and ten intersecting pairs of segments with endpoints on a circle.

Homework

15. (a) Describe an algorithm that sorts an input array $A[1..n]$ by calling a subroutine SQRTSORT(k), which sorts the subarray $A\left[k+1..k+\sqrt{n}\right]$ in place, given an arbitrary integer k between 0 and $n - \sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SQRTSORT; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SQRTSORT in the worst case?

♣(b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SqrtSort, prove that no algorithm can sort using $o(f(n))$ calls to SqrtSort. (See Chapter ••**Lower Bounds**••.) Homework

(c) Now suppose SqrtSort is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.) Homework

Selection

16. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets Homework

- $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
- $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \le w(S)/2$ and $w(S_{>x}) \le w(S)/2$.

Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

17. Consider the generalization of the Blum-Floyd-Pratt-Rivest-Tarjan Select algorithm shown in Figure 1.20, which partitions the input array into $\lceil n/b \rceil$ blocks of size b, instead of $\lceil n/5 \rceil$ blocks of size 5, but is otherwise identical. In the pseudocode below, the necessary modifications are indicated in red.

(a) State a recurrence for the running time of Mom$_b$Select, assuming that b is a constant (so the subroutine MedianOfB runs in $O(1)$ time). In particular, how do the sizes of the recursive subproblems depend on the constant b? Consider even b and odd b separately. Exam

(b) What is the worst-case running time of Mom$_1$Select? *[Hint: This is a trick question.]* Exam

♥(c) What is the worst-case running time of Mom$_2$Select? *[Hint: This is an unfair question.]* Grad homework

♥(d) What is the worst-case running time of Mom$_3$Select? Finding an upper bound on the running time is straightforward; the hard part is showing that this analysis is actually tight. *[Hint: See problem 10.]* Upper bound: Exam
Tight: Homework

```
MomₕSelect(A[1 .. n], k):
  if n ≤ b²
      use brute force
  else
      m ← ⌈n/b⌉
      for i ← 1 to m
          M[i] ← MedianOfB(A[b(i − 1) + 1 .. bi])
      mom_b ← Mom_bSelect(M[1 .. m], ⌊m/2⌋)
      r ← Partition(A[1 .. n], mom_b)
      if k < r
          return Mom_bSelect(A[1 .. r − 1], k)
      else if k > r
          return Mom_bSelect(A[r + 1 .. n], k − r)
      else
          return mom_b
```

Figure 1.20. A parametrized family of selection algorithms; see problem 17.

♥(e) What is the worst-case running time of $\text{Mom}_4\text{Select}$? Again, the hard part is showing that the analysis cannot be improved.[7]

Upper bound: Ex
Tight: Homewor

Homework

(f) For any constants $b \geq 5$, the algorithm $\text{Mom}_b\text{Select}$ runs in $O(n)$ time, but different values of b lead to different constant factors. Let $M(b)$ denote the minimum number of comparisons required to find the median of b numbers. The exact value of $M(b)$ is known only for $b \leq 13$:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

For each b between 5 and 13, find an upper bound on the running time of $\text{Mom}_b\text{Select}$ of the form $T(n) \leq \alpha_b n$ for some explicit constant α_b. (For example, on page 15 we showed that $\alpha_5 \leq 16$.)

Homework

(g) Which value of b yields the smallest constant α_b? *[Hint: This is a trick question.]*

Exam. Yes, really.

18. Prove that the variant of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm shown in Figure 1.21, which uses an extra layer of small medians to choose the main pivot, runs in $O(n)$ time.

Exam

19. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

[7] The median of four elements is either the second smallest or the second largest. In 2014, Ke Chen and Adrian Dumitrescu proved that if we modify $\text{Mom}_4\text{Select}$ to find second-smallest elements when $k < n/2$ and second-largest elements when $k > n/2$, the resulting algorithm runs in $O(n)$ time! See their paper "Select with Groups of 3 or 4 Takes Linear Time" (WADS 2015, arXiv:1409.3600) for details.

```
MomomSelect(A[1 .. n], k):
  if n ≤ 81
      use brute force
  else
      m ← ⌈n/3⌉
      for i ← 1 to m
          M[i] ← MedianOf3(A[3i − 2 .. 3i])
      mm ← ⌈m/3⌉
      for j ← 1 to mm
          Mom[j] ← MedianOf3(M[3j − 2 .. 3j])
      momom ← MomomSelect(Mom[1 .. mm], ⌊mm/2⌋)
      r ← Partition(A[1 .. n], momom)
      if k < r
          return MomomSelect(A[1 .. r − 1], k)
      else if k > r
          return MomomSelect(A[r + 1 .. n], k − r)
      else
          return momom
```

Figure 1.21. Selection by median of medians of medians; see problem 18).

