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

— Henry David Thoreau

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)

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

— Henry Ford

1 Recursion

1.1 Simplify and delegate

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem (or set of problems) Y means to write an algorithm for X, using an algorithm or Y as a subroutine or black box.

For example, the congressional apportionment algorithm described in Lecture 0 reduces the problem of apportioning Congress to the problem of maintaining a priority queue under the operations INSERT and EXTRACTMAX. Those data structure operations are black boxes; the apportionment algorithm does not depend on any specific implementation. Conversely, when we design a particular priority queue data structure, we typically neither know nor care how our data structure will be used. Whether or not the Census Bureau plans to use our code to apportion Congress is completely irrelevant to our design choices. As a general rule, when we design algorithms, we may not know—*and we should not care*—how the basic building blocks we use are implemented, or how your algorithm might be used as a basic building block to solve a bigger problem.

A particularly powerful kind of reduction is *recursion*, which can be defined loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more ***simpler instances of the same problem***.

If the 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. 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 the simpler subproblems.[1]

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly, namely, that there is no infinite sequence of reductions to 'simpler' and 'simpler' subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that can be solved by some other method; otherwise, the recursive algorithm will never terminate. This finiteness condition is almost always satisfied trivially, but we should always be wary of 'obvious' recursive algorithms that actually recurse forever.[2]

[1] I used to refer to 'elves' instead of the Recursion Fairy, referring to the traditional fairy tale about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves have finished everything overnight. Someone more entheogenically experienced than I might recognize them as Terence McKenna's 'self-adjusting machine elves'.

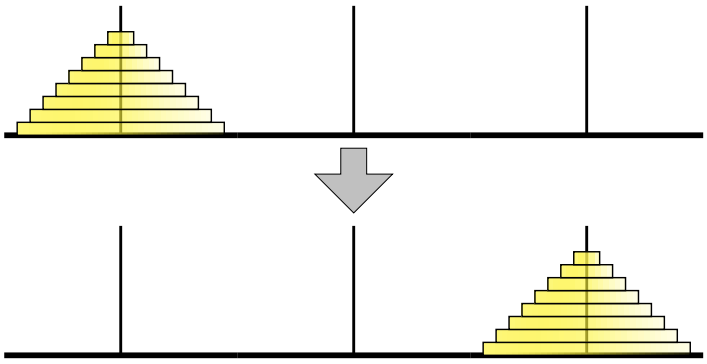
[2] All too often, 'obvious' is a synonym for 'false'.

1.2 Tower of Hanoi

The Tower of Hanoi puzzle was first published by the mathematician François Éduoard Anatole Lucas in 1883, under the pseudonym 'N. Claus (de Siam)' (an anagram of 'Lucas d'Amiens'). 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.

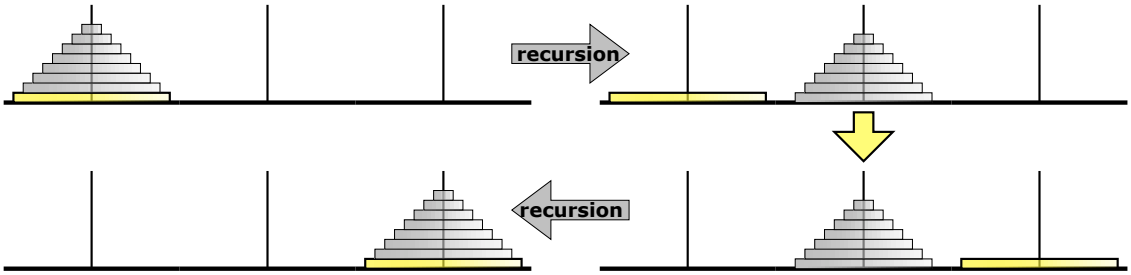
Of course, being good computer scientists, we read this story and immediately substitute n for the hardwired constant sixty-four. How can we move a tower of n disks from one needle to another, using a third needles as an occasional placeholder, never placing any disk on top of a smaller disk?



