Where, however, the ambiguity cannot be cleared up, either by the rule of faith or by the context, there is nothing to hinder us to point the sentence according to any method we choose of those that suggest themselves.

— Augustine of Hippo, *De doctrina Christiana* (397CE)
Translated by Marcus Dods (1892)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin,
Scientific American (1886)

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— attributed to Bertrand Russell

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

— Yogi Berra (giving directions to his house)

2

Backtracking

This chapter describes another important recursive strategy called ***backtracking***. A backtracking algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates ***every*** alternative and then chooses the best one.

2.1 N Queens

The prototypical backtracking problem is the classical ***n Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym "Schachfreund") for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens are attacking each other.

For readers not familiar with the rules of chess, this means that no two queens are in the same row, the same column, or the same diagonal.

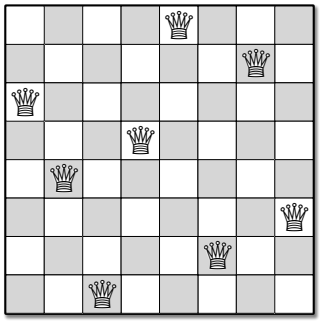


Figure 2.1. Gauss's first solution to the 8 queens problem, represented by the array $[5, 7, 1, 4, 2, 8, 6, 3]$

In a letter written to his friend Heinrich Schumacher in 1850, the eminent mathematician Carl Friedrich Gauss wrote that one could easily confirm Franz Nauck's claim that the Eight Queens problem has 92 solutions by trial and error in a few hours. ("*Schwer ist es übrigens nicht, durch ein methodisches Tatonniren sich diese Gewissheit zu verschaffen, wenn man 1 oder ein paar Stunden daran wenden will.*") His description *Tatonniren* comes from the French *tâtonner*, meaning to feel, grope, or fumble around blindly, as if in the dark.

Gauss's letter described the following recursive strategy for solving the n-queens problem; the same strategy was described in 1882 by the French recreational mathematician Édouard Lucas, who attributed the method to Emmanuel Laquière. We place queens on the board one row at a time, starting with the top row. To place the rth queen, we methodically try all n squares in row r from left to right in a simple for loop. If a particular square is attacked by an earlier queen, we ignore that square; otherwise, we tentatively place a queen on that square and *recursively* grope for consistent placements of the queens in later rows.

Figure 2.2 shows the resulting algorithm, which recursively enumerates *all* complete n-queens solutions that are consistent with a given partial solution. Following Gauss, we represent the positions of the queens using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen. When PLACEQUEENS is called, the input parameter r is the index of the first empty row, and the prefix $Q[1..r-1]$ contains the positions of the first $r-1$ queens. In particular, to compute all n-queens solutions with no restrictions, we would call PLACEQUEENS$(Q[1..n], 1)$. The outer for-loop considers all possible placements of a queen on row r; the inner for-loop checks whether a candidate placement of row r is consistent with the queens that are already on the first $r-1$ rows.

The execution of PLACEQUEENS can be illustrated using a ***recursion tree***. Each node in this tree corresponds to a recursive subproblem, and thus to a legal partial solution; in particular, the root corresponds to the empty board

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

Figure 2.2. Gauss and Laquière's backtracking algorithm for the n queens problem.

(with $r = 0$). Edges in the recursion tree correspond to recursive calls. Leaves correspond to partial solutions that cannot be further extended, either because there is already a queen on every row, or because every position in the next empty row is attacked by an existing queen. The backtracking search for complete solutions is equivalent to a depth-first search of this tree.

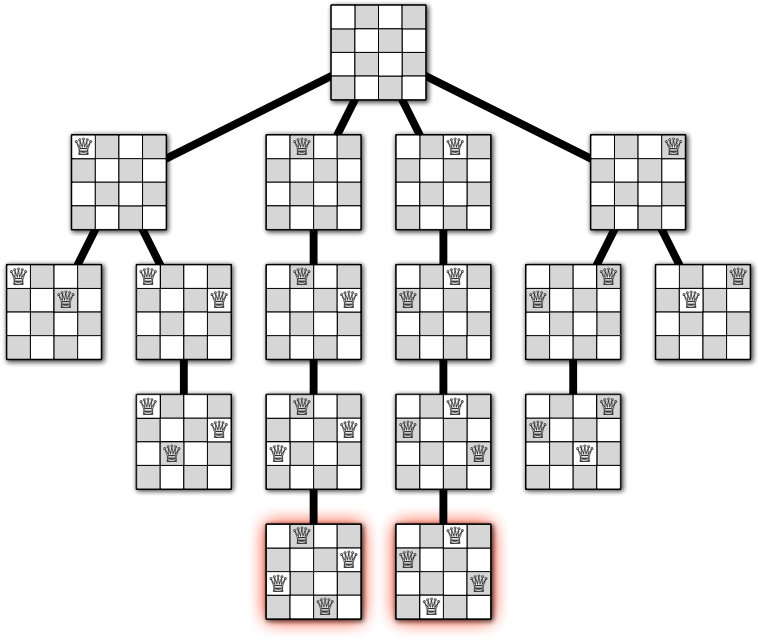


Figure 2.3. The complete recursion tree of Gauss and Laquière's algorithm for the 4 queens problem.

