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)

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

1 Recursion

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 by magic.

For example, the Huntington-Hill algorithm described in Lecture 0 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.

As a general rule, when you design algorithms, you may not know how the basic building blocks you use are implemented, or how your algorithms might be used as building blocks to solve even bigger problems. Even when you *do* know how the black boxes work, you should pretend that you don't.

1.2 Simplify and Delegate

Recursion is a a particularly powerful kind of reduction, which can be described 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. 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 Freakin' Business So Butt Out.[1]

[1]When I was a student, I used to blame recursion on "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

Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name, the Induction Hypothesis.

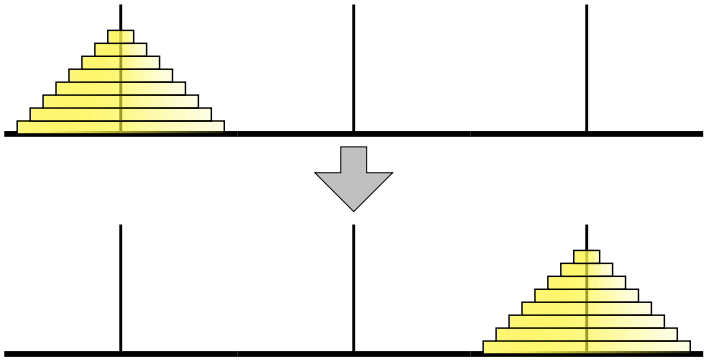
There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to 'simpler' and 'simpler' subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that can be solved by some other method; otherwise, the recursive algorithm will loop forever. This finiteness condition is almost always satisfied trivially, but we should always be wary of 'obvious' recursive algorithms that actually recurse forever. (All too often, 'obvious' is a synonym for 'false'.)

1.3 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:[2]

> *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, as good computer scientists, our first instinct on reading this story is to sustitute the variable n for the hardwired constant 64. How can we move a tower of n disks from one needle to another, using a third needle as an occasional placeholder, without ever placing a disk on top of a smaller disk?



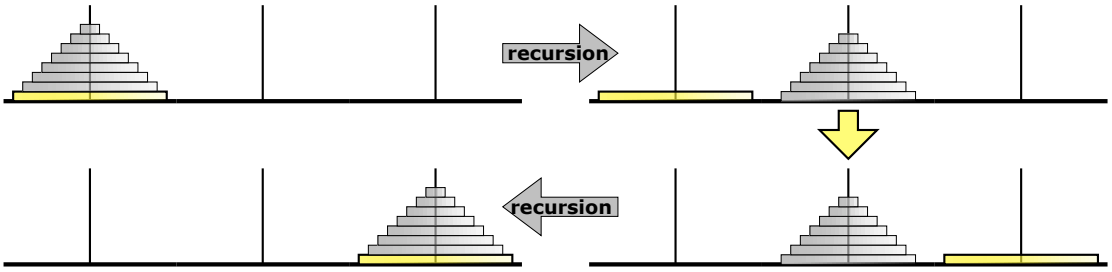
The Tower of Hanoi puzzle

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...

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

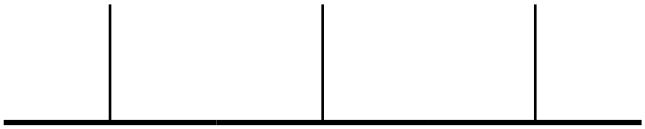
[2]This English translation is from W. W. Rouse Ball and H. S. M. Coxeter's book *Mathematical Recreations and Essays*.

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

Our recursive reduction 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 in no time at all.



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—or more generally, what happens when the recursion is unrolled—it's not necessary. For even slightly more complicated algorithms, unrolling the recursion is far more confusing than illuminating. 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. This algorithm moves a stack of n disks from a source needle (*src*) to a destination needle (*dst*) using a third temporary needle (*tmp*) as a placeholder.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)
      move disk n from src to dst
      HANOI(n − 1, 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 (or guessing and checking by induction) quickly gives us the 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.

1.4 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

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. Here's a complete description of the algorithm; to keep the recursive structure clear, we separate out the merge step as an independent 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 this algorithm is correct, we apply our old friend induction twice, first to the MERGE subroutine then to the top-level MERGESORT algorithm.