(b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1\,..\,n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

20. Describe an algorithm to compute the median of an array $A[1\,..\,5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a ***decision tree***: A binary tree where each internal node contains a comparison of the form "$A[i] \gtrless A[j]$?" and each leaf contains an index into the array. Homework

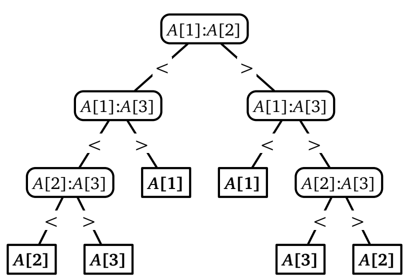


Figure 1.22. Finding the median of a 3-element array using at most 3 comparisons

Exam

21. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median element in the union of A and B in $\Theta(\log n)$ time. You can assume that the arrays contain no duplicate elements.

Homework, google

(b) Suppose we are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B$ in $\Theta(\log(m+n))$ time. For example, if $k=1$, your algorithm should return the smallest element of $A \cup B$.) *[Hint: Use your solution to part (a).]*

Homework

(c) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B \cup C$ in $O(\log n)$ time.

Homework

(d) Finally, suppose we are given a two dimensional array $A[1..m, 1..n]$ in which every row $A[i, \cdot]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Use the linear-time SELECT algorithm as a subroutine.]*

Arithmetic

Homework

22. In 1854, archaeologists discovered Babylonian clay tablets, carved around 2000BC, that list the squares of integers up to 59. This discovery led some scholars to conjecture that Babylonians performed multiplication by reduction to squaring, using an identity like $x \cdot y = (x^2 + y^2 - (x-y)^2)/2$. Unfortunately, those same scholars are silent on how the Babylonians supposedly squared larger numbers. Four thousand years later, we can finally rescue the ancient Babylonian mathematicians from their lives of drudgery through the power of recursion!

(a) Describe a variant of Karatsuba's algorithm that squares any n-digit number in $O(n^{\lg 3})$ time, by reducing to squaring three $\lceil n/2 \rceil$-digit numbers. (Karatsuba actually did this in 1960.)

(b) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 6})$ time, by reducing to squaring six $\lceil n/3 \rceil$-digit numbers.

♥(c) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 5})$ time, by reducing to squaring only *five* $(n/3 + O(1))$-digit numbers. *[Hint: What is $(a+b+c)^2 + (a-b+c)^2$?]*

Homework

23. (a) Describe and analyze a variant of Karatsuba's algorithm that multiplies any m-digit number and any n-digit number, for any $n \geq m$, in $O(nm^{\lg 3 - 1})$ time.

(b) Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time, using the algorithm from part (a) as a subroutine. (The standard algorithm that computes one digit at a time requires $\Theta(n^2)$ time.)

(c) Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time. *[Hint: Watch out for an extra log factor in the running time.]*

♥(d) Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n) \log n)$ time.

24. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n: Homework

```
FACTORIAL(n):
  if n = 0
      return 1
  else
      return n · FACTORIAL(n − 1)
```

(a) How many multiplications does this algorithm perform?

(b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. *[Hint: $(n/2)^{n/2} < n! < n^n$.]*

(c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of FACTORIAL. We can multiply any k-digit number and any l-digit number in $O(k \cdot l)$ time using the grade-school algorithm (or the Russian peasant algorithm). What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?

(d) The following algorithm also computes the factorial function, but using a different grouping of the multiplications:

```
FACTORIAL2(n, m):          ⟨⟨Compute n!/(n − m)!⟩⟩
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return FACTORIAL2(n, ⌊m/2⌋) · FACTORIAL2(n − ⌊m/2⌋, ⌈m/2⌉)
```

What is the running time of FACTORIAL2(n, n) if we use grade-school multiplication? *[Hint: Ignore the floors and ceilings.]*

(e) Describe and analyze a variant of Karastuba's algorithm that multiplies any k-digit number and any l-digit number, for any $k \geq l$, in $O(k \cdot l^{\lg 3 - 1}) = O(k \cdot l^{0.585})$ time.