2.2 Game Trees

Consider the following simple two-player game[1] played on an $n \times n$ square grid with a border of squares; let's call the players Horace Fahlberg-Remsen and Vera Rebaudi.[2] Each player has n tokens that they move across the board from one side to the other. Horace's tokens start in the left border, one in each row, and move ***hor***izontally to the right; symmetrically, Vera's tokens start in the top border, one in each column, and move ***ver***tically downward. The players alternate turns. In each of his turns, Horace either *moves* one of his tokens one step to the right into an empty square, or *jumps* one of his tokens over exactly one of Vera's tokens into an empty square two steps to the right. If no legal moves or jumps are available, Horace simply passes. Similarly, Vera either moves or jumps one of her tokens downward in each of her turns, unless no moves or jumps are possible. The first player to move all their tokens off the edge of the board wins. (It's not hard to prove that as long as there are tokens on the board, at least one player can legally move, and therefore someone eventually wins.)

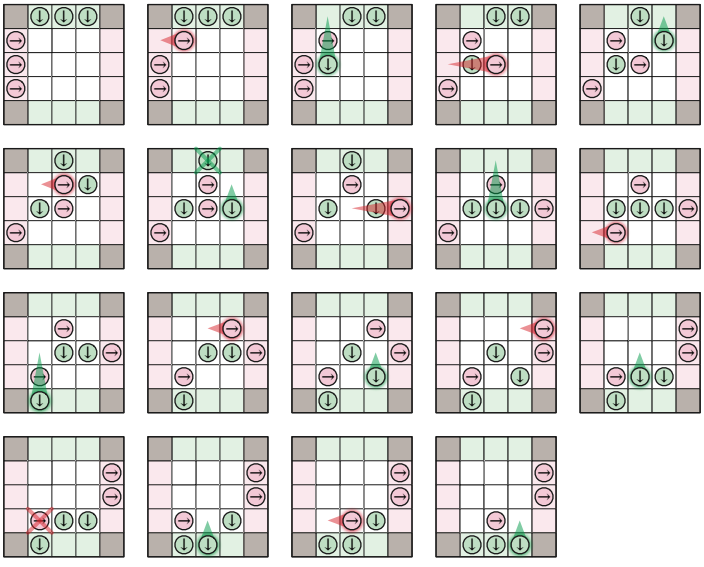


Figure 2.4. Vera wins the 3 × 3 fake-sugar-packet game.

[1] I don't know what this game is called, or even if I'm remembering the rules correctly; I learned it (or something like it) from Lenny Pitt, who recommended playing it with fake-sugar packets at restaurants.

[2] Constantin Fahlberg and Ira Remsen synthesized saccharin for the first time in 1878, while Fahlberg was a postdoc in Remsen's lab investigating coal tar derivatives. In 1900, Ovidio Rebaudi published the first chemical analysis of *ka'a he'ê*, a medicinal plant cultivated by the Guaraní for more than 1500 years, now more commonly known as *Stevia rebaudiana*.

Unless you've seen this game before[3], you probably don't have any idea how to play it well. Nevertheless, there is a relatively simple backtracking algorithm that can play this game—or any two-player game without randomness or hidden information that ends after a finite number of moves—*perfectly*. That is, if we drop you into the middle of a game, and it is *possible* to win against another perfect player, the algorithm will tell you how to win.

A ***state*** of the game consists of the locations of all the pieces and the identity of the current player. These states can be connected into a *game tree*, which has an edge from state x to state y if and only if the current player in state x can legally move to state y. The root of the game tree is the initial position of the game, and every path from the root to a leaf is a complete game.

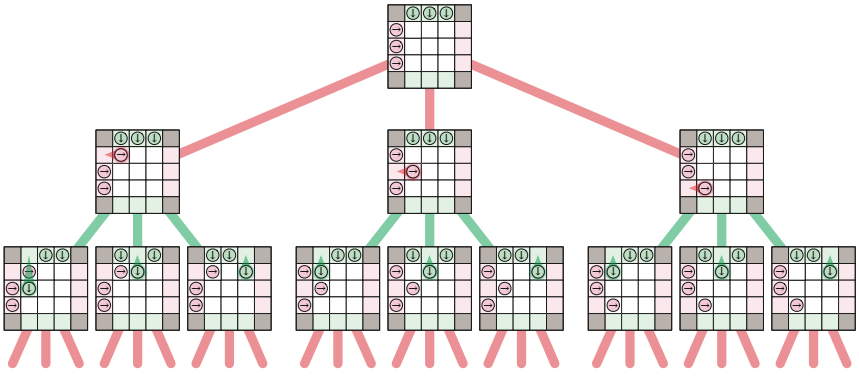


Figure 2.5. The first two levels of the fake-sugar-packet game tree.

To navigate through this game tree, we recursively define a game state to be ***good*** or ***bad*** as follows:

- A game state is *good* if either the current player has already won, or if the current player can move to a bad state for the opposing player.
- A game state is *bad* if either the current player has already lost, or if every available move leads to a good state for the opposing player.