- We prove MERGE is correct by induction on $n - k + 1$, which is the total size of the two sorted subarrays $A[i..m]$ and $A[j..n]$ that remain to be merged into $B[k..n]$ when the kth iteration of the main loop begins. There are five cases to consider. Yes, five.
  - If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.)
  - If $i \le m$ and $j > n$, the subarray $A[j..n]$ is empty. Because both subarrays are sorted, the smallest element in the union of the two subarrays is $A[i]$. So the assignment $B[k] \leftarrow A[i]$ is correct. The inductive hypothesis implies that the remaining subarrays $A[i+1..m]$ and $A[j..n]$ are correctly merged into $B[k+1..n]$.
  - Similarly, if $i > m$ and $j \le n$, the assignment $B[k] \leftarrow A[j]$ is correct, and The Recursion Fairy correctly merges—sorry, I mean the inductive hypothesis implies that the MERGE algorithm correctly merges—the remaining subarrays $A[i..m]$ and $A[j+1..n]$ into $B[k+1..n]$.

  - If $i \le m$ and $j \le n$ and $A[i] < A[j]$, then the smallest remaining element is $A[i]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly merges the rest of the subarrays.
  - Finally, if $i \le m$ and $j \le n$ and $A[i] \ge A[j]$, then the smallest remaining element is $A[j]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly does the rest.

- Now we prove MERGESORT correct by induction; there are two cases to consider. Yes, two.
  - If $n \le 1$, 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 the previous argument).

What's the running time? Because the MERGESORT algorithm is recursive, its running time will be 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:

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

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

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

The "all levels equal" case of the recursion tree method now immediately implies the closed-form solution $\boldsymbol{T(n) = O(n \log n)}$. (Recursion trees and domain transformations are described in detail in a separate note on solving recurrences.)

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	E	I	L	N	R	O	X	S	M	P	T
Recurse:	A	E	G	I	L	M	N	O	P	R	S	T	X

A quicksort example.

Here's a more detailed description of the algorithm. 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.

```
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):
  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 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 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 r, the *rank* of the chosen pivot element:

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

If we could 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

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

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

In fact, as we will see later, 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. In practice, 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

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

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 will '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 is $O(n \log n)$. 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 lecture.

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 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 n-element array, for any integer k between 1 and n, using a variant of an algorithm called either "quick select" or "one-armed quicksort". The 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]
  elseChoose 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.

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

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

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

As with quicksort, we get the solution $T(n) = O(n^2)$ when $\ell = n-1$, which happens when the chosen pivot element is either the smallest element or largest element of the array.

On the other hand, we could avoid this quadratic behavior if we could somehow magically choose a *good* pivot, where $\ell \leq \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

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

This recurrence expands into a descending geometric series, which is dominated by its largest term, so $T(n) = O(n)$.

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.

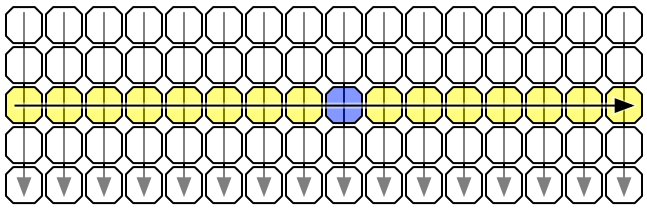
```
BFPRTSELECT(A[1..n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MEDIANOFFIVE(A[5i − 4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← SELECT(M[1..m], ⌊m/2⌋)             ⟨⟨Recursion!⟩⟩