The Tower of Hanoi puzzle

The trick 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 needle before we can move the nth disk. And then after we move the nth 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...

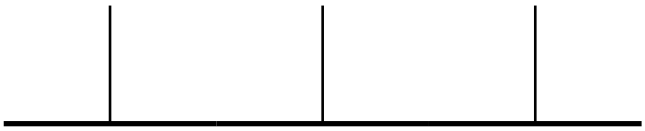
STOP!! That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy (or, to carry the original story further, to the junior monks at the temple).



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

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

Our algorithm does make one subtle but 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 case directly. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one needle to another.



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

While it's tempting to think about how all those smaller disks get moved—in other words, what happens when the recursion is unfolded—it's not necessary. In fact, for more complicated problems, unfolding the recursive calls is merely distracting. Our *only* task is to reduce the problem 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 (or more formally, the inductive hypothesis implies that our algorithm correctly moves) the top $n-1$ disks, so our algorithm is clearly correct.

Here's the recursive Hanoi algorithm in more typical pseudocode.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n, src, tmp, dst)
      move disk n from src to dst
      HANOI(n, tmp, dst, src)
```

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 $n \geq 1$. The annihilator method lets us quickly derive a closed form solution $\boldsymbol{T(n) = 2^n - 1}$. 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 before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

1.3 MergeSort

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

1. Divide the array $A[1..n]$ into two subarrays $A[1..m]$ and $A[m+1..n]$, where $m = \lfloor n/2 \rfloor$.
2. Recursively mergesort the subarrays $A[1..m]$ and $A[m+1..n]$.
3. Merge the newly-sorted subarrays $A[1..m]$ and $A[m+1..n]$ into a single sorted list.

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	S	R	T	X

A Mergesort example.

The first step is completely trivial—we only need to compute the median index m—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 $A[1..m]$ and $A[m+1..n]$ can be merged using a simple linear-time algorithm. Here's a complete specification of the Mergesort algorithm; for simplicity, we separate out the merge step as a subroutine.

```
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]
```

To prove that the algorithm is correct, we use our old friend induction. We can prove that MERGE is correct using induction on the total size of the two subarrays $A[i..m]$ and $A[j..n]$ left to be merged into $B[k..n]$. The base case, where at least one subarray is empty, is straightforward; the algorithm just copies it into B. Otherwise, the smallest remaining element is either $A[i]$ or $A[j]$, since both subarrays are sorted, so $B[k]$ is assigned correctly. The remaining subarrays—either $A[i+1..m]$ and $A[j..n]$, or $A[i..m]$ and $A[j+1..n]$—are merged correctly into $B[k+1..n]$ by the inductive hypothesis.[4] This completes the proof.

Now we can prove MERGESORT correct by another round of straightforward induction. The base cases $n \le 1$ are trivial. Otherwise, by the inductive hypothesis, the two smaller subarrays $A[1..m]$ and $A[m+1..n]$ are sorted correctly, and by our earlier argument, merged into the correct sorted output.

What's the running time? Since we have a recursive algorithm, we're going to get a recurrence of some sort. MERGE clearly takes linear time, since it's a simple for-loop with constant work per iteration. We get the following recurrence for MERGESORT:

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

Aside: Domain Transformations. Except for the floor and ceiling, this recurrence falls firmly into the "all levels equal" case of the recursion tree method, or its corollary, the Master Theorem. If we simply ignore the floor and ceiling, the method suggests the solution $T(n) = O(n \log n)$. We can easily check that this answer is correct using induction, but there is a simple method for solving recurrences like this directly, called *domain transformation*.

First we overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \le 2T\big(\lceil n/2 \rceil\big) + O(n) \le 2T(n/2+1) + O(n).$$

Now we define a new function $S(n) = T(n+\alpha)$, where α is a constant chosen so that $S(n)$ satisfies the familiar recurrence $S(n) \le 2S(n/2) + O(n)$. To figure out the appropriate value for α, we compare two

[4] "The inductive hypothesis" is just a technical nickname for our friend the Recursion Fairy.

versions of the recurrence for $T(n+\alpha)$:

$$S(n) \le 2S(n/2) + O(n) \quad \Longrightarrow \quad T(n+\alpha) \le 2T(n/2+\alpha) + O(n)$$
$$T(n) \le 2T(n/2+1) + O(n) \quad \Longrightarrow \quad T(n+\alpha) \le 2T((n+\alpha)/2+1) + O(n+\alpha)$$

For these two recurrences to be equal, we need $n/2+\alpha = (n+\alpha)/2+1$, which implies that $\alpha = 2$. The recursion tree method tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

We can use domain transformations to remove floors, ceilings, and lower order terms from any recurrence. But now that we realize this, we don't need to bother grinding through the details ever again!

1.4 Quicksort

Quicksort was 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. Split the array into three subarrays containing the items less than the pivot, the pivot itself, and the items bigger 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:	M	A	E	G	I	L	N	R	X	O	S	P	T
Recurse:	A	E	G	I	L	M	N	O	P	S	R	T	X

A Quicksort example.

Here's a more formal specification of the Quicksort algorithm. The separate PARTITION subroutine takes the original position of the pivot element as input and returns the post-partition pivot position as output.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      k ← PARTITION(A, p)
      QUICKSORT(A[1..k−1])
      QUICKSORT(A[k+1..n])
```

```
PARTITION(A[1..n], p):
  if (p ≠ n)
      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]
  if (i ≠ n)
      swap A[i] ↔ A[n]
  return i
```

