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

— Henry David Thoreau

When you come to a fork in the road, take it.

— Yogi Berra

Ha ha! Cookies on dowels!

— Phil Ken Sebben [played by Stephen Colbert]
"Harvey Birdman, Attorney at Law"

1 Recursion

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 the previous lecture reduces the problem of apportioning Congress to the problem of maintaining a priority queue under the operations INSERT and EXTRACTMAX. In this class, we'll generally treat primitive data structures like arrays, linked lists, stacks, queues, hash tables, binary search trees, and priority queues as black boxes, adding them to the basic vocabulary of our model of computation. When we design algorithms, we may not know—and we should not care—how these basic building blocks will actually be implemented.

In some sense, *every* algorithm is simply a reduction to some underlying model of computation. Whenever you write a C program, you're really just reducing some problem to the "simpler" problems of compiling C, allocating memory, formatting output, scheduling jobs, and so forth. Even machine language programs are just reductions to the hardware-implemented problems of retrieving and storing memory and performing basic arithmetic. The underlying hardware implementation reduces those problems to timed Boolean logic; low-level hardware design reduces Boolean logic to basic physical devices such as wires and transistors; and the laws of physics reduce wires and transistors to underlying mathematical principles. At least, that's what the people who actually build hardware have to assume.[1]

A particularly powerful kind of reduction is *recursion*, which can be defined loosely as a reducing a problem to one or more **simpler instances of the *same* problem**. If the self-reference is confusing, it's useful 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 rest.[2]

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'

[1] The situation is exactly analogous to that of mathematical proof. Normally, when we prove a theorem, we implicitly rely on a several earlier results. These eventually trace back to axioms, definitions, and the rules of logic, but it is extremely rare for any proof to be expanded to pure symbol manipulation. Even when proofs are written in excruciating Bourbakian formal detail, the accuracy of the proof depends on the consistency of the formal system in which the proof is written. This consistency is simply taken for granted. In some cases (for example, first-order logic and the first-order theory of the reals) it is possible to prove consistency, but this consistency proof necessarily (thanks be to Gödel) relies on some different formal system (usually some extension of Zermelo-Fraenkel set theory), which is itself assumed to be consistent. Yes, it's turtles all the way down.

[2] I used to refer to 'elves' instead of the Recursion Fairy, referring to the traditional fairy tale in which an old shoemaker repeatedly leaves his work half-finished when he goes to bed, only to discover upon waking that elves have finished his work while he slept.

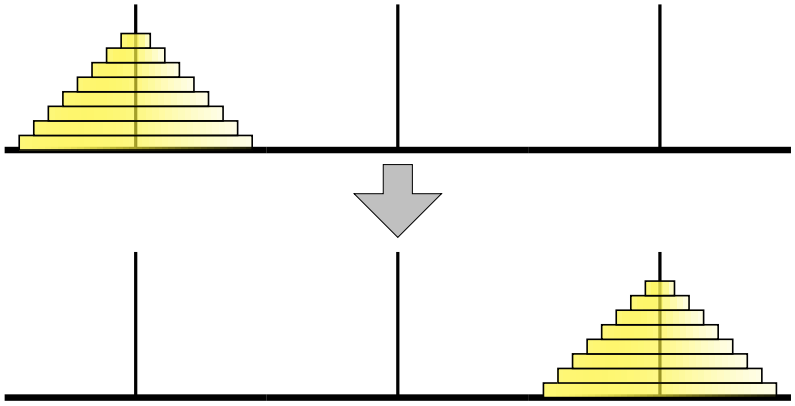
subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that is solved by some other method; otherwise, the algorithm will never terminate. This finiteness condition is *usually* easy to satisfy, but we should always be wary of 'obvious' recursive algorithms that actually recurse forever.

1.1 Tower of Hanoi

The Tower of Hanoi puzzle was first published by the French mathematician François Éduoard Anatole Lucas in 1883, under the pseudonym 'N. Claus (de Siam)' (an anagram of 'Lucas d'Amiens'). The following year, the French scientist 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.[4] 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?