      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
```

The recursive structure of the algorithm requires a slightly larger base case. There's absolutely nothing special about the constant 25 in the pseudocode; for theoretical purposes, any other constant like 42 or 666 or 8765309 would work just as well.

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 $M[1..\lceil n/5 \rceil]$. Then we recursively compute the median of this new array. Finally we use the median of medians — hence 'mom' — as the pivot in one-armed quicksort.

The key insight is that neither of these two subarrays can be too large. The median of medians is larger than $\lceil \lceil n/5 \rceil / 2 \rceil - 1 \approx n/10$ block medians, and each of those medians is larger than two other elements in its block. Thus, mom is larger than at least $3n/10$ elements in the input array, and 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 $7n/10$.

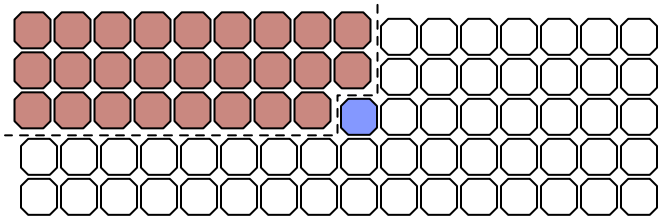
We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that *the algorithm does not 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 $T(n) = O(n)$.

Finer analysis reveals that the constant hidden by the $O()$ is 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 at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. We need another $n - 1$ comparisons to partition the array after the recursive call returns. So a more accurate recurrence for the total number of comparisons is

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

The recursion tree method implies the upper bound

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

1.8 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 $\Theta(n^2)$ time—altogether, there are $\Theta(n^2)$ digits in the partial products, and for each digit, we spend constant time. The Egyptian/Russian peasant multiplication algorithm described in the first lecture also runs in $\Theta(n^2)$ time.

Perhaps 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

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

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

In the mid-1950s, the famous Russian mathematician Andrey Kolmogorov conjectured that there is *no* algorithm to multiply two n-digit numbers in $o(n^2)$ time. However, in 1960, after Kolmogorov posed his conjecture at a seminar at Moscow University, Anatoliĭ Karatsuba, one of the students in the seminar, discovered 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 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
```

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.[3] Karastuba'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 instead of decimal.

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 to the development of the *Fast Fourier transform*, which we discuss in detail in the next lecture note. The fastest multiplication algorithm known, published by Martin Fürer in 2007 and based on FFTs, runs in $n \log n 2^{O(\log^* n)}$ time. Here, $\log^* n$ is 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.

1.9 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.

[3] 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. The same basic trick was used (non-recursively) by Gauss in the 1800s to multiply complex numbers using only three real 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.[4] 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 $T(n) \leq T(\lfloor n/2 \rfloor) + 2$, 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 ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). It is an open question whether the absolute minimum number of multiplications for a given exponent n can be computed efficiently.

Exercises

1. Prove that the Russian peasant multiplication algorithm runs in $\Theta(n^2)$ time.

2. (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**

[4] 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 $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

Draw George's binary tree.

(b) Prove that there is no algorithm to reconstruct an *arbtirary* binary tree from its preorder and postorder node sequences.

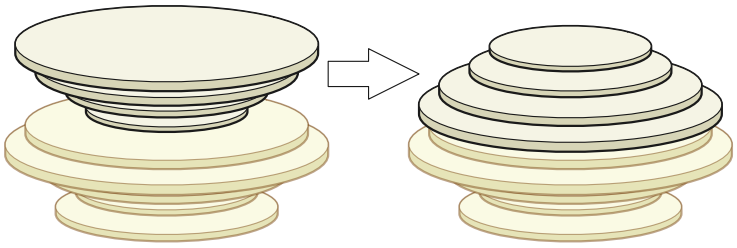
(c) Recall that a binary tree is *full* if every non-leaf node has exactly two children. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.

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

(e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence. Assume all input keys are distinct. For extra credit, describe an algorithm that runs in $O(n)$ time.

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

3. 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 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?

4. Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed.

(a) Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces, each composed of 3 squares.

(b) Describe and analyze an algorithm to compute such a tiling, given 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.

5. Prove that the original 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 exactly the same sequence of 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) Repeatedly make the only legal move that satisfies the following constraints:
- 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 ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).

If there is no move that satisfies these three constraints, the puzzle is solved.

(b) Start by putting your finger on the top of needle 0. Then repeat the following steps:
i. If n is odd, move your finger to the next needle ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
ii. If n is even, move your finger to the previous needle ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).
iii. Make the only legal move that does not require you to lift your finger. If there is no such move, the puzzle is solved.

