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

— Yogi Berra

*Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign." have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and And, and And and and, and and and And, and And and and, and and and Chips, as well as after Chips?*[1]

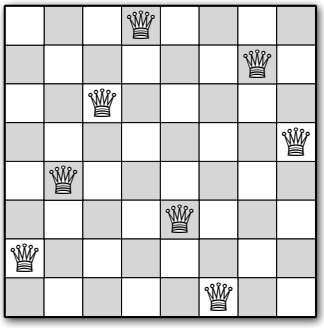
— unknown, from the original FreeBSD 'fortune' file

# 3 Backtracking

In this lecture, I want to describe another recursive algorithm strategy called *backtracking*. A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between two alternatives to the next component of the solution, it simply tries both options recursively.

## 3.1 $n$-Queens

The prototypical backtracking problem is the classical ***n-Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 for the standard $8 \times 8$ board, and both solved and generalized to larger boards by Franz Nauck in 1850. The problem is to place $n$ queens on an $n \times n$ chessboard, so that no two queens can attack each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, column, or diagonal.



One solution to the 8 queens problem, represented by the array [4,7,3,8,2,5,1,6]

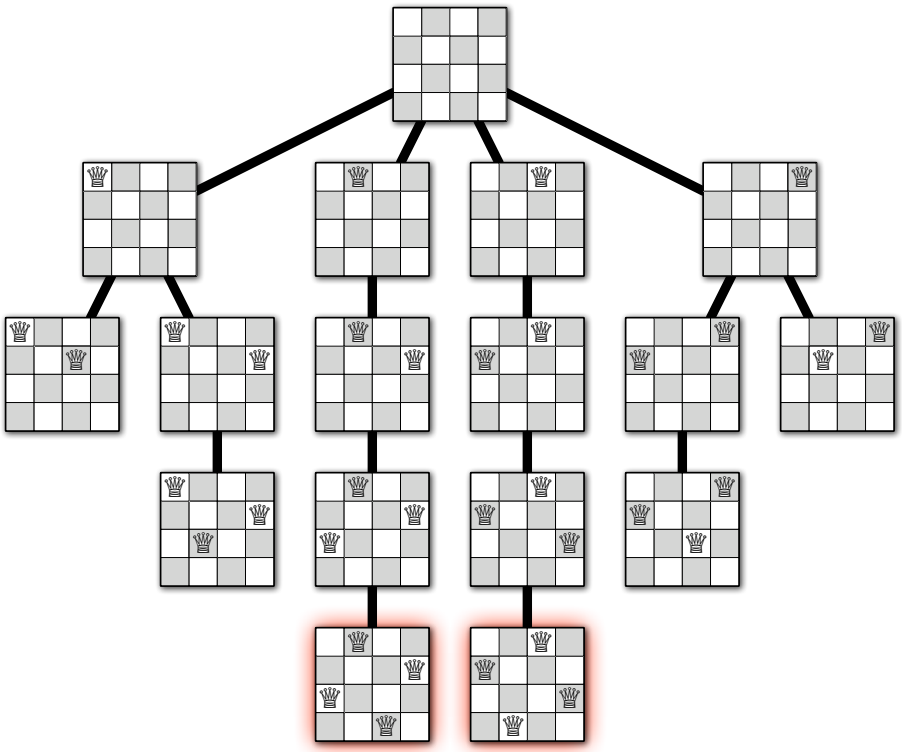
Obviously, in any solution to the $n$-Queens problem, there is exactly one queen in each column. So we will represent our possible solutions using an array $Q[1..n]$, where $Q[i]$ indicates the square in column $i$ that contains a queen, or 0 if no queen has yet been placed in column $i$. To find a solution, we will put queens on the board row by row, starting at the top. A *partial* solution is an array $Q[1..n]$ whose first $r-1$ entries are positive and whose last $n-r+1$ entries are all zeros, for some integer $r$.

The following recursive algorithm recursively enumerates all complete $n$-queens solutions that are consistent with a given partial solution. The input parameter $r$ is the first empty row. Thus, to compute all $n$-queens solutions with no restrictions, we would call RECURSIVENQUEENS$(Q[1..n], 1)$.

