

*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

## 2 Reductions and Recursion (September 6 and 8)

### 2.1 Reductions

*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.<sup>1</sup>

### 2.2 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  $T$ ? Suppose we have an black box that can tell us immediately whether such a subset exists. How quickly could we construct such a subset, using this black box as a subroutine? Notice that there can be more than one such subset. For example, given the inputs  $X = \{8, 6, 7, 5, 3, 10, 9\}$  and  $T = 15$ , we could return  $\{8, 7\}$  or  $\{7, 5, 3\}$  or  $\{6, 9\}$  or  $\{5, 10\}$ .<sup>2</sup>

There are two trivial cases. If the target value  $T$  is zero, then we can immediately return the empty set. On the other hand, if our black box says that no subset exists with the target sum, we can abort immediately. For the general case, consider an arbitrary element  $x \in X$ . (If  $X$  is empty

<sup>1</sup>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. It's turtles all the way down.

<sup>2</sup>See <http://www.cribbage.org/rules/> for more possibilities.

and  $T = 0$ , we've already returned the empty set. If  $X$  is empty and  $T \neq 0$ , then no subset of  $X$  sums to  $T$ , so we've already aborted. Thus,  $X$  is not empty.) We have two different ways to proceed.

- Ask the black box if any subset of  $X \setminus \{x\}$  sums to  $T$ . If the answer is no, then *every* subset of  $X$  that sums to  $T$  must *include* the element  $x$ . Otherwise, we can safely discard  $x$ .
- Ask the black box if any subset of  $X \setminus \{x\}$  sums to  $T - x$ . If the answer is no, then *every* subset of  $X$  that sums to  $T$  must *exclude* the element  $x$ . In this case, we simply discard  $x$ . Otherwise, we can safely assume that  $x$  is in our desired subset.

If we decide the  $x$  is in our output subset, we record this fact and decrease the target sum by  $x$ . Then, no matter what we decided, we remove  $x$  from the set and recurse. The resulting algorithm(s) look(s) like this:

```

CONSTRUCTSUBSET( $X[1..n], T$ ):
  if  $T = 0$ 
    return  $\emptyset$ 
  else if  $\neg$ EXISTSSUBSET( $X[1..n], T$ )
    return NULL
   $j \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $n$ 
    if (one of)  $\begin{cases} \text{EXISTSSUBSET}(X[i+1..n], T - X[i]) \\ \neg\text{EXISTSSUBSET}(X[i+1..n], T) \end{cases}$ 
       $j \leftarrow j + 1$ 
       $Y[j] \leftarrow X[i]$ 
       $T \leftarrow T - X[i]$ 
  return  $Y[1..j]$ 

```

Both versions of the algorithm make at most  $n$  calls to the EXISTSSUBSET subroutine in the worst case. The algorithm might be faster if  $T$  is small or if the elements of  $A$  are large, but that doesn't matter. Algorithms are usually—and *always* in this class—analyzed in terms of their worst-case behavior.

Now let's turn the problem around. Suppose we are given a black box CONSTRUCTSUBSET( $X, T$ ) that constructs a subset of  $X$  that sums to  $T$ . Now deciding whether such a subset *exists* is completely trivial: if CONSTRUCTSUBSET succeeds, return TRUE, else return FALSE. The existence algorithm makes only one call to the construction subroutine.

### 2.3 Recursion!

But (I'm sure you can see where this is going) if we try to put these two reductions together into a single algorithm, we end up in an infinite loop: CONSTRUCTSUBSET calls EXISTSSUBSET, which calls CONSTRUCTSUBSET, which calls EXISTSSUBSET, round and round, and nothing ever actually gets done. To break out of this infinite loop, what we need to do is make something smaller, so that eventually the cascade will always reach a trivial base case. For example, in our implementation of EXISTSSUBSET, we can remove one element of the set before calling the construction subroutine. Of course, we can't do this if the set is already empty, so we'd better handle that case directly. While we're at it, let's handle a couple of other trivial cases, too.

```

EXISTSSUBSET( $X[1..n], T$ ):
  if  $T = 0$ 
    return TRUE
  else if  $T < 0$  or  $n = 0$ 
    return FALSE
  else if CONSTRUCTSUBSET( $X[2..n], T$ )  $\neq$  NULL
    return TRUE
  else if CONSTRUCTSUBSET( $X[2..n], T - X[1]$ )  $\neq$  NULL
    return TRUE
  else
    return FALSE

```

Now we've entered into the wild and woolly world of *recursion*. Recursion is really just a special form of reduction. A recursive algorithm is really just an algorithm to solve a problem, given another algorithm to solve **simpler instances of the same problem** as a subroutine. If the self-reference is confusing, it's useful to imagine that someone else is going to solve the simpler problems. I usually refer to that 'someone else' as The Recursion Fairy.<sup>3</sup> Your only task is to simplify the original problem, or to solve it directly when simplification is impossible. The recursion fairy will take care of the rest.