♥(f) What are the running times of FACTORIAL(n) and FACTORIAL2(n, n) if we use the modified Karatsuba multiplication from part (e)?

25. The ***greatest common divisor*** of two positive integer x and y, denoted $\mathbf{gcd}(x, y)$, is the largest integer d such that both x/d and y/d are integers. Euclid described the Homework: (a)+(b) or (c)+(d) or (e)+(f)

following recursive algorithm[8] to compute $\gcd(x, y)$ in his *Elements*, written around 300BC:

```
EuclidGCD(x, y):
  if x = y
      return x
  else if x > y
      return EuclidGCD(x − y, y)
  else
      return EuclidGCD(x, y − x)
```

(a) Prove that EuclidGCD correctly computes $\gcd(x, y)$. Specifically:

  i. Prove that EuclidGCD(x, y) divides both x and y.

  ii. Prove that every divisor of x and y is also a divisor of EuclidGCD(x, y).

(b) What is the worst-case running time of EuclidGCD(x, y), as a function of x and y? (Assume that computing $x - y$ requires $O(\log x + \log y)$ time.)

(c) Prove that the following algorithm also computes $\gcd(x, y)$:

```
FastEuclidGCD(x, y):
  if x = y
      return x
  else if x > y
      return FastEuclidGCD(x mod y, y)
  else
      return FastEuclidGCD(x, y mod x)
```

(d) What is the worst-case running time of FastEuclidGCD(x, y), as a function of x and y? (Assume that computing $x \bmod y$ takes $O(\log x \cdot \log y)$ time.)

(e) Prove that the following algorithm also computes $\gcd(x, y)$:

```
BinaryGCD(x, y):
  if x = y
      return x
  else if x and y are both even
      return 2 · BinaryGCD(x/2, y/2)
  else if x is even
      BinaryGCD(x/2, y)
  else if y is even
      BinaryGCD(x, y/2)
  else if x > y
      return BinaryGCD((x − y)/2, y)
  else
      return BinaryGCD(x, (y − x)/2)
```

[8] Euclid's algorithm is often incorrectly described as the *first* recursive algorithm, or even the first non-trivial algorithm, but only because Western scholars have a culturally ingrained habit of fetishizing the Greeks, and therefore ignoring mere λογιστικός. In particular, the Egyptian duplation and mediation algorithm—which I claim is both nontrivial and recursive—predates Euclid by at least 1500 years, and that's *not* the most sophisticated algorithm documented during that era.

(f) What is the worst-case running time of FastEuclidGCD(x, y), as a function of x and y? (Assume that computing $x - y$ takes $O(\log x + \log y)$ time, and computing $z/2$ requires $O(\log z)$ time.)

Arrays

26. Suppose you are given a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed. Describe and analyze an algorithm to compute a tiling of the board by without gaps or overlaps by **L**-shaped tiles, each composed of 3 squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time. *[Hint: First prove that such a tiling always exists.]* Homework

27. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party by introducing them to each other. Members of the same political party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.[9]

    (a) Suppose more than half of the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party. Exam

    (b) Now suppose there are more than two parties, but one party has a *plurality*: more people belong to that party than to any other party. Present a practical procedure to precisely pick the people from the plurality political party as parsimoniously as possible, presuming the plurality party is composed of at least p people. Pretty please. Homework

28. Smullyan Island has three types of inhabitants: *knights* always speak the truth; *knaves* always lie; and *normals* sometimes speak the truth and sometimes don't. Everyone on the island knows everyone else's name and type (knight, knave, or normal). You want to learn the type of every inhabitant. Homework

    You can ask any inhabitant to tell you the type of any other inhabitant. Specifically, if you ask "Hey X, what is Y's type?" then X will respond as follows:

    - If X is a knight, then X will respond with Y's correct type.
    - If X is a knave, then X could respond with *either* of the types that Y is *not*.
    - If X is a normal, then X could respond with *any* of the three types.