[1] If you ever decide to read this sentence out loud, be sure to pause briefly between 'Fish and and' and 'and and and And', 'and and and And' and 'and And and and', 'and And and and' and 'and and and And', 'and and and And' and 'and And and and', and 'and And and and' and 'and and and Chips'!

```
RecursiveNQueens(Q[1..n], r):
  if r = n + 1
      print Q
  else
      for j ← 1 to n
          legal ← True
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = r − j + i) or (Q[i] = r + i − j)
                  legal ← False
          if legal
              Q[r] ← j
              RecursiveNQueens(Q[1..n], r + 1)
```

Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a *recursion tree*. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the $n$-Queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen.



The complete recursion tree for our algorithm for the 4-queens problem.

## 3.2 Subset Sum

Let's consider a more complicated problem, called SubsetSum: Given a set $X$ of positive integers and *target* integer $T$, is there a subset of elements in $X$ 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 subsets $\{8,7\}$ or $\{7,5,3\}$ or $\{6,9\}$ or $\{5,10\}$. 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, because the elements of the empty set add up to zero.[2] 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 is a subset of $X$ that sums to $T$ if and only if one of the following statements is true:

- There is a subset of $X$ that *includes* $x$ and whose sum is $T$.
- There is a subset of $X$ that *excludes* $x$ and whose sum is $T$.

In the first case, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$; in the second case, there must be a subset of $X \setminus \{x\}$ that sums to $T$. 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 $T$, so TRUE is the correct output. Otherwise, if $T$ is negative or the set $X$ is empty, then no subset of $X$ sums to $T$, so FALSE is the correct output. 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.

Here is a similar recursive algorithm that actually *constructs* 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

[2] There's no base case like the vacuous base case!

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 as 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 $X$.[3]

### 3.3 Longest Increasing Subsequence

Now suppose we are given a sequence of integers, and we want to find the longest subsequence whose elements are in increasing order. More concretely, the input is an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \le i_1 < i_2 < \cdots i_k \le n$ 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 $x$ is
> 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.

[3]Indeed, because SUBSETSUM is NP-hard, there is almost certainly no way to solve it in subexponential time.

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 $T(n) \leq 2T(n-1) + O(1)$, so as usual the annihilator method implies that $T(n) = O(2^n)$. 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.

The following alternative strategy 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 new 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; indeed, if the input sequence is sorted in increasing order, this assumption is correct.

## 3.4 Optimal Binary Search Trees

Our next example combines recursive backtracking with the divide-and-conquer strategy.

Hopefully you remember that the cost of a successful search in a binary search tree is proportional to the number of ancestors of the target node.[4] As a result, the worst-case search time is proportional to the depth of the tree. To minimize the worst-case search time, the height of the tree should be as small as possible; ideally, the tree is perfectly balanced.

In many applications of binary search trees, it is more important to minimize the total cost of several searches than to minimize the worst-case cost of a single search. If $x$ is a more 'popular' search target

[4] An *ancestor* of a node $v$ is either the node itself or an ancestor of the parent of $v$. A *proper* ancestor of $v$ is either the parent of $v$ or a proper ancestor of the parent of $v$.

than $y$, we can save time by building a tree where the depth of $x$ is smaller than the depth of $y$, even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of $n$ keys $A[1..n]$ and an array of corresponding *access frequencies* $f[1..n]$. Over the lifetime of the search tree, we will search for the key $A[i]$ exactly $f[i]$ times. Our task is to build the binary search tree that minimizes the *total* search time.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree $T$ with $n$ nodes. Let $v_i$ denote the node that stores $A[i]$, and let $r$ be the index of the root node. Ignoring constant factors, the cost of searching for $A[i]$ is the number of nodes on the path from the root $v_r$ to $v_i$. Thus, the total cost of performing all the binary searches is given by the following expression:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \cdot \#\text{nodes between } v_r \text{ and } v_i$$

Every search path includes the root node $v_r$. If $i < r$, then all other nodes on the search path to $v_i$ are in the left subtree; similarly, if $i > r$, all other nodes on the search path to $v_i$ are in the right subtree. Thus, we can partition the cost function into three parts as follows:

$$\begin{aligned} Cost(T, f[1..n]) = &\sum_{i=1}^{r-1} f[i] \cdot \#\text{nodes between } \mathit{left}(v_r) \text{ and } v_i \\ &+ \sum_{i=1}^{n} f[i] \\ &+ \sum_{i=r+1}^{n} f[i] \cdot \#\text{nodes between } \mathit{right}(v_r) \text{ and } v_i \end{aligned}$$

Now the first and third summations look exactly like our original expression (*) for $Cost(T, f[1..n])$. Simple substitution gives us our recursive definition for *Cost*:

$$\boxed{Cost(T, f[1..n]) = Cost(\mathit{left}(T), f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; Cost(\mathit{right}(T), f[r+1..n])}$$

The base case for this recurrence is, as usual, $n = 0$; the cost of performing no searches in the empty tree is zero.