Now let's analyze our recursive algorithm for the subset sum problem. Let  $E(n)$  denote the running time of the existence algorithm, and let  $C(n)$  denote the running time of the construction algorithm. Our earlier analysis gives us the following mutual recurrence.

$$E(n) \leq 2C(n-1) + O(1) \qquad C(n) \leq \sum_{i=1}^n E(i) + O(n)$$

It's not too hard to prove by induction (hint, hint) that  $C(n)$  and  $E(n)$  are both  $O(2^n)$ , but maybe we should first notice that this division into construction and existence algorithms makes everything more complicated than it needs to be. (I just did that to show off the idea of reductions.)

Here's a much simpler recursive existence algorithm, which follows the same basic pattern as our earlier algorithm, but calls itself directly instead of going through the construction subroutine.

```

EXISTSSUBSET( $X[1..n], T$ ):
  if  $T = 0$ 
    return TRUE
  else if  $T < 0$  or  $n = 0$ 
    return FALSE
  else
    return EXISTSSUBSET( $X[2..n], T$ )  $\vee$  EXISTSSUBSET( $X[2..n], T - X[1]$ )

```

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 can correctly check for each of those possibilities.

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.<sup>4</sup>

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

<sup>3</sup>I used to simply refer to 'elves'. There is a traditional fairy tale in which an old shoemaker always left his work half-finished when he went to bed, only to discover upon waking that elves had finished his work while he was asleep.

<sup>4</sup>Actually,  $T(n) = -O(1)$  is also a solution, but running times are always positive, so that doesn't make any sense.

```

CONSTRUCTSUBSET( $X[1..n], T$ ):
  if  $T = 0$ 
    return  $\emptyset$ 
  if  $T < 0$  or  $n = 0$ 
    return NONE
   $Y \leftarrow$  CONSTRUCTSUBSET( $X[2..n], T$ )
  if  $Y \neq$  NONE
    return  $Y$ 
   $Y \leftarrow$  CONSTRUCTSUBSET( $X[2..n], T - X[1]$ )
  if  $Y \neq$  NONE
    return  $Y \cup \{X[1]\}$ 
  return NONE

```

## 2.4 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  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  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 kind of object we're looking for: a sequence of integers.

A *sequence of integers* is either nothing  
or an integer followed by a sequence of integers.

This definition immediately suggests a recursive definition for a “best” sequence of integers.

The *best sequence of integers* is  
either nothing  
or the best, over all integers  $x$ , of  
 $x$  followed by the best sequence of integers that can follow  $x$ .

This is actually not quite a recursive definition; we've defined *best sequence* in terms of *best sequence that can follow  $x$* , not in terms of itself. Fortunately, it's fairly easy to extend the algorithm so that it is truly recursive.

The *best sequence of integers that can follow  $x$*  is  
either nothing  
or the best, over all integers  $y$  that can follow  $x$ , of  
 $y$  followed by the best sequence of integers that can follow  $x$ .

Or if you prefer formulas to English:

$$\text{SEQUENCE} = \left\{ \begin{array}{l} \emptyset \text{ or} \\ \text{INTEGER} \cdot \text{SEQUENCE} \end{array} \right.$$

$$\text{BESTSEQUENCEAFTER}(x) = \left\{ \begin{array}{l} \emptyset \text{ or} \\ \text{best}\{y \cdot \text{BESTSEQUENCEAFTER}(y) \mid y \text{ can follow } x\} \end{array} \right.$$

This recursive definition does not specify the set from which we can choose  $x$  and  $y$ , and it leaves the notions “best” and “can follow” undefined. These details clearly depend on the specific problem at hand. For the longest increasing subsequence problem, we can fill in these details as follows:

- The elements are taken from the input array  $A[1..n]$ .
- One sequence is better than another if it has more elements.
- An element  $A[j]$  can follow  $A[i]$  if and only if  $j > i$  and  $A[j] > A[i]$ .