[9] Of course, real-world politics is much messier than this simplified model, but this is a theory class!

The inhabitants will ignore any questions not of this precise form; in particular, you may not ask an inhabitant about their own type. Asking the same inhabitant the same question multiple times always yields the same answer, so there's no point in asking any question more than once.

(a) Suppose you know that a strict majority of inhabitants are knights. Describe an efficient algorithm to identify the type of every inhabitant.

(b) Prove that if at most half the inhabitants are knights, it is impossible to determine the type of every inhabitant.

Homework
Exam: (b)(d)(e)

29. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.

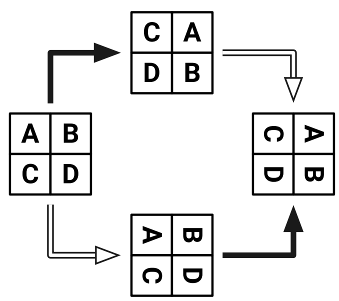


Figure 1.23. Two algorithms for rotating a pixel map.

(a) Prove that both versions of the algorithm are correct when n is a power of 2.

(b) *Exactly* how many blits does the algorithm perform when n is a power of 2?

(c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of 2. How many blits does your modified algorithm perform?

(d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(e) What if a $k \times k$ blit takes only $O(k)$ time?

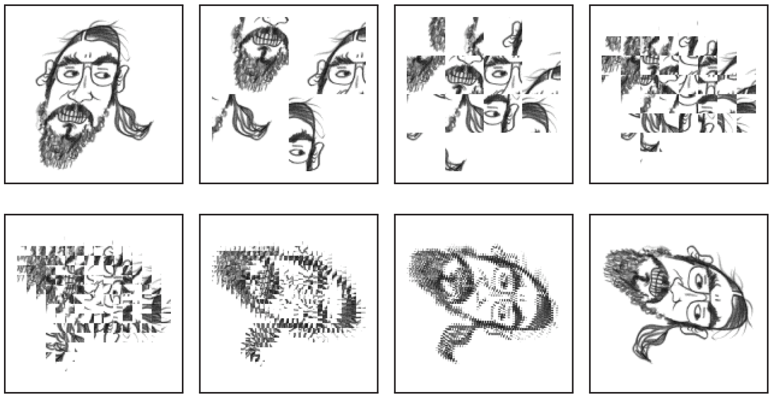


Figure 1.24. The first rotation algorithm (blit then recurse) in action.

30. An array $A[0\,..\,n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example, Exam

4	6	9	8	7	5	1	2	3	is bitonic, but
3	6	9	8	7	5	1	2	4	is *not* bitonic.

Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

31. Suppose we are given an array $A[1\,..\,n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$. Exam

    (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

    (b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

32. Suppose we are given an array $A[1\,..\,n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array: Exam

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time.

*[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

Exam

33. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that some prefix $A[1..k]$ is sorted in increasing order, the corresponding suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$.

    For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

    (a) Describe and analyze an algorithm to compute the unknown integer k.

    (b) Describe and analyze an algorithm to determine if the given array contains a given number x.

34. You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

    Exam

    (a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than either of its neighbors. How many boxes does your algorithm open in the worst case?

    Homework

    ♥(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

    Homework

    ♣♥(c) Prove that your solution to part (b) is optimal up to a constant factor. (See Chapter ••**Lower Bounds**••.)

Homework

35. (a) Let $n = 2^\ell - 1$ for some positive integer ℓ. Suppose someone claims to hold an unsorted array $A[1..n]$ of *distinct* ℓ-bit strings; thus, exactly one ℓ-bit string does *not* appear in A. Suppose further that the ***only*** way we can access A is by calling the function $\textsc{FetchBit}(i, j)$, which returns the jth bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in A using only $O(n)$ calls to $\textsc{FetchBit}$.

    Homework

    ♥(b) Now suppose $n = 2^\ell - k$ for some positive integers k and ℓ, and again we are given an array $A[1..n]$ of *distinct* ℓ-bit strings. Describe an algorithm to find the k strings that are missing from A using only $O(n \log k)$ calls to $\textsc{FetchBit}$.

Trees

36. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree.