Equivalently, a non-leaf node in the game tree is good if it has at least one bad child, and a non-leaf node is bad if all its children are good. By induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake. This recursive definition was proposed by Ernst Zermelo in 1913.[4]

This recursive definition immediately suggests the following recursive backtracking algorithm to determine whether a given game state is good or bad. At

[3] If you have, please tell me where!

[4] In fact, Zermelo considered the more subtle class of games that have a finite number of states, but that allow infinite sequences of moves. (Zermelo defined infinite play to be a draw.)

its core, this algorithm is just a depth-first search of the game tree; equivalently, the game tree is the recursion tree of the algorithm! A simple modification of this backtracking algorithm finds a good move (or even all possible good moves) if the input is a good game state.

```
PLAYANYGAME(X, player):
  if player has already won in state X
      return GOOD
  if player has already lost in state X
      return BAD
  for all legal moves X ⇝ Y
      if PLAYANYGAME(Y, ¬player) = BAD
          return GOOD        ⟨⟨X ⇝ Y is a good move⟩⟩
  return BAD                 ⟨⟨There are no good moves⟩⟩
```

All game-playing programs are ultimately based on this simple backtracking strategy. However, since most games have an enormous number of states, it is not possible to traverse the entire game tree in practice. Instead, game programs employ other heuristics[5] to *prune* the game tree, by ignoring states that are obviously (or "obviously") good or bad, or at least better or worse than other states, and/or by cutting off the tree at a certain depth (or *ply*) and using a more efficient heuristic to evaluate the leaves.

2.3 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, because the subsets $\{8, 7\}$ and $\{7, 5, 3\}$ and $\{6, 9\}$ and $\{5, 10\}$ all sum to 15. 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 empty set is a subset of *every* set X, and the elements of the empty set add up to zero.[6] 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.

[5] A heuristic is an algorithm that doesn't work. (Except in practice. Sometimes. Maybe.)

[6] . . . because what else could they add up to?

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$(X \setminus \{x\}, T)$. The resulting recursive algorithm is shown below.

```
⟨⟨Does any subset of X sum to T?⟩⟩
SUBSETSUM(X, T):
  if T = 0
      return TRUE
  else if T < 0 or X = ∅
      return FALSE
  else
      x ← any element of X
      with  ← SUBSETSUM(X \ {x}, T − x)    ⟨⟨Recurse!⟩⟩
      wout ← SUBSETSUM(X \ {x}, T)         ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

Correctness

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 $X[n]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

Analysis

In order to analyze the algorithm, we have to describe a few implementation details more precisely. To begin, let's assume that the input set X is given as an array $X[1..n]$.

The previous recursive algorithm allows us to choose *any* element $x \in X$ in the main recursive case. Purely for the sake of efficiency, it is helpful to choose an element x such that the remaining subset $X \setminus \{x\}$ has a concise description, which can be computed quickly, so that setting up the recursive calls requires minimal overhead. Specifically, we will let x be the last element $X[n]$; then the subset $X \setminus \{x\}$ is stored in the prefix $X[1..n-1]$. Passing a complete *copy* of this prefix to the recursive calls would take too long—we need $\Theta(n)$ time just to make the copy—so instead, we push only two values: a reference to the array (or its starting address) and the length of the prefix. (Alternatively, we could avoid passing a reference to X to *every* recursive call by making X a global variable.)

```
⟨⟨Does any subset of X[1..i] sum to T?⟩⟩
SUBSETSUM(X, i, T):
  if T = 0
      return TRUE
  else if T < 0 or i = 0
      return FALSE
  else
      with ← SUBSETSUM(X, i − 1, T − X[i])    ⟨⟨Recurse!⟩⟩
      wout ← SUBSETSUM(X, i − 1, T)           ⟨⟨Recurse!⟩⟩
      return (with ∨ wout)
```

With these implementation choices, the running time $T(n)$ of our algorithm satisfies the recurrence $T(n) \le 2T(n-1) + O(1)$. The solution $T(n) = O(2^n)$ follows easily using either recursion trees or the even simpler "Oh yeah, we already solved this recurrence for the Tower of Hanoi" method. In the worst case—for example, when T is larger than the sum of all elements of X—the recursion tree for this algorithm is a complete binary tree with depth n, and the algorithm considers all 2^n subsets of X.

Variants

With only minor changes, we can solve several variants of SUBSETSUM. For example, Figure 2.6 shows an algorithm that actually *constructs* a subset of X that sums to T, if one exists, or returns the error value NONE if no such subset exists; this algorithm uses exactly the same recursive strategy as our earlier decision algorithms. This algorithm also runs in $O(2^n)$ time; the analysis is simplest if we assume a set data structure that allows us to insert a single element in $O(1)$ time (for example, a linked list), but in fact the running time is still $O(2^n)$ even if insertion requires $O(n)$ time (for example, a *sorted* linked list). Similar variants allow us to count subsets that sum to a particular value, or choose the *best* subset (according to some other criterion) that sums to a particular value.

Most other problems that are solved by backtracking have this property: the same recursive strategy can be used to solve many different variants of the same problem. For example, it is easy to modify the recursive strategy described in the previous section, which determines whether a given game position is good or bad, to instead return a good move, or a list of all good moves. For this reason, when we *design* backtracking algorithms, we should aim for the simplest possible variant of the problem, computing a number or even a single boolean instead of more complex information or structure.