Just as we did for mergesort, we need two induction proofs to show that QUICKSORT is correct—weak induction to prove that PARTITION correctly partitions the array, and then straightforward strong induction to prove that QUICKSORT correctly sorts assuming PARTITION is correct. I'll leave the gory details as an exercise for the reader.

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 k, the rank of the chosen pivot:

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

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

$$T(n) = 2T\left(\lceil n/2 \rceil - 1\right) + T\left(\lfloor n/2 \rfloor\right) + O(n) \leq 2T(n/2) + O(n),$$

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

In fact, it is possible to locate the median element in an unsorted array in linear time. However, the algorithm is fairly complicated, and the hidden constant in the $O()$ notation is quite large. So in practice, programmers settle for something simple, like choosing the first or last element of the array. In this case, k can be anything from 1 to n, so we have

$$T(n) = \max_{1 \leq k \leq n} \left(T(k-1) + T(n-k) + O(n)\right)$$

In the worst case, the two subproblems are completely unbalanced—either $k = 1$ or $k = n$—and the recurrence becomes $T(n) \leq T(n-1) + O(n)$. The solution is $T(n) = O(n^2)$. Another common heuristic is 'median of three'—choose three elements (usually at the beginning, middle, and end of the array), and take the middle one as the pivot. Although this is better in practice than just choosing one element, we can still have $k = 2$ or $k = n - 1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \leq T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element will 'usually' fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This suggests that the *average-case* running time is $O(n \log n)$. Although this intuition is correct, we are still far from a *proof* that quicksort is usually efficient. We will formalize this intuition about average cases in a later lecture.

1.5 The Pattern

Both mergesort and and quicksort follow the same general three-step pattern of all divide and conquer algorithms:

1. **Divide** the problem into several *smaller independent* subproblems.
2. **Delegate** each subproblem to the Recursion Fairy to get a sub-solution.
3. **Combine** the sub-solutions together into the final solution.

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 usually involves strong induction. Analyzing the running time requires setting up and solving a recurrence, which often (but unfortunately not always!) can be solved using recursion trees (or, if you insist, the Master Theorem), perhaps after a simple domain transformation.