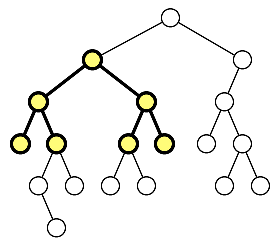


Figure 1.25. The largest complete subtree of this binary tree has depth 2.

37. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order: HWo
    - Preorder: `I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X`
    - Postorder: `H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I`

    Draw George's binary tree.

    (b) Recall that a binary tree is *full* if every non-leaf node has exactly two children. Exam
      i. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.
      ii. Prove that there is no algorithm to reconstruct an *arbitrary* binary tree from its preorder and postorder node sequences.

    (c) Describe and analyze a recursive algorithm to reconstruct an *arbitrary* binary tree, given its preorder and *inorder* node sequences as input. Exam

    (d) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence. Exam

    ♥(e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence, ***in $O(n)$ time***. Homework

    In parts (b)–(e), assume that all keys are distinct and that the input is consistent with at least one binary tree.

38. Suppose we have n points scattered inside a two-dimensional box. A ***kd-tree***[10] recursively subdivides the points as follows. First we split the box into two smaller Homework

[10]The term "kd-tree" (pronounced "kay *dee* tree") was originally an abbreviation for "k-dimensional tree", but more modern usage ignores this etymology, in part because nobody in their right mind would

boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point inside the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.

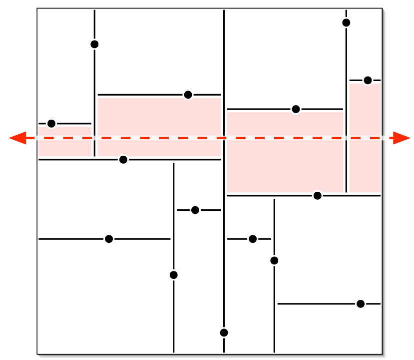


Figure 1.26. A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells are there, as a function of n? Prove your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(16) = 4$.]*

(c) Suppose we are given n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

(d) Describe an analyze an efficient algorithm that counts, given a kd-tree containing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

Exam

39. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a ***central*** vertex if each of these smaller trees has at most $n/2$ vertices.

Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree. *[Hint: First prove that every tree has a central vertex.]*

ever use the letter k to denote dimension instead of the *obviously* superior d. Etymological consistency would require calling the data structure in this problem a "2d-tree", or even a "2-d tree", but the standard nomenclature is now "two-dimensional kd-tree". See also: B-tree (maybe), alpha shape, beta skeleton, epsilon net, Potomac River, Mississippi River, Lake Michigan, Lake Tahoe, Manhattan Island, the La Brea Tar Pits, Sahara Desert, Mount Kilimanjaro, South Vietnam, East Timor, the Milky Way Galaxy, the City of Townsville, and self-driving automobiles.

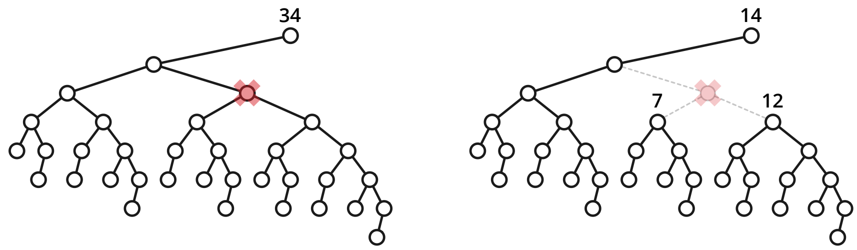


Figure 1.27. Deleting a central vertex in a 34-node binary tree, leaving subtrees with 14, 7, and 12 nodes.

♥40. Let T be a binary tree whose nodes store distinct numerical values. Recall that T is a ***binary search tree*** if and only if either (1) T is empty, or (2) T satisfies the following recursive conditions: Fun Homework

- The left subtree of T is a binary search tree.
- All values in the left subtree of T are smaller than the value at the root of T.
- The right subtree of T is a binary search tree.
- All values in the right subtree of T are larger than the value at the root of T.

Describe and analyze an algorithm to transform an *arbitrary* binary tree T with distinct node values into a binary search tree, using ***only*** the following operations:

- ***Rotate*** an arbitrary node upward, as shown in Figure 1.28.[11]
- ***Swap*** the left and right subtrees of an arbitrary node, as shown in Figure 1.29.