Mechanically plugging these details back into our recursive definition gives us the following algorithm. (This algorithm only computes the *length* of the longest increasing subsequence, but it is fairly easy to modify the algorithm to compute the sequence itself.)

```

«Find the longest increasing subsequence of A.»
LIS(A[1..n]):
  max ← 0
  for i ← 1 to n
    L ← 1 + LISAFter(A[i], A[i + 1..n])
    if L > max
      max ← L
  return max

```

```

«Find the longest increasing subsequence of A that can follow x.»
LISAFter(x, A[1..n]):
  max ← 0
  for i ← 1 to n
    if A[i] > x
      L ← 1 + LISAFter(A[i], A[i + 1..n])
      if L > max
        max ← L
  return max

```

Now that we have the algorithm, we can simplify it in a couple of different ways. First, notice that every call to LISAFter has the form LISAFter( $A[i], A[i + 1..n]$ ). We're really just passing in the subarray  $A[i..n]$ , but with the first element separated; the algorithm becomes slightly simpler if we don't make this artificial separation. Second, the two algorithms are almost identical; we can make them *exactly* identical by adding a sentinel value  $-\infty$  to the start of the array.

```

LIS(A[1..n]):
  A[0] ←  $-\infty$ 
  return LISAFter(A[0..n])

```

```

LISAFter(A[0..n]):
  max ← 0
  for i ← 1 to n
    if A[i] > x
      L ← 1 + LISAFter(A[i..n])
      if L > max
        max ← L
  return max

```

Let  $T(n)$  denote the running time of LISAFter; the running time of LIS is clearly  $T(n) + O(1)$ . Reading directly from the algorithm, we derive the following recurrence for  $T(n)$ :

$$T(n) = \Theta(n) + \sum_{i=1}^n T(n-i) = \Theta(n) + \sum_{j=0}^{n-1} T(j)$$

This is only an upper bound. We can't predict in advance how many recursive calls LISAFter will make, so we simply assume the worst. To get a solution, we first transform it from a 'full history

recurrence' to a 'limited history recurrence' by shifting and subtracting, as follows:

$$\begin{aligned}
 T(n) &= \Theta(n) + \sum_{j=0}^{n-1} T(j) \\
 T(n-1) &= \Theta(n-1) + \sum_{j=0}^{n-2} T(j) \\
 T(n) - T(n-1) &= \Theta(1) + T(n-1) \\
 T(n) &= 2T(n-1) + \Theta(1)
 \end{aligned}$$

Once the recurrence is in this form, we can easily derive the solution  $T(n) = \Theta(2^n)$  using the annihilator method.

## 2.5 Alternately...

We could have derived almost exactly the same algorithm using a completely different recursive definition. Instead of thinking in terms of sequences, we could think in terms of subsets:

To compute the best subset of  $A$ : If  $A = \emptyset$ , return  $\emptyset$ . Otherwise, pick an element  $x \in A$ , and return either the best subset of  $A$  that includes  $x$ , or the best subset of  $A \setminus \{x\}$ , whichever is better.

We implicitly used this outline earlier, to solve the subset sum problem. For that problem, a subset is infinitely good if its elements sum to  $T$  and infinitely bad otherwise, and a "good subset of  $A$  containing  $x$ " is the union of  $\{x\}$  and a subset of  $A \setminus \{x\}$  that sums to  $T - x$ .

We can use this outline for the longest increasing subsequence problem by declaring a subset  $\{A[i_1], A[i_2], \dots, A[i_k]\} \subset A$  to be good if and only if  $i_j < i_k \iff A[i_j] < A[i_k]$ , and declaring that larger good subsets are better than smaller good subsets. This approach leads to the following recursive algorithm:

<p>SUBSETLIS(<math>A[1..n]</math>):  <math>A[0] \leftarrow -\infty</math>  return SUBSETLISWITH(<math>-\infty, A[1..n]</math>)</p>
--

<p>SUBSETLISWITH(<math>prev, A[1..n]</math>):  if <math>n = 0</math>  return 0  else  <math>max \leftarrow</math> SUBSETLISWITH(<math>prev, A[2..n]</math>)  if <math>A[1] &gt; prev</math>  <math>L \leftarrow 1 +</math> SUBSETLISWITH(<math>A[1], A[2..n]</math>)  if <math>L &gt; max</math>  <math>max \leftarrow L</math>  return <math>max</math></p>
--

For the running time, we have the recurrence

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

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

## 2.6 3SAT

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  $\Phi$  with  $n$  variables is satisfiable. Of course this is trivial if  $\Phi$  is the empty formula, so suppose