1.6 Median Selection

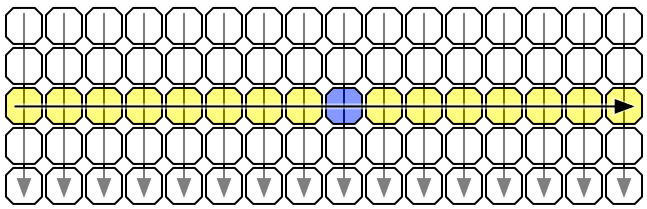
So, how *do* we find the median element of an array in linear time? The following algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth largest element in an array, using the following recursive divide-and-conquer strategy. The subroutine PARTITION is the same as the one used in QUICKSORT.

```
SELECT(A[1..n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          B[i] ← SELECT(A[5i − 4..5i], 3)        ⟨⟨Brute force!⟩⟩
      mom ← SELECT(B[1..m], ⌊m/2⌋)               ⟨⟨Recursion!⟩⟩
      r ← PARTITION(A[1..n], mom)
      if k < r
          return SELECT(A[1..r − 1], k)          ⟨⟨Recursion!⟩⟩
      else if k > r
          return SELECT(A[r + 1..n], k − r)      ⟨⟨Recursion!⟩⟩
      else
          return mom
```

If the input array is too large to handle by brute force, we divide it 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 find the median of each block by brute force and collect those medians into a new array. Then we recursively compute the median of the new array (the median of medians — hence 'mom') and use it to partition the input array. Finally, either we get lucky and the median-of-medians is the kth largest element of A, or we recursively search one of the two subarrays.

The key insight is that these two subarrays cannot be too large or too small. The median-of-medians is larger than $\lceil \lceil n/5 \rceil /2 \rceil - 1 \approx n/10$ medians, and each of those medians is larger than two other elements in its block. In other words, the median-of-medians 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 final recursive call searches an array of size $7n/10$.

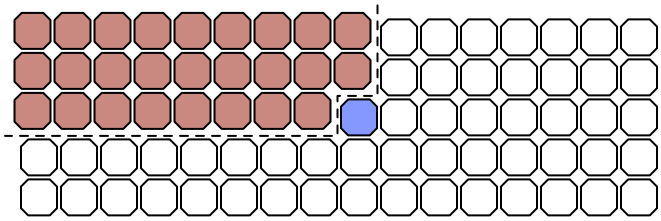
We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let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



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. A symmetric argument applies when our target element is smaller than the median-of-medians.



Discarding approximately 3/10 of the array

We conclude that the worst-case running time of the algorithm obeys the following recurrence:

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

The recursion tree method implies the solution $T(n) = O(n)$.

A finer analysis reveals that the hidden constants are quite large, even if we count only comparisons; this is not a practical algorithm for small inputs. (In particular, mergesort uses fewer comparisons in the worst case when $n < 4{,}000{,}000$.) Selecting the median of 5 elements requires 6 comparisons, so we need $6n/5$ comparisons to set up the recursive subproblem. We need another $n-1$ comparisons to partition the array after the recursive call returns. So the actual recurrence is

$$T(n) \leq 11n/5 + T(n/5) + T(7n/10).$$

The recursion tree method implies the upper bound

$$T(n) \leq \frac{11n}{5} \sum_{i \geq 0} \left(\frac{9}{10}\right)^i = \frac{11n}{5} \cdot 10 = 22n.$$

1.7 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the standard iterative 'ripple-carry' algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? At least in the United States, every grade school student (supposedly) learns to multiply by breaking the problem into n one-digit multiplications and n additions:

```
   31415962
× 27182818
  251327696
   31415962
 251327696
  62831924
251327696
 31415962
219911734
62831924
---------
853974377340916
```

We could easily formalize this algorithm as a pair of nested for-loops. The algorithm runs in $O(n^2)$ time—altogether, there are $O(n^2)$ digits in the partial products, and for each digit, we spend constant time.