```
⟨⟨Return a subset of X[1..i] that sums to T⟩⟩
⟨⟨or NONE if no such subset exists⟩⟩
ConstructSubset(X, i, T):
  if T = 0
      return ∅
  if T < 0 or n = 0
      return NONE
  Y ← ConstructSubset(X, i − 1, T)
  if Y ≠ NONE
      return Y
  Y ← ConstructSubset(X, i − 1, T − X[i])
  if Y ≠ NONE
      return Y ∪ {X[i]}
  return NONE
```

Figure 2.6. A recursive backtracking algorithm for the construction version of SUBSETSUM.

2.4 The General Pattern

Backtracking algorithms are commonly used to make a *sequence of decisions*, with the goal of building a recursively defined structure satisfying certain constraints. Often (but not always) this goal structure is itself a sequence. For example:

- In the n-queens problem, the goal is a sequence of queen positions, one in each row, such that no two queens attack each other. For each row, the algorithm *decides* where to place the queen.
- In the game tree problem, the goal is a sequence of legal moves, such that each move is as good as possible for the player making it. For each game state, the algorithm *decides* the best possible next move.
- In the SUBSETSUM problem, the goal is a sequence of input elements that have a particular sum. For each input element, the algorithm *decides* whether to include it in the output sequence or not.

(Hang on, why is the goal of *subset* sum finding a *sequence*? That was a deliberate design decision. We imposed a convenient ordering on the input set—by representing it using an array, as opposed to some other more amorphous data structure—that we can exploit in our recursive algorithm.)

In each recursive call to the backtracking algorithm, we need to make ***exactly one*** decision, and our choice must be consistent with all previous decisions. Thus, each recursive call requires not only the portion of the input data we have not yet processed, but also a suitable summary of the decisions we have already made. For the sake of efficiency, the summary of past decisions should be as small as possible. For example:

- For the n-queens problem, we must pass in not only the number of empty rows, but the positions of all previously placed queens. Here, unfortunately, we must remember our past decisions in complete detail.
- For the game tree problem, we only need to pass in the current state of the game, including the identity of the next player. We don't need to remember *anything* about our past decisions, because who wins from a given game state does not depend on the moves that created that state.[7]
- For the SUBSETSUM problem, we need to pass in both the remaining available integers and the remaining target value, which is the original target value minus the *sum* of the previously chosen elements. Precisely which elements were previously chosen is unimportant.

When we design new recursive backtracking algorithms, we must figure out *in advance* what information we will need about past decisions *in the middle of the algorithm*. If this information is nontrivial, our recursive algorithm might need to solve a more general problem than the one we were originally asked to solve. (We've seen this kind of generalization before: To find the *median* of an unsorted array in linear time, we derived an algorithm to select the kth smallest element for *arbitrary* k.)

Finally, once we've figured out what recursive problem we *really* need to solve, we solve that problem by ***recursive brute force***: Try *all* possibilities for the next decision that are consistent with past decisions, and let the Recursion Fairy worry about the rest. No being clever here. No skipping "obviously" stupid choices. Try everything. You can make the algorithm faster later.

2.5 Text Segmentation (*Interpunctio Verborum*)

Suppose you are given a string of letters representing text in some foreign language, but without any spaces or punctuation, and you want to break this string into its individual constituent words. For example, you might be given the following passage from Cicero's famous oration in defense of Lucius Licinius Murena in 62BCE, in the standard *scriptio continua* of classical Latin:[8]

[7] Many games *appear* to violate this independence condition. For example, the standard rules of both chess and checkers allow a player to declare a draw if the same arrangement of pieces occurs three times, and the Chinese rules for go simply forbid repeating any earlier arrangement of stones. Thus, for these games, a game state formally includes not only the current positions of the pieces but the entire history of previous moves.

[8] In·fact·most·classical·Latin·manuscripts·separated·words·with·small·dots·called·*interpuncts*. Interpunctuation all but disappeared by the 3rd century in favor of *scriptio continua*. Empty spaces between words were introduced by Irish monks in the 8th century and slowly spread across Europe over the next several centuries. *Scriptio continua* survives in early 21st-century English in the form of URLs and hashtags. #octotherps4lyfe

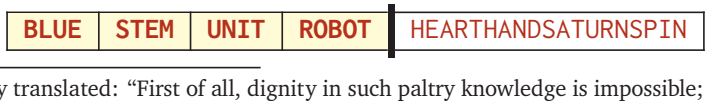
PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