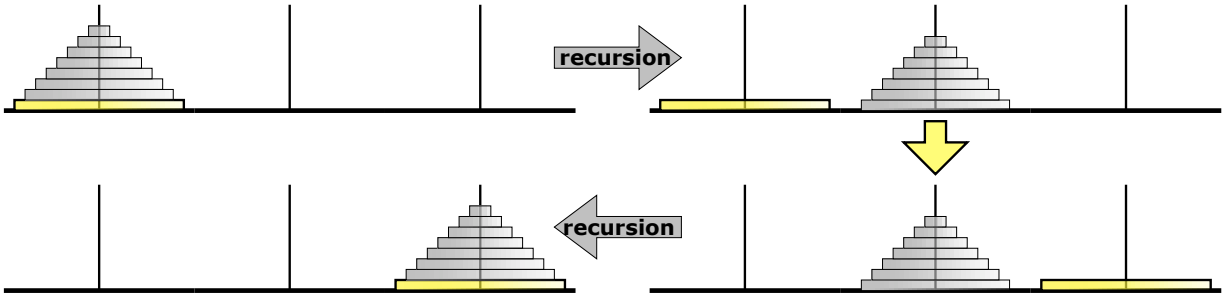
The Tower of Hanoi puzzle

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...

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

[4] Recognizing that the underlying mathematical abstraction would be unchanged, we may also freely use 'cookies' and 'dowels' instead of 'discs' and 'needles'. Ha ha... underlying!

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 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 bottom disk

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, opening up the recursion is a distraction. 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 $\boxed{T(n) = 2^n - 1}$. 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, with a thunderclap, the world will vanish.

The Hanoi algorithm has two very simple non-recursive formulations, for those of us who do not have an army of assistants to take care of smaller piles. Let's label the needles 0, 1, and 2,

and suppose the problem is to move n disks from needle 0 to needle 2 (as shown on the previous page). The non-recursive algorithm can be described with four simple rules. The proof that these rules force the same behavior as the recursive algorithm is a straightforward exercise in induction. (Hint, hint.)[5]

- 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 ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).
- Never move the same disk twice in a row.
- When there is no legal move, the puzzle is solved.

An even shorter formulation ties the algorithm more closely with its analysis. 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:

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

The Hanoi algorithm can now be described in one line:

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

On move 2^n, this rule requires us to move disk $n + 1$, which doesn't exist, so the algorithm ends. At this point, the puzzle is solved. Again, the proof that this algorithm is equivalent to our recursive formulation is a simple exercise in induction. (Hint, hint.)

1.2 Subset Sum

Let's start with a concrete example, the *subset sum* problem: Given a set X of positive integers and *target* integer T, is there a subset of element sin X that add up to T? Notice that there can be more than one such subset. For example, if $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$, the answer is TRUE, thanks to the subset $\{8, 7\}$ or $\{7, 5, 3\}$ or $\{6, 9\}$ or $\{5, 10\}$.[6] On the other hand, if $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$, the answer is FALSE.

There are two trivial cases. If the target value T is zero, then we can immediately return TRUE, since the elements of the empty set add up to zero.[7] On the other hand, if $T < 0$, or if $T \neq 0$ but the set X is empty, then we can immediately return FALSE.

For the general case, consider an arbitrary element $x \in X$. (We've already handled the case where X is empty.) There are two possibilities to consider.

- There is a subset of X that *includes* x and sums to T. Equivalently, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$.
- There is a subset of X that *excludes* x and sums to T. Equivalently, there must be a subset of $X \setminus \{x\}$ that sums to T.

[5] This means **Pay attention! This might show up on an exam! You might want to do this!**

[6] See http://www.cribbage.org/rules/ for more possibilities.

[7] The empty set is always the best base case!

So we can solve SUBSETSUM(X, T) by reducing it to two simpler instances: SUBSETSUM$(X \setminus \{x\}$, $T - x)$ and SUBSETSUM$(X \setminus \{x\}, T)$. Here's how the resulting recusive algorithm might look if X is stored in an array.

```
SUBSETSUM(X[1..n], T):
  if T = 0
      return TRUE
  else if T < 0 or n = 0
      return FALSE
  else
      return SUBSETSUM(X[2..n], T) ∨ SUBSETSUM(X[2..n], T − X[1])
```

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T. Otherwise, if there is a subset that sums to T, then either it contains $X[1]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

The running time $T(n)$ clearly satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, so the running time is $T(n) = O(2^n)$ by the annihilator method.