We can get a more efficient algorithm by 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
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      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
```

You can easily prove by induction that this algorithm is correct. The running time for this algorithm is given by the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1,$$

which solves to $T(n) = O(n^2)$ by the recursion tree method (after a simple domain transformation). Hmm... I guess this didn't help after all.

But there's a trick, first published by Anatoliĭ Karatsuba in 1962.[5] We can compute the middle coefficient $bc + ad$ using only *one* recursive multiplication, by exploiting yet another bit of algebra:

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

This trick lets use replace the last three lines in the previous algorithm as follows:

```
FASTMULTIPLY(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      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
```

[5] However, the same basic trick was used non-recursively by Gauss in the 1800s to multiply complex numbers using only three real multiplications.

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

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1.$$

After a domain transformation, we can plug this into a recursion tree to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, a significant improvement over our earlier quadratic-time algorithm.[6]

Of course, in practice, all this is done in binary instead of decimal.

We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to get even faster multiplication algorithms. Ultimately, this idea leads to the development of the *Fast Fourier transform*, a more complicated divide-and-conquer algorithm that can be used to multiply two n-digit numbers in $O(n \log n)$ time.[7] We'll talk about Fast Fourier transforms in detail in another lecture.

1.8 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). All we really require is that a belong to a multiplicative group.[8] 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.

[6] Karatsuba actually 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 a bit messier: $a-b$ and $c-d$ are still m-digit numbers, but $a+b$ and $c+d$ might have $m+1$ digits. The simplification presented here is due to Donald Knuth.

[7] This fast algorithm for multiplying integers using FFTs was discovered by Arnold Schönhange and Volker Strassen in 1971. The $O(n \log n)$ running time requires the standard assumption that $O(\log n)$-bit integer arithmetic can be performed in constant time; the number of bit operations is $O(n \log n \log\log n)$.

[8] A *multiplicative group* $(G, \otimes)$ is a set G and a function $\otimes : G \times G \to G$, satisfying three axioms:

1. There is a *unit* element $1 \in G$ such that $1 \otimes g = g \otimes 1$ for any element $g \in G$.
2. Any element $g \in G$ has a *inverse* element $g^{-1} \in G$ such that $g \otimes g^{-1} = g^{-1} \otimes g = 1$
3. The function is *associative*: for any elements $f, g, h \in G$, we have $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

```
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 $T(n) \leq T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, recursion trees 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 ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). Nobody knows of an efficient algorithm that always uses the minimum possible number of multiplications.

Exercises

1. (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:

   - 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) Describe and analyze a recursive algorithm for reconstructing a binary tree, given its preorder and postorder node sequences.

   (c) Describe and analyze a recursive algorithm for reconstructing a binary tree, given its preorder and *inorder* node sequences.

2. Prove that the recursive Tower of Hanoi algorithm is *exactly equivalent* to each of the following non-recursive algorithms; in other words, prove that all three algorithms move the same disks, to and from the same needles, in the same order. The needles are labeled 0, 1, and 2, and our problem is to move a stack of n disks from needle 0 to needle 2 (as shown on page 2).

   (a) Follow these four rules:

   - Never move the same disk twice in a row.
   - If n is even, always move the smallest disk forward ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
   - If n is odd, always move the smallest disk backward ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).
   - When there is no move that satisfies the other rules, the puzzle is solved.

   (b) Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. 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 bit in the binary representation of n.) The function $\rho(n)$ is sometimes called the 'ruler' function, because its behavior resembles the marks on a ruler:

   $$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 6, 1, 2, 1, 3, 1, \ldots.$$

   Here's the non-recursive algorithm in one line:

   In step i, move disk $\rho(i)$ forward if $n - i$ is even, backward if $n - i$ is odd.