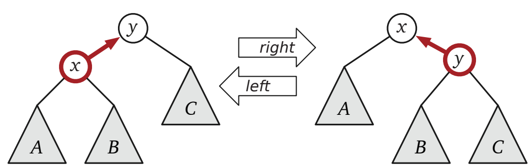


Figure 1.28. Left to right: right rotation at x. Right to left: left rotation at y.

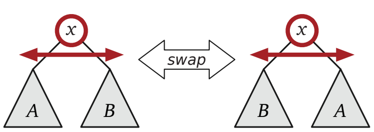


Figure 1.29. Swapping the subtrees of x

For both of these operations, some, all, or none of the subtrees A, B, and C shown in Figures 1.28 and 1.29 may be empty. Figure 1.30 shows a sequence of eight operations transforming a five-node binary tree into a binary search tree.

Your algorithm cannot directly modify parent or child pointers, and it cannot allocate new nodes or delete old nodes; the ***only*** way it can modify T is using

[11]Rotations preserve the inorder sequence of nodes in a binary tree. Partly for this reason, rotations are used to maintain several types of balanced binary search trees, including AVL trees, red-black trees, splay trees, scapegoat trees, and treaps. Some of these data structures are described in later chapters.

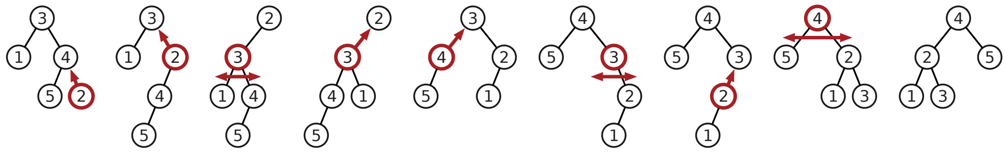


Figure 1.30. “Sorting” a binary tree: rotate 2, rotate 2, swap 3, rotate 3, rotate 4, swap 3, rotate 2, swap 4.

rotations and swaps. On the other hand, you may *compute* anything you like for free, as long as that computation does not modify T. In other words, the running time of your algorithm is *defined* to be the number of rotations and swaps that it performs.

For full credit, your algorithm should use as few rotations and swaps as possible in the worst case. *[Hint: $O(n^2)$ operations is not too difficult, but we can do better.]*

Fun Homework

♥41. Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversals of binary trees. Bob understands the basic idea behind the traversal algorithms, but whenever he tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob frantically submits code with the following structure:

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      [redacted]Order(left(v))
      [redacted]Order(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      [redacted]Order(left(v))
      print label(v)
      [redacted]Order(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      [redacted]Order(left(v))
      [redacted]Order(right(v))
      print label(v)
```

Each [redacted] in this pseudocode hides one of the prefixes Pre, In, or Post. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

PreOrder(*left*(v)) PreOrder(*right*(v))
InOrder(*left*(v)) InOrder(*right*(v))
PostOrder(*left*(v)) PostOrder(*right*(v))

Thus, there are precisely 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Now suppose you are given the output of Bob's traversal algorithms, executed on some ***unknown*** binary tree T. Bob's output has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. You may assume that these traversal sequences are consistent with exactly one binary tree T; in particular, the vertex labels of the unknown tree T are distinct, and every internal node in T has exactly two children.

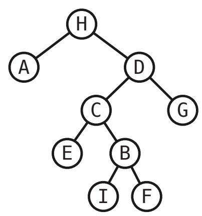
(a) Describe an algorithm to reconstruct the unknown tree T from the given traversal sequences.

(b) Describe an algorithm that either reconstructs Bob's code from the given traversal sequences, or correctly reports that the traversal sequences are consistent with more than one set of algorithms.

For example, given the input

$$Pre[1..n] = [\texttt{H A E C B I F G D}]$$
$$In[1..n] = [\texttt{A H D C E I F B G}]$$
$$Post[1..n] = [\texttt{A E I B F C D G H}]$$

your first algorithm should return the following tree:



and your second algorithm should reconstruct the following code:

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      PreOrder(left(v))
      PostOrder(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      PostOrder(left(v))
      print label(v)
      PreOrder(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      InOrder(left(v))
      InOrder(right(v))
      print label(v)
```