Along similar lines, here's a recursive algorithm for actually *constructing* a subset of X that sums to T, if one exists. This algorithm also runs in $O(2^n)$ time.

```
CONSTRUCTSUBSET(X[1..n], T):
  if T = 0
      return ∅
  if T < 0 or n = 0
      return NONE

  Y ← CONSTRUCTSUBSET(X[2..n], T)
  if Y ≠ NONE
      return Y

  Y ← CONSTRUCTSUBSET(X[2..n], T − X[1])
  if Y ≠ NONE
      return Y ∪ {X[1]}

  return NONE
```

These two algorithms are examples of a general algorithmic technique called *backtracking*. You can imagine the algorithm searching through a binary tree of recursive possibilities like a maze, trying to find a hidden treasure ($T = 0$), and backtracking whenever it reaches a dead end ($T < 0$ or $n = 0$). For *some* problems, there are tricks that allow the recursive algorithm to recognize some branches dead ends without exploring them directly, thereby speeding up the algorithm; two such problems are described later in these notes. Alas, SUBSETSUM is not one of the those problems; in the worst case, our algorithm explicitly considers *every* subset of X.

1.3 Longest Increasing Subsequence

Suppose we want to find the longest increasing subsequence of a sequence of n integers. That is, we are given an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \leq i_1 < i_2 < \cdots i_k \leq n$ such that $A[i_j] < A[i_{j+1}]$ for all j.

To derive a recursive algorithm for this problem, we start with a recursive definition of the kinds of objects we're playing with: sequences and subsequences.

> A *sequence of integers* is either empty
> or an integer followed by a sequence of integers.

This definition suggests the following strategy for devising a recursive algorithm. If the input sequence is empty, there's nothing to do. Otherwise, we should figure out what to do with the first element of the input sequence, and let the Recursion Fairy take care of everything else. We can formalize this strategy somewhat by giving a recursive definition of subsequence (using array notation to represent sequences):

> The only *subsequence* of the empty sequence is the empty sequence.
> A *subsequence* of $A[1..n]$ is either a subsequence of $A[2..n]$
> or $A[1]$ followed by a subsequence of $A[2..n]$.

We're not just looking for just *any* subsequence, but a *longest* subsequence with the property that elements are in *increasing* order. So let's try to add those two conditions to our definition. (I'll omit the familiar vacuous base case.)

> The *LIS* of $A[1..n]$ is either the LIS of $A[2..n]$
> or $A[1]$ followed by the LIS of $A[2..n]$ *with elements larger than* $A[1]$,
> *whichever is longer*.

This definition is correct, but it's not quite recursive—we're defining 'longest increasing subsequence' in terms of the *different* 'longest increasing subsequence with elements larger than x', which we haven't properly defined yet. Fortunately, this second object has a very similar recursive definition. (Again, I'm omitting the vacuous base case.)

> If $A[1] \le x$, the LIS of $A[1..n]$ with elements larger than x must be
> the LIS of $A[2..n]$ with elements larger than x.
>
> Otherwise, the LIS of $A[1..n]$ with elements larger than x is
> either the LIS of $A[2..n]$ with elements larger than x
> or $A[1]$ followed by the LIS of $A[2..n]$ with elements larger than $A[1]$,
> whichever is longer.

The longest increasing subsequence without restrictions can now be redefined as the longest increasing subsequence with elements larger than $-\infty$. Rewriting this recursive definition into pseudocode gives us the following recursive algorithm.

```
LIS(A[1..n]):
  return LISBIGGER(−∞, A[1..n])
```

```
LISBIGGER(prev, A[1..n]):
  if n = 0
      return 0
  else
      max ← LISBIGGER(prev, A[2..n])
      if A[1] > prev
          L ← 1 + LISBIGGER(A[1], A[2..n])
          if L > max
              max ← L
      return max
```

The running time of this algorithm satisfies the recurrence

$$T(n) \leq O(1) + 2T(n-1),$$

which implies that $T(n) = O(2^n)$ by the annihilator method. We really shouldn't be surprised by this running time; in the worst case, the algorithm examines each of the 2^n subsequences of the input array.