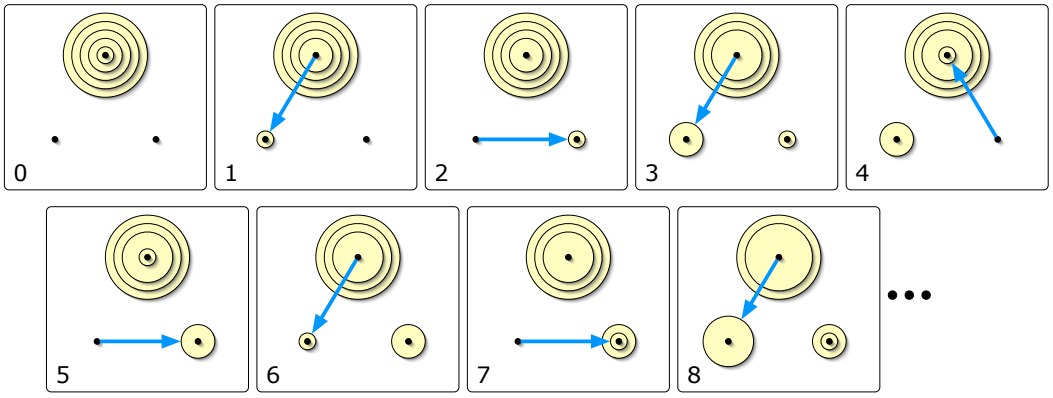
   When this rule requires us to move disk $n + 1$, the algorithm ends.

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

   (a) Suppose you are forbidden to move any disk directly between needle 1 and needle 2; *every* move must involve needle 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?

(b) Suppose you are only allowed to move disks from needle 0 to needle 2, from needle 2 to needle 1, or from needle 1 to needle 0. Equivalently, Suppose the needles 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. *Exactly* how many moves does your algorithm make?

★(c) Finally, suppose your only restriction is that you may never move a disk directly from needle 1 to needle 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 is considerably harder to analyze than the other two variants.]*