(c) 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 in the binary representation of n.) Because its behavior resembles the marks on a ruler, $\rho(n)$ is sometimes called the *ruler function*:

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

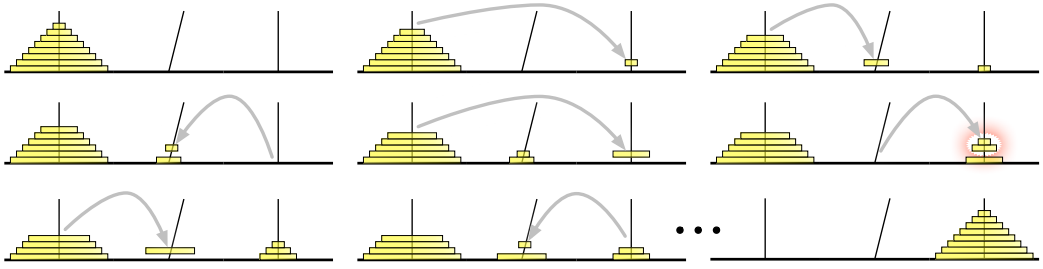
Here's the non-recursive algorithm in one line:

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 puzzle is solved.

6. 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.

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.



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

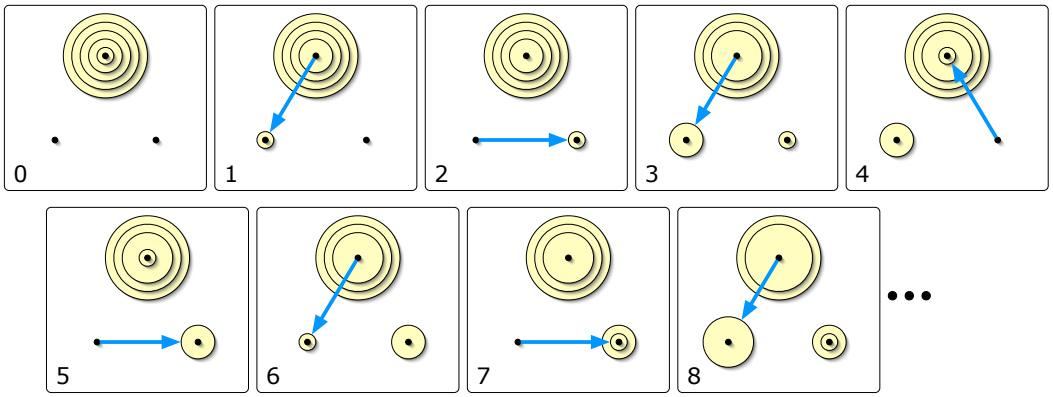
Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa then they could than could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly crowned Pope Gregory IX excommunicated Frederick II, making the local priests 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?

7. 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 5, 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?



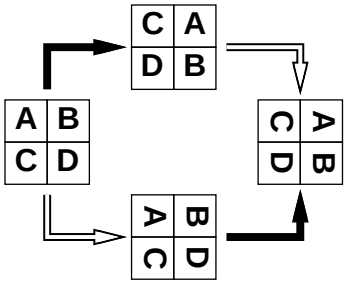
A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

★(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 variant is considerably harder to analyze than the other two.]*

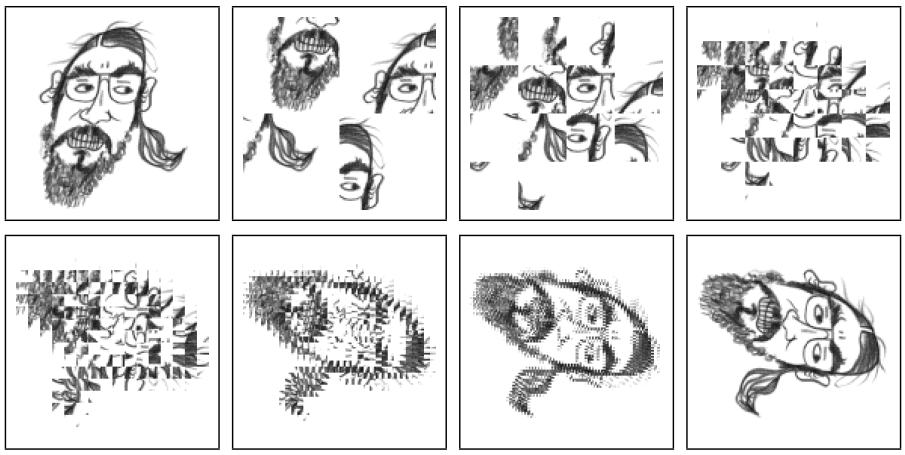
8. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function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