In lecture, a student suggested the following alternate strategy, which avoids defining a new object with the 'larger than x' constraint. We still only have to decide whether to include or exclude the first element $A[1]$. We consider the case where $A[1]$ is excluded exactly the same way, but to consider the case where $A[1]$ is included, we remove any elements of $A[2..n]$ that are larger than $A[1]$ *before* we recurse. This modified strategy gives us the following algorithm:

```
FILTER(A[1..n], x):
  j ← 1
  for i ← 1 to n
      if A[i] > x
          B[j] ← A[i];  j ← j + 1
  return B[1..j]
```

```
LIS(A[1..n]):
  if n = 0
      return 0
  else
      max ← LIS(prev, A[2..n])
      L ← 1 + LIS(A[1], FILTER(A[2..n], A[1]))
      if L > max
          max ← L
      return max
```

The FILTER subroutine clearly runs in $O(n)$ time, so the running time of LIS satisfies the recurrence $T(n) \leq 2T(n-1) + O(n)$, which solves to $T(n) \leq O(2^n)$ by the annihilator method. This upper bound pessimistically assumes that FILTER never actually removes any elements, but this can actually happen in the worst case, when the input sequence is sorted.

*1.4 3SAT

This section assumes you are already familiar with NP-completeness.

Now let's consider the mother of all NP-hard problems, 3SAT. Given a boolean formula in conjunctive normal form, with at most three literals in each clause, our task is to determine whether any assignment of values of the variables makes the formula true. Yes, this problem is NP-hard, which means that a polynomial algorithm is almost certainly impossible. Too bad; we have to solve the problem anyway.

The trivial solution is to try every possible assignment. We'll evaluate the running time of our 3SAT algorithms in terms of the number of variables in the formula, so let's call that n. Provided any clause appears in our input formula at most once—a condition that we can easily enforce in polynomial time—the overall input size is $O(n^3)$. There are 2^n possible assignments, and we can evaluate each assignment in $O(n^3)$ time, so the overall running time is $O(2^n n^3)$.

Since polynomial factors like n^3 are essentially noise when the overall running time is exponential, from now on I'll use $\text{poly}(n)$ to represent some arbitrary polynomial in n; in other words, $\text{poly}(n) = n^{O(1)}$. For example, the trivial algorithm for 3SAT runs in time $O(2^n \text{poly}(n))$.

We can make this algorithm smarter by exploiting the special recursive structure of 3CNF formulas:

A 3CNF formula is either nothing
or a clause with three literals $\wedge$ a 3CNF formula

Suppose we want to decide whether some 3CNF formula Φ with n variables is satisfiable. Of course this is trivial if Φ is the empty formula, so suppose

$$\Phi = (x \vee y \vee z) \wedge \Phi'$$

for some literals x, y, z and some 3CNF formula Φ'. By distributing the $\wedge$ across the $\vee$s, we can rewrite Φ as follows:

$$\Phi = (x \wedge \Phi') \vee (y \wedge \Phi') \vee (z \wedge \Phi')$$

For any boolean formula Ψ and any literal x, let $\Psi|x$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by assuming x is true. It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge \Psi|x$, which implies that

$$\Phi = (x \wedge \Phi'|x) \vee (y \wedge \Phi'|y) \vee (z \wedge \Phi'|z).$$

Thus, in any satisfying assignment for Φ, either x is true and $\Phi'|x$ is satisfiable, or y is true and $\Phi'|y$ is satisfiable, or z is true and $\Phi'|z$ is satisfiable. Each of the smaller formulas has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

$$T(n) \leq 3T(n-1) + \text{poly}(n),$$

whose solution is $O(3^n \, \text{poly}(n))$. So we've actually done *worse!*

But these three recursive cases are not mutually exclusive! If $\Phi'|x$ is *not* satisfiable, then x *must* be false in any satisfying assignment for Φ. So instead of recursively checking $\Phi'|y$ in the second step, we can check the even simpler formula $\Phi'|\bar{x}y$. Similarly, if $\Phi'|\bar{x}y$ is not satisfiable, then we know that y must be false in any satisfying assignment, so we can recursively check $\Phi'|\bar{x}\bar{y}z$ in the third step.