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

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

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

For any boolean formula  $\Psi$  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  $\Phi$ , 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  $\Phi$ . 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( $\Phi$ ):
  if  $\Phi = \emptyset$ 
    return TRUE
   $(x \vee y \vee z) \wedge \Phi' \leftarrow \Phi$ 
  if 3SAT( $\Phi|x$ )
    return TRUE
  if 3SAT( $\Phi|\bar{x}y$ )
    return TRUE
  return 3SAT( $\Phi|\bar{x}\bar{y}z$ )

```

The running time of 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  $\lambda \approx 1.83928675521\dots$  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  $\Phi$  but its negation  $\bar{x}$  does not. It's not hard to prove (hint, hint) that if  $\Phi$  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  $\Phi$  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( $\Phi$ ):
  if  $\Phi = \emptyset$ 
    return TRUE
  if  $\Phi$  has a pure literal  $x$ 
    return 3SAT( $\Phi|x$ )
   $(x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi' \leftarrow \Phi$ 
  if 3SAT( $\Phi|xu$ )
    return TRUE
  if 3SAT( $\Phi|x\bar{u}v$ )
    return TRUE
  if 3SAT( $\Phi|\bar{x}y$ )
    return TRUE
  return 3SAT( $\Phi|\bar{x}\bar{y}z$ )

```

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

Naturally, this approach can be extended much further. Currently the fastest (deterministic) algorithm for 3SAT runs in  $O(1.473^n)$  time<sup>5</sup>, but there is absolutely no reason to believe that this is the best possible.

## 2.7 Maximum Independent Set

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.

```

MAXIMUMINDSETSIZE( $G$ ):
  if  $G = \emptyset$ 
    return 0
  else
     $v \leftarrow$  any node in  $G$ 
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$ 
     $withoutv \leftarrow \text{MAXIMUMINDSETSIZE}(G \setminus \{v\})$ 
    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.

<sup>5</sup>Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science* 329(1-3):303-313, 2004.

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  $O(3^{n/3} \text{poly}(n)) = O(1.44224957032^n)$  time.

```

MAXIMUMINDSETSIZE(G):
  if  $G = \emptyset$ 
    return 0
  else if  $G$  has a node  $v$  with degree 0
    return  $1 + \text{MAXIMUMINDSETSIZE}(G \setminus \{v\})$      $\ll n - 1 \gg$ 
  else if  $G$  has a node  $v$  with degree 1
     $w \leftarrow v$ 's neighbor
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$      $\ll n - 2 \gg$ 
     $withw \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(w))$      $\ll \leq n - 2 \gg$ 
    return  $\max\{withv, withw\}$ 
  else if  $G$  has a node  $v$  with degree greater than 2
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$      $\ll \leq n - 4 \gg$ 
     $withoutv \leftarrow \text{MAXIMUMINDSETSIZE}(G \setminus \{v\})$      $\ll \leq n - 1 \gg$ 
    return  $\max\{withv, withoutv\}$ 
  else  $\ll \text{every node in } G \text{ has degree } 2 \gg$ 
     $v \leftarrow$  any node;  $u, w \leftarrow v$ 's neighbors
     $withu \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(u))$      $\ll \leq n - 3 \gg$ 
     $withv \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(v))$      $\ll \leq n - 3 \gg$ 
     $withw \leftarrow 1 + \text{MAXIMUMINDSETSIZE}(G \setminus N(w))$      $\ll \leq n - 3 \gg$ 
    return  $\max\{withu, withv, withw\}$ 

```

## 2.8 Generalities

Recursion is reduction from a problem to a simpler instance 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 of length  $n$  is either empty, an atom, or a sequence of length  $\lfloor n/2 \rfloor$  followed by a sequence of length  $\lceil n/2 \rceil$ .
- A set is either the empty set or the union of a set and an atom.
- A nonempty set either is a singleton or the union of two nonempty sets.
- A binary tree is either nothing or a node pointing to two binary trees.
- A full binary tree is either a node, or a node pointing to two full binary trees.
- A full binary tree is either a node, or a full binary tree with two nodes glued to one of its leaves.
- A full binary tree is either a node, or a full binary tree where one leaf has been replaced by a full binary tree.
- A triangulated polygon is nothing, or a triangle glued to a triangulated polygon [not obvious]
- A triangulated polygon is nothing, or a triangle glued to two triangulated polygons [even less obvious]
- A non-empty triangulated polygon is a triangle, or two non-empty triangulated polygons glued together along an edge [even less obvious]
- A string of balanced parentheses is either nothing, or two strings of balanced parentheses, or a string of balanced parentheses inside a pair of parentheses [  $S = \varepsilon | SS | (S)$  ]

$$\bullet \sum_{i=1}^n a_i = \begin{cases} 0 & \text{if } n = 0 \\ \sum_{i=1}^{n-1} a_i + a_n & \text{otherwise} \end{cases}$$

$$\bullet \sum_{i=1}^n a_i = \begin{cases} 0 & \text{if } n = 0 \\ a_1 & \text{if } n = 1 \\ \sum_{i=1}^{\lfloor n/2 \rfloor} a_i + \sum_{i=\lfloor n/2 \rfloor + 1}^n a_i & \text{otherwise} \end{cases}$$