A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

4. (a) Prove that the following algorithm actually sorts its input.

```
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.

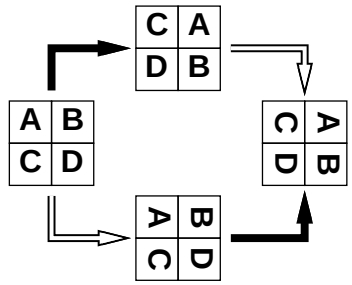
(d) Solve the recurrence, and prove that your solution is correct. *[Hint: Ignore the ceiling.]*

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

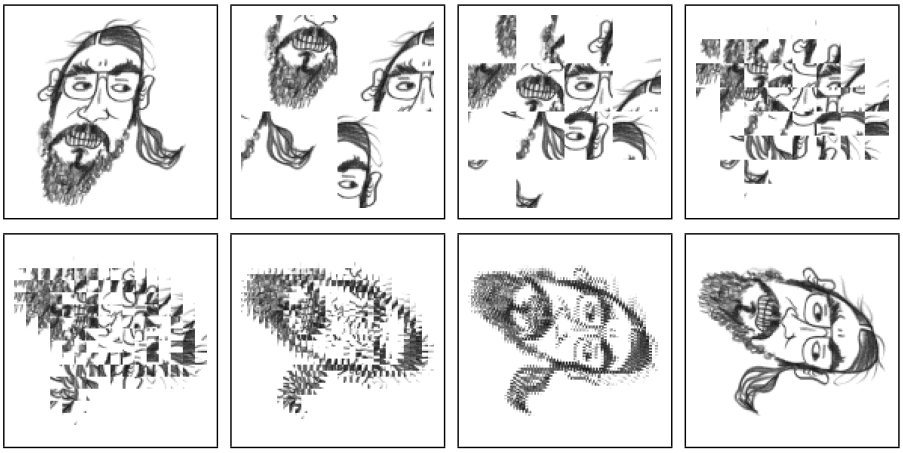
5. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `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. Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.



Two algorithms for rotating a pixel map.
Black arrows indicate blitting the blocks into place; white arrows indicate recursively rotating the blocks.



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

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

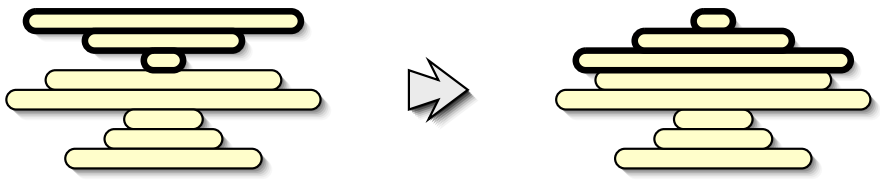
(b) *Exactly* how many blits does the algorithm perform when n is a power of two?

(c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of two. 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?

6. 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.



Flipping the top three 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?

7. 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.

(a) Suppose $m = 1$. 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?

*(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?

*(c) Prove that your solution to part (b) is optimal up to a constant factor.

8. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log n)$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, your algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicate elements. *[Hint: First solve the special case $k = n$.]*

(b) 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 in $O(\log n)$ time.

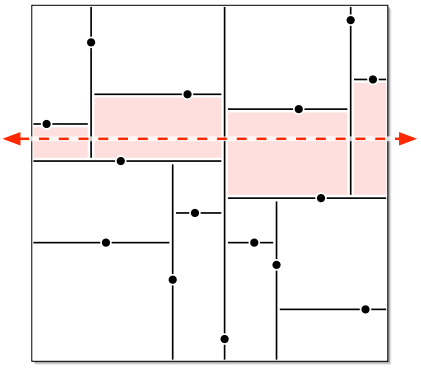
(c) Finally, suppose we are given a two dimensional array $A[1..m][1..n]$ in which every row $A[i][\,]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Use the linear-time*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

9. (a) Describe and analyze an algorithm to sort an array $A[1..n]$ by calling a subroutine SQRTSORT(k), which sorts the subarray $A[k+1..k+\sqrt{n}]$ 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.

(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.)

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



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.

(c) Suppose we have 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 storing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

11. You are at a political convention with n delegates, each one 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 two delegates belong to the *same* party or not by introducing them to each other—members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

(a) Suppose a majority (more than half) of the delegates are from the same political party. Describe an efficient algorithm that identifies a member (*any* member) of the majority party.

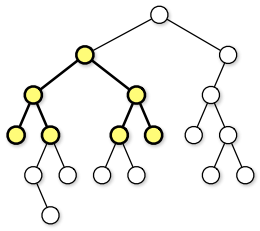
(b) Now suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality political party as parsimoniously as possible. (Please.)

12. The median of a set of size n is its $\lceil n/2 \rceil$th largest element, that is, the element that is as close as possible to the middle of the set in sorted order. In this lecture, we saw a fairly complicated algorithm to compute the median in $O(n)$ time.

    During your lifelong quest for a simpler linear-time median-finding algorithm, you meet and befriend the Near-Middle Fairy. Given any set X, the Near-Middle Fairy can find an element $m \in X$ that is *near* the middle of X in $O(1)$ time. Specifically, at least a third of the elements of X are smaller than m, and at least a third of the elements of X are larger than m.

    Describe and analyze a simple recursive algorithm to find the median of a set in $O(n)$ time if you are allowed to ask the Near-Middle Fairy for help.

13. 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 the root and the depth of this subtree.



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

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

```
FACTORIAL(n):
  if n = 0
      return 0
  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: Use Stirling's approximation:* $n! \approx \sqrt{2\pi n} \cdot (n/e)^n$*.]*

    (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. The grade-school multiplication algorithm takes $O(k \cdot l)$ time to multiply a k-digit number and an l-digit number. What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?

    (d) The following algorithm also computes $n!$, but groups the multiplication differently:

```
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 can multiply any k-digit number and any l-digit number, where $k \geq l$, 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)?