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  (x ∨ y ∨ z) ∧ Φ' ← Φ
  if 3SAT(Φ|x)
      return TRUE
  if 3SAT(Φ|x̄y)
      return TRUE
  return 3SAT(Φ|x̄ȳz)
```

The running time off this algorithm obeys the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \text{poly}(n),$$

where $\text{poly}(n)$ denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on. The annihilator method gives us the solution

$$T(n) = O(\lambda^n \, \text{poly}(n)) = \boxed{O(1.83928675522^n)}$$

where $\lambda \approx 1.83928675521\ldots$ is the largest root of the characteristic polynomial $r^3 - r^2 - r - 1$. (Notice that we cleverly eliminated the polynomial noise by increasing the base of the exponent ever so slightly.)

We can improve this algorithm further by eliminating *pure* literals from the formula before recursing. A literal x is *pure* in if it appears in the formula Φ but its negation $\bar{x}$ does not. It's not

hard to prove (hint, hint) that if Φ has a satisfying assignment, then it has a satisfying assignment where every pure literal is true. If $\Phi = (x \vee y \vee z) \wedge \Phi'$ has no pure literals, then some in Φ contains the literal $\bar{x}$, so we can write

$$\Phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi'$$

for some literals u and v (each of which might be y, $\bar{y}$, z, or $\bar{z}$). It follows that the first recursive formula $\Phi|x$ has contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we tested the variables y and x in the second and third cases of our previous algorithm:

$$\Phi|x = (u \vee v) \wedge \Phi'|x = (u \wedge \Phi'|xu) \vee (v \wedge \Phi'|x\bar{u}v).$$

Here is our new faster algorithm:

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  if Φ has a pure literal x
      return 3SAT(Φ|x)
  (x ∨ y ∨ z) ∧ (x̄ ∨ u ∨ v) ∧ Φ' ← Φ
  if 3SAT(Φ|xu)
      return TRUE
  if 3SAT(Φ|xūv)
      return TRUE
  if 3SAT(Φ|x̄y)
      return TRUE
  return 3SAT(Φ|x̄ȳz)
```

The running time $T(n)$ of this new algorithm satisfies the recurrence

$$T(n) = 2T(n-2) + 2T(n-3) + \text{poly}(n),$$

and the annihilator method implies that

$$T(n) = O(\mu^n \,\text{poly}(n)) = \boxed{O(1.76929235425^n)}$$

where $\mu \approx 1.76929235424\ldots$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Naturally, this approach can be extended much further. As of 2004, the fastest (deterministic) algorithm for 3SAT runs in $O(1.473^n)$ time[8], but there is absolutely no reason to believe that this is the best possible.

*1.5 Maximum Independent Set

This section assumes you are already familiar with graphs and NP-completeness.

Finally, suppose we are asked to find the largest independent set in an undirected graph G. Once again, we have an obvious, trivial algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, the algorithm might look like this.

[8]Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science* 329(1–3):303–313, 2004.

```
MAXIMUMINDSETSIZE(G):
  if G = ∅
      return 0
  else
      v ← any node in G
      withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))
      withoutv ← MAXIMUMINDSETSIZE(G \ {v})
      return max{withv, withoutv}.
```

Here, $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors. Our algorithm is exploiting the fact that if an independent set contains v, then by definition it contains none of v's neighbors. In the worst case, v has no neighbors, so $G \setminus \{v\} = G \setminus N(v)$. Thus, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n-1) + \text{poly}(n) = O(2^n \text{ poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph. If the graph is non-empty, then every maximal independent set either includes or excludes each vertex. Thus, the number of maximal independent sets satisfies the recurrence $M(n) \leq 2M(n-1)$, with base case $M(1) = 1$. The annihilator method gives us $M(n) \leq 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

We can improve this upper bound by more carefully examining the worst case of the recurrence. If v has no neighbors, then $N(v) = \{v\}$, and both recursive calls consider a graph with $n-1$ nodes. But in this case, v is in *every* maximal independent set, so one of the recursive calls is redundant. On the other hand, if v has at least one neighbor, then $G \setminus N(v)$ has at most $n-2$ nodes. So now we have the following recurrence.

$$M(n) \leq \max \left\{ \begin{array}{l} M(n-1) \\ M(n-1) + M(n-2) \end{array} \right\} = O(1.61803398875^n)$$

The upper bound is derived by solving each case separately using the annihilator method and taking the worst of the two cases. The first case gives us $M(n) = O(1)$; the second case yields our old friends the Fibonacci numbers.