Now our task is to compute the tree $T_{\text{opt}}$ that minimizes this cost function. Suppose we somehow magically knew that the root of $T_{\text{opt}}$ is $v_r$. Then the recursive definition of $Cost(T, f)$ immediately implies that the left subtree $\text{left}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $\text{right}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$. **Once we choose the correct key to store at the root, the Recursion Fairy will automatically construct the rest of the optimal tree for us.** More formally, let $OptCost(f[1..n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

$$\boxed{OptCost(f[1..n]) = \min_{1 \le r \le n} \left\{ OptCost(f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; OptCost(f[r+1..n]) \right\}}$$

Again, the base case is $\mathit{OptCost}(f[1..0]) = 0$; the best way to organize no keys, which we will plan to search zero times, is by storing them in the empty tree!

This recursive definition can be translated mechanically into a recursive algorithm, whose running time $T(n)$ satisfies the recurrence

$$T(n) = \Theta(n) + \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big).$$

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$. The recurrence looks harder than it really is. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$\begin{aligned} T(n) &= \Theta(n) + 2\sum_{k=0}^{n-1} T(k) \\ T(n-1) &= \Theta(n-1) + 2\sum_{k=0}^{n-2} T(k) \\ T(n) - T(n-1) &= \Theta(1) + 2T(n-1) \\ T(n) &= 3T(n-1) + \Theta(1) \end{aligned}$$

The solution $\boxed{T(n) = \Theta(3^n)}$ now follows from the annihilator method.

It's worth emphasizing here that our recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with $n$ nodes satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1} N(r-1) \cdot N(n-r), \quad N(0) = 1,$$

which has the closed-from solution $N(n) = \Theta(4^n/\sqrt{n})$. Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees. A full enumeration of binary search trees would consider all possible *pairings* of left and right subtrees; hence the product in the recurrence for $N(n)$.

The next two sections assume familiarity with NP-completeness, which I usually cover *first* in the graduate algorithms class. Please refer to the NP-completeness notes for self-contained definitions and examples of 3SAT and Maximum Independent Set. For this reason, I don't cover this material in the undergraduate class.

## 3.5 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(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 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

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

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 $\Phi$ 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(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

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

and the annihilator method implies that

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

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[5], but there is absolutely no reason to believe that this is the best possible.

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

### 3.6 Maximum Independent Set

Finally, suppose we are given an undirected graph $G$ and are asked to find the size of the *largest independent set*, that is, the largest subset of the vertices of $G$ with no edges between them. Once again, we have an obvious recursive 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 = ∅
      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; an independent set is maximal if every vertex in $G$ is either already in the set or a neighbor of a vertex in the set. 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}
```

## Exercises

1. (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of $A$ and $B$ is both a subsequence of $A$ and a subsequence of $B$. Give a simple recursive definition for the function $lcs(A,B)$, which gives the length of the *longest* common subsequence of $A$ and $B$.

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of $A$ and $B$ is another sequence that contains both $A$ and $B$ as subsequences. Give a simple recursive definition for the function $scs(A,B)$, which gives the length of the *shortest* common supersequence of $A$ and $B$.

   (c) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even $i$, and $X[i] > X[i+1]$ for all odd $i$. Give a simple recursive definition for the function $los(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array $A$ of integers.

   (d) Give a simple recursive definition for the function $sos(A)$, which gives the length of the shortest oscillating supersequence of an arbitrary array $A$ of integers.

   (e) Call a sequence $X[1..n]$ *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all $i$. Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest accelerating subsequence of an arbitrary array $A$ of integers.

*2. Describe an algorithm to solve 3SAT in time $O(\phi^n \text{poly}(n))$, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*