9. (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.]*

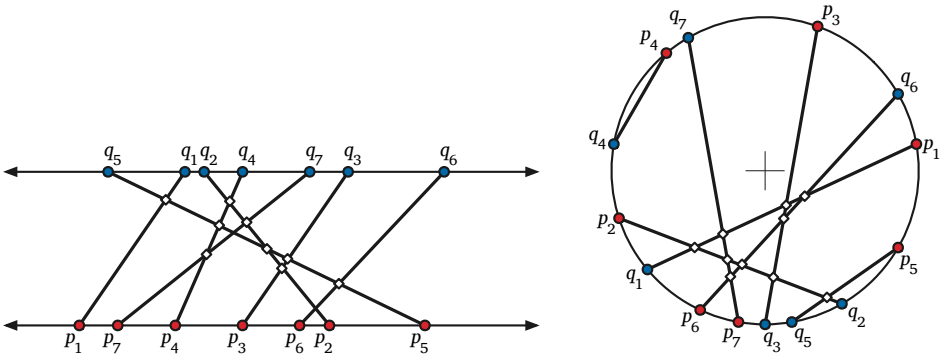
(e) Prove that the number of *swaps* executed by StoogeSort is at most $\binom{n}{2}$.

10. 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.]*

11. (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.

(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) Solve the previous problem in $O(n \log n)$ time.



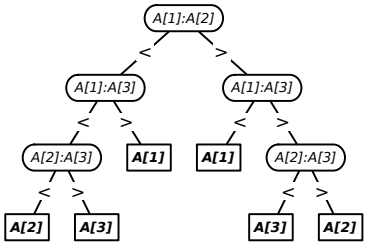
Nine intersecting pairs of segments with endpoints on parallel lines,
and ten intersecting pairs of segments with endpoints on a circle.

12. 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.)

13. 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.



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

14. Consider the following generalization of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm, 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.

```
SELECT_b(A[1..n], k):
  if n ≤ b^2
      use brute force
  else
      m ← ⌈n/b⌉
      for i ← 1 to m
          M[i] ← MEDIANOFB(A[b(i−1)+1..bi])
      mom ← SELECT(M[1..m], ⌊m/2⌋)
      r ← PARTITION(A[1..n], mom)
      if k < r
          return SELECT_b(A[1..r−1], k)
      else if k > r
          return SELECT_b(A[r+1..n], k−r)
      else
          return mom
```

(a) State a recurrence for the running time of SELECT_b, 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.

(b) What is the running time of SELECT_1? *[Hint: This is a trick question.]*

*(c) What is the running time of SELECT_2? *[Hint: This is an unfair question.]*

(d) What is the running time of SELECT_3?

(e) What is the running time of SELECT_4?

(f) For any constants $b \geq 5$, the algorithm SELECT_b 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 SELECT_b of the form $T(n) \leq \alpha_b n$ for some explicit constant α_b. (For example, on page 8 we showed that $\alpha_5 \leq 22$.)

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

15. 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:

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?]*

16. 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 either 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.

17. (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 $A \cup B \cup C$ in $O(\log n)$ time.

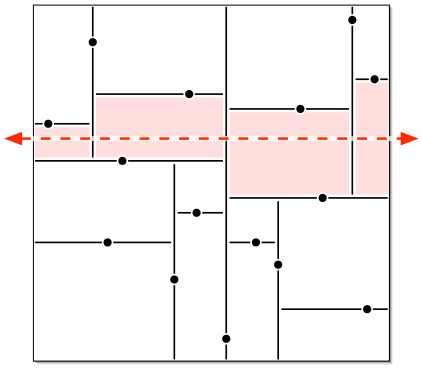
(c) Finally, suppose we are given a two dimensional array $A[1..m][1..n]$ in which every row $A[i][\,]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Use the linear-time*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

18. (a) Describe an algorithm that sorts an input 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. *[Hint: See Lecture 19.]*

    (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.)

19. 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.

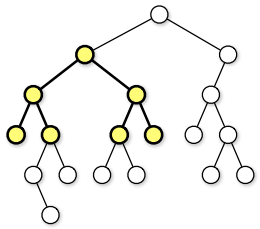
    (a) How many cells are there, as a function of n? Prove your answer is correct.

    (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. *[Hint: There is more than one function f such that $f(16) = 4$.]*

    (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).]*

20. 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.

21. 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. 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 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)?