We can improve this bound even more by examining the new worst case: v has exactly one neighbor w. In this case, either v or w appears in any maximal independent set. Thus, instead of recursively searching in $G \setminus \{v\}$, we should recursively search in $G \setminus N(w)$, which has at most $n-1$ nodes. On the other hand, if G has no nodes with degree 1, then $G \setminus N(v)$ has at most $n-3$ nodes.

$$M(n) \leq \max \left\{ \begin{array}{l} M(n-1) \\ 2M(n-2) \\ M(n-1) + M(n-3) \end{array} \right\} = O(1.46557123188^n)$$

The base of the exponent is the largest root of the characteristic polynomial $r^3 - r^2 - 1$. The second case implies a bound of $O(\sqrt{2}^n) = O(1.41421356237^n)$, which is smaller.

We can apply this improvement technique one more time. If G has a node v with degree 3 or more, then $G \setminus N(v)$ has at most $n-4$ nodes. Otherwise (since we have already considered nodes of degree 0 and 1), every node in the graph has degree 2. Let u, v, w be a path of three nodes in G (possibly with u adjacent to w). In any maximal independent set, either v is present and u, w are absent, or u is present and its two neighbors are absent, or w is present and its two neighbors are absent. In all three cases, we recursively count maximal independent sets in a graph with $n-3$

nodes.

$$M(n) \leq \max \left\{ \begin{array}{l} M(n-1) \\ 2M(n-2) \\ M(n-1) + M(n-4) \\ 3M(n-3) \end{array} \right\} = O(3^{n/3}) = O(1.44224957031^n)$$

The third case implies a bound of $O(1.3802775691^n)$, where the base is the largest root of $r^4 - r^3 - 1$.

Unfortunately, we cannot apply the same improvement trick again. A graph consisting of $n/3$ triangles (cycles of length three) has exactly $3^{n/3}$ maximal independent sets, so our upper bound is tight in the worst case.

Now from this recurrence, we can derive an efficient algorithm to compute the largest independent set in G in $O(3^{n/3} \operatorname{poly}(n)) = O(1.44224957032^n)$ time.

```
MAXIMUMINDSETSIZE(G):
  if G = ∅
      return 0

  else if G has a node v with degree 0
      return 1 + MAXIMUMINDSETSIZE(G \ {v})        ⟨⟨n − 1⟩⟩

  else if G has a node v with degree 1
      w ← v's neighbor
      withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))      ⟨⟨n − 2⟩⟩
      withw ← 1 + MAXIMUMINDSETSIZE(G \ N(w))      ⟨⟨≤ n − 2⟩⟩
      return max{withv, withw}

  else if G has a node v with degree greater than 2
      withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))      ⟨⟨≤ n − 4⟩⟩
      withoutv ← MAXIMUMINDSETSIZE(G \ {v})        ⟨⟨≤ n − 1⟩⟩
      return max{withv, withoutv}

  else ⟨⟨every node in G has degree 2⟩⟩
      v ← any node;  u, w ← v's neighbors
      withu ← 1 + MAXIMUMINDSETSIZE(G \ N(u))      ⟨⟨≤ n − 3⟩⟩
      withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))      ⟨⟨≤ n − 3⟩⟩
      withw ← 1 + MAXIMUMINDSETSIZE(G \ N(w))      ⟨⟨≤ n − 3⟩⟩
      return max{withu, withv, withw}
```

1.6 Generalities

Recursion is reduction from a problem to one or more simpler instances of the *same* problem. Almost every recursive algorithm (and inductive proof) closely follows a recursive definition for the object being computed. Here are a few simple recursive definitions that can be used to derive recursive algorithms:

- A natural number is either 0, or the successor of a natural number.
- A sequence is either empty, or an atom followed by a sequence.
- A sequence is either empty, an atom, or the concatenation of two shorter sequences.
- A set is either empty, or the union of a set and an atom.
- A nonempty set either is a singleton, or the union of two nonempty sets.
- A rooted tree is either nothing, or a node pointing to zero or more rooted trees.

- A binary tree is either nothing, or a node pointing to two binary trees.
- A triangulated polygon is nothing, or a triangle glued to a triangulated polygon [*not* obvious!]
- $$\sum_{i=1}^{n} a_i = \begin{cases} 0 & \text{if } n = 0 \\ \sum_{i=1}^{n-1} a_i + a_n & \text{otherwise} \end{cases}$$
- $$\sum_{i=1}^{n} a_i = \begin{cases} 0 & \text{if } n = 0 \\ a_1 & \text{if } n = 1 \\ \sum_{i=1}^{k} a_i + \sum_{i=k+1}^{n} a_i & \text{otherwise, for some } 1 \le k \le n-1 \end{cases}$$