ATQVEINTERPVNCTIONIBUSVERBORVMOCCVPATAE

A fluent Latin reader would parse this string (in modern orthography) as *Primus dignitas in tam tenui scientia non potest esse; res enim sunt parvae, prope in singulis litteris atque interpunctionibus verborum occupatae*.[9] Text segmentation is not only a problem in classical Latin and Greek, but in several modern languages and scripts including Balinese, Burmese, Chinese, Japanese, Javanese, Khmer, Lao, Thai, Tibetan, and Vietnamese. Similar problems arise in segmenting unpunctuated English text into sentences,[10] segmenting text into lines for typesetting, speech and handwriting recognition, curve simplification, and several types of time-series analysis. For purposes of illustration, I'll stick to segmenting sequences of letters in the modern English alphabet into modern English words.

Of course, some strings can be segmented in several different ways; for example, BOTHEARTHANDSATURNSPIN can be decomposed into English words as either BOTH·EARTH·AND·SATURN·SPIN or BOT·HEART·HANDS·AT·URNS·PIN, among many other possibilities. For now, let's consider an extremely simple segmentation problem: Given a string of characters, can it be segmented into English words *at all*?

To make the problem concrete (and language-agnostic), let's assume we have access to a subroutine IsWord(w) that takes a string w as input, and returns True if w is a "word", or False if w is not a "word". For example, if we are trying to decompose the input string into palindromes, then a "word" is a synonym for "palindrome", and therefore IsWord(ROTATOR) = True but IsWord(PALINDROME) = False.

Just like the SubsetSum problem, the *input* structure is a sequence, this time containing letters instead of numbers, so it is natural to consider a decision process that consumes the input characters in order from left to right. Similarly, the *output* structure is a sequence of words, so it is natural to consider a process that produces the output words in order from left to right. Thus, jumping into the middle of the segmentation process, we might imagine the following picture:



[9] Loosely translated: "First of all, dignity in such paltry knowledge is impossible; this is trivial stuff, mostly concerned with individual letters and the placement of points between words." Cicero was openly mocking the legal expertise of his friend(!) and noted jurist Servius Sulpicius Rufus, who had accused Murena of bribery, after Murena defeated Rufus in election for consul. Murena was acquitted, thanks in part to Cicero's acerbic defense, although he was almost certainly guilty. #librapondo #nunquamestfidelis

[10] St. Augustine's *De doctrina Christiana* devotes an entire chapter to removing ambiguity from Latin scripture by adding punctuation.

Here the black bar separates our past decisions—splitting the first 17 letters into four words—from the portion of the input string that we have not yet processed.

The next stage in our imagined process is to ***decide*** where the next word in the output sequence ends. For this specific example, there are four possibilities for the next output word—HE, HEAR, HEART, and HEARTH. We have *no idea* which of these choices, if any, is consistent with a complete segmentation of the input string. We could be "smart" at this point and try to *figure out* which choices are good, but that would require *thinking*! Instead, let's "stupidly" try every possibility by brute force, and let the Recursion Fairy do all the real work.

- First *tentatively* accept HE as the next word, and let the Recursion Fairy make the rest of the decisions.

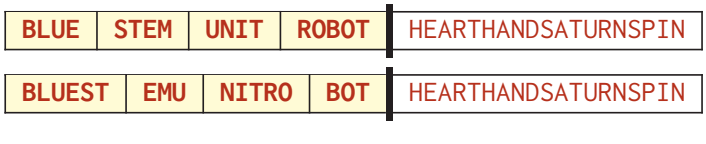
- Then *tentatively* accept HEAR as the next word, and let the Recursion Fairy make all remaining decisions.

- Then *tentatively* accept HEART as the next word, and let the Recursion Fairy make all remaining decisions.

- Finally, *tentatively* accept HEARTH as the next word, and let the Recursion Fairy make all remaining decisions.

As long as the Recursion Fairy reports success at least once, we report success. On the other hand, if the Recursion Fairy *never* reports success—in particular, if the set of possible next words is empty—then we report failure.

None of our past decisions affect which choices are available now; all that matters is the suffix of characters that we have not yet processed. In particular, several different sequences of past decisions might have led us to the same suffix, but they all leave us with exactly the same set of choices now.



Thus, we can simplify our picture of the recursive process by discarding *everything* left of the black bar:

We are now left with a simple and natural backtracking strategy: *Select the first output word, and recursively segment the rest of the input string.*

To get a complete recursive algorithm, we need a base case. Our recursive strategy breaks down when we reach the end of the input string, because there is no next word. Fortunately, the empty string has a unique segmentation into zero words!

Putting all the pieces together, we arrive at the following simple recursive algorithm:

```
SPLITTABLE(A[1..n]):
  if n = 0
      return TRUE
  for i ← 1 to n
      if ISWORD(A[1..i])
          if SPLITTABLE(A[i+1..n])
              return TRUE
  return FALSE
```

Index Formulation

In practice, passing arrays as input parameters to algorithm is rather slow; we should really find a more compact way to describe our recursive subproblems. *For purposes of designing the algorithm*, it's incredibly useful to treat the original input array as a global variable, and then reformulate the problem and the algorithm in terms of array indices instead of explicit subarrays.

For our string segmentation problem, the argument of any recursive call is always a ***suffix*** $A[i..n]$ of the original input array. So if we treat the input array $A[1..n]$ as a global variable, we can reformulate our recursive problem as follows:

> Given an index i, find a segmentation of the suffix $A[i..n]$.

To describe our algorithm, we need two boolean functions:

- For any indices i and j, let $\text{IsWord}(i, j) = \text{TRUE}$ if and only if the substring $A[i..j]$ is a word. (We're assuming this function is given to us.)
- For any index i, let $\mathit{Splittable}(i) = \text{TRUE}$ if and only if the suffix $A[i..n]$ can be split into words. (This is the function we need to implement.)

For example, $\text{IsWord}(1, n) = \text{TRUE}$ if and only if the entire input string is a single word, and $\mathit{Splittable}(1) = \text{TRUE}$ if and only if the entire input string can be segmented. Our earlier recursive strategy gives us the following recurrence:

$$\mathit{Splittable}(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(\text{IsWord}(i, j) \wedge \mathit{Splittable}(j+1)\big) & \text{otherwise} \end{cases}$$

This is *exactly* the same algorithm as we saw earlier; the only thing we've changed is the notation. The similarity is even more apparent if we rewrite the recurrence in pseudocode:

```
⟨⟨Is the suffix A[i..n] Splittable?⟩⟩
SPLITTABLE(i):
  if i > n
      return TRUE
  for j ← i to n
      if ISWORD(i, j)
          if SPLITTABLE(j + 1)
              return TRUE
  return FALSE
```

Although it may look like a trivial notational difference, using index notation instead of array notation is an important habit, not only to speed up backtracking algorithms in practice, but for developing dynamic programming algorithms, which we discuss in the next chapter.

♥Analysis

It should come as no surprise that most backtracking algorithms have exponential worst-case running times. Analyzing the precise running times of many of these algorithms requires techniques that are beyond the scope of this book. Fortunately, most of the backtracking algorithms we will encounter *in this book* are only intermediate results on the way to more efficient algorithms, which means their exact worst-case running time is not actually important. (First make it work; then make it fast.)

But just for fun, let's analyze the running time of our recursive algorithm SPLITTABLE. Because we don't know what ISWORD is doing, we can't know how long each call to ISWORD takes, so we're forced to analyze the running time in terms of the number of calls to ISWORD.[11] SPLITTABLE calls ISWORD on every prefix of the input string, and *possibly* calls itself recursively on every suffix of the output string. Thus, the "running time" of SPLITTABLE obeys the scary-looking recurrence

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

This really isn't as bad as it looks, especially once you've seen the trick.

First, we replace the $O(n)$ term with an explicit expression αn, for some unknown (and ultimately unimportant) constant α. Second, we conservatively

[11] In fact, as long as ISWORD runs in *polynomial* time, SPLITTABLE runs in $O(2^n)$ time.

assume that the algorithm actually makes every possible recursive call.[12] Then we can transform the "full history" recurrence into a "limited history" recurrence by subtracting the recurrence for $T(n-1)$, as follows:

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

This final recurrence simplifies to $T(n) = 2T(n-1) + \alpha$. At this point, we can confidently guess (or derive via recursion trees, or remember from our Tower of Hanoi analysis) that $T(n) = O(2^n)$; indeed, this upper bound is not hard to prove by induction from the original full-history recurrence.

Moreover, this analysis is tight. There are exactly 2^{n-1} possible ways to segment a string of length n—each input character either ends a word or doesn't, except the last input character, which always ends the last word. In the worst case, our SPLITTABLE algorithm explores each of these 2^{n-1} possibilities.

Variants

Now that we have the basic recursion pattern in hand, we can use it to solve many different variants of the segmentation problem, just as we did for the SUBSETSUM problem. Here I'll describe just one example; more variations are considered in the exercises. As usual, the original input to our problem is an array $A[1..n]$.

If a string can be segmented in more than one sequence of words, we may want to find the *best* segmentation according to some criterion; conversely, if the input string cannot be segmented into words, we may want to compute the best segmentation we can find, rather than merely reporting failure. To meet both of these goals, suppose we have access to a second function SCORE that takes a string w as input and returns a numerical value. For example, we might assign higher scores to longer or more common words, lower scores to shorter or more obscure words, slightly negative scores for minor spelling errors, and more negative scores to obvious non-words. Our goal is to find a segmentation that maximizes the sum of the scores of the segments.

[12] This assumption is wildly conservative for English *word* segmentation, since most strings of letters are not English words, but *not* for the similar problem of segmenting sequences of English *words* into grammatically correct English *sentences*. Consider, for example, a sequence of n copies of the word "buffalo", or n copies of the work "police", or n copies of the word "can", for any positive integer n. (At the Moulin Rouge, dances that are preservable in metal cylinders by other dances have the opportunity to fire dances that happen in prison restroom trash receptacles.)

For any index i, let $MaxScore(i)$ denote the maximum score of any segmentation of the suffix $A[i..n]$; we need to compute $MaxScore(1)$. This function satisfies the following recurrence:

$$MaxScore(i) = \begin{cases} 0 & \text{if } i > n \\ \max\limits_{i \le j \le n} \big(\textsc{Score}(A[i..j]) + MaxScore(j+1) \big) & \text{otherwise} \end{cases}$$

This is essentially the same recurrence as the one we developed for *Splittable*; the only difference is that the boolean operations $\vee$ and $\wedge$ have been replaced by the numerical operations max and $+$.

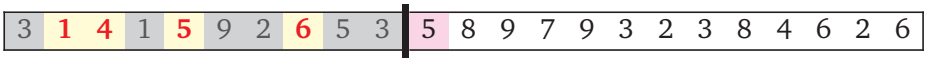
2.6 Longest Increasing Subsequence

For any sequence S, a ***subsequence*** of S is another sequence obtained from S by deleting zero or more elements, without changing the order of the remaining elements; the elements of the subsequence need not be contiguous in S. For example, when you drive down a major street in any city, you drive through a *sequence* of intersections with traffic lights, but you only have to stop at a *subsequence* of those intersections, where the traffic lights are red. If you're very lucky, you never stop at all: the empty sequence is a subsequence of S. On the other hand, if you're very unlucky, you may have to stop at every intersection: S is a subsequence of itself.

As another example, the strings BENT, ACKACK, SQUARING, and SUBSEQUENT are all subsequences of the string SUBSEQUENCEBACKTRACKING, as are the empty string and the entire string SUBSEQUENCEBACKTRACKING, but the strings QUEUE and EQUUS and TALLYHO are not. A subsequence whose elements are contiguous in the original sequence is called a ***substring***; for example, MASHER and LAUGHTER are both subsequences of MANSLAUGHTER, but only LAUGHTER is a substring.

Now suppose we are given a sequence of *integers*, and we need to find the longest subsequence whose elements are in increasing order. More concretely, the input is an integer array $A[1..n]$, and we need to compute the longest possible sequence of indices $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ such that $A[i_k] < A[i_{k+1}]$ for all k.

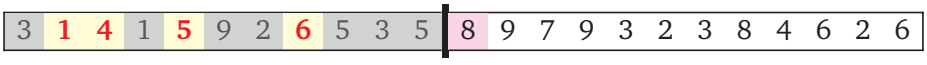
One natural approach to building this ***longest increasing subsequence*** is to *decide*, for each index j in order from 1 to n, whether or not to include $A[j]$ in the subsequence. Jumping into the middle of this decision sequence, we might imagine the following picture:



As in our segmentation example, the black bar separates our past decisions from the portion of the input we have not yet processed. Numbers we have

already decided to include are highlighted; numbers we have already decided to exclude are grayed out. (Notice that the numbers we've decided to include are increasing!) Our algorithm must decide whether or not to include the number immediately after the black bar.

In this example, we *cannot* include 5, because then the selected numbers would no longer be in increasing order. So let's skip ahead to the next decision:

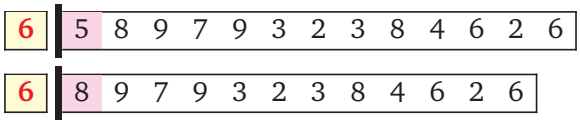


Now we *can* include 8, but it's not obvious whether we *should*. Rather than trying to be "smart", our backtracking algorithm will use simple brute force.

- First *tentatively* include the 8, and let the Recursion Fairy make the rest of the decisions.
- Then *tentatively* exclude the 8, and let the Recursion Fairy make the rest of the decisions.

Whichever choice leads to a longer increasing subsequence is the right one. (This is precisely the same recursion pattern we used to solve SUBSETSUM.)

Now for the key question: *What do we need to remember about our past decisions?* We can only include $A[j]$ if the resulting subsequence is in increasing order. If we assume (inductively!) that the numbers previously selected from $A[1..j-1]$ are in increasing order, then we can include $A[j]$ if and only if $A[j]$ is larger than the last number selected from $A[1..j-1]$. Thus, the only information we need about the past is ***the last number selected so far***. We can now revise our pictures by erasing everything we don't need:



So the problem our recursive strategy is *actually* solving is the following:

> Given an integer *prev* and an array $A[1..n]$, find the longest increasing subsequence of A in which every element is larger than *prev*.

As usual, our recursive strategy requires a base case. Our current strategy breaks down when we get to the end of the array, because there is no "next number" to consider. But an empty array has exactly one subsequence, namely, the *empty* sequence. Vacuously, every element in the empty sequence is larger than whatever value you want, and every pair of elements in the empty sequence appears in increasing order. Thus, the longest increasing subsequence of the empty array has length 0.

Here's the resulting recursive algorithm:

```
LISbigger(prev, A[1..n]):
  if n = 0
      return 0
  else if A[1] ≤ prev
      return LISbigger(prev, A[2..n])
  else
      skip ← LISbigger(prev, A[2..n])
      take ← LISbigger(A[1], A[2..n]) + 1
      return max{skip, take}
```

Okay, but remember that passing arrays around on the call stack is expensive; let's try to rephrase everything in terms of array indices, assuming that the array $A[1..n]$ is a global variable. The integer *prev* is typically an array element $A[i]$, and the remaining array is always a suffix $A[j..n]$ of the original input array. So we can reformulate our recursive *problem* as follows:

> Given two indices i and j, where $i < j$, find the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$.

Let $LISbigger(i, j)$ denote the *length* of the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$. Our recursive strategy gives us the following recurrence:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\left\{ \begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

Alternatively, if you prefer pseudocode:

```
LISbigger(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISbigger(i, j + 1)
  else
      skip ← LISbigger(i, j + 1)
      take ← LISbigger(j, j + 1) + 1
      return max{skip, take}
```

Finally, we need to connect our recursive strategy to the original problem: Finding the longest increasing subsequence of an array *with no other constraints*. The simplest approach is to add an artificial sentinel value $-\infty$ to the beginning of the array.

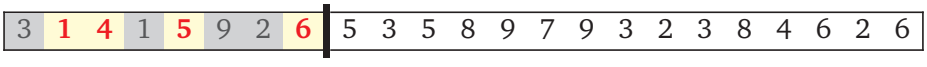
```
LIS(A[1..n]):
  A[0] ← −∞
  return LISbigger(0, 1)
```

The running time of LISbigger satisfies the Hanoi recurrence $T(n) \leq 2T(n-1)+O(1)$, which as usual 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.

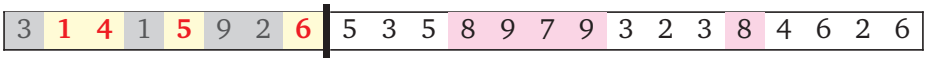
2.7 Longest Increasing Subsequence, Take 2

This is not the only backtracking strategy we can use to find longest increasing subsequences. Instead of considering the *input* array one element at a time, we could try to construct the *output* sequence one element at a time. That is, instead of "Is $A[i]$ the next element of the output sequence?", we could ask directly, "Where is the next element of the output sequence, if any?"

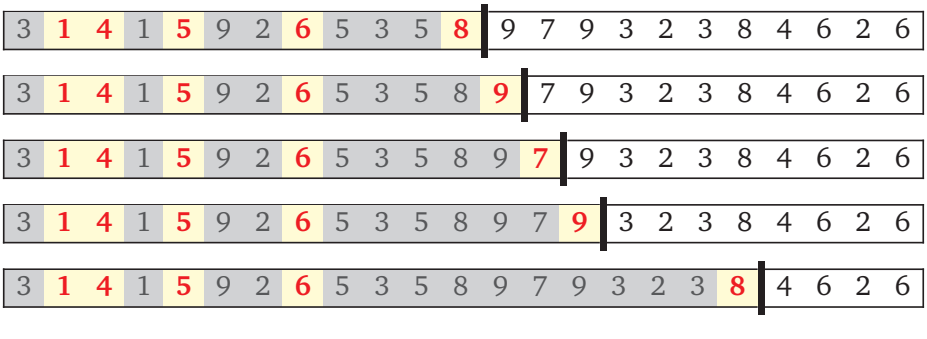
Jumping into the middle of this strategy, we might be faced with the following picture. Here we just decided to include the 6 just left of the black bar in our output sequence, and we need to decide which element to the right of the bar to include next.



Of course, we only need to consider numbers on the right that are bigger than 6.

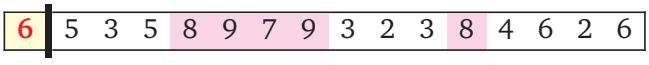


We have no idea which of those larger numbers is the best choice, and trying to cleverly *figure out* the best choice is just going to get us into trouble. Instead, we enumerate all possibilities by brute force, and let the Recursion Fairy evaluate each one.



The subset of numbers we can consider as the next element depends *only* on the last number we decided to include. Thus, we can simplify our picture of

the decision procedure by discarding all the other numbers that we've already processed.



The remaining numbers are just a suffix of the original input array. Thus, if we think of the input array $A[1..n]$ as a global variable, we can formally express our recursive problem in terms of indices as follows:

> Given an index i, find the longest increasing subsequence of $A[i..n]$ that begins with $A[i]$.

Let $LISfirst(i)$ denote the length of the longest increasing subsequence of $A[i..n]$ that begins with $A[i]$. We can now formulate our recursive backtracking strategy as the following recursive definition:

$$LISfirst(i) = 1 + \max\left\{LISfirst(j) \;\middle|\; j > i \text{ and } A[j] > A[i]\right\}$$

Because we are dealing with sets of natural numbers, we define $\max\varnothing = 0$. Then we automatically have $LISfirst(i) = 1$ if $A[j] \le A[i]$ for all $j > i$; in particular, $LISfirst(n) = 1$. These are the base cases for our recurrence. We can also express this recursive definition in pseudocode as follows:

```
LISFIRST(i):
  best ← 0
  for j ← i + 1 to n
      if A[j] > A[i]
          best ← max{best, LISFIRST(j)}
  return 1 + best
```

Finally, we need to reconnect this recursive algorithm to our original problem—finding the longest increasing subsequence without knowing its first element. One natural approach that works is to try all possible first elements by brute force. Equivalently, we can add a sentinel element $-\infty$ to the beginning of the array, find the longest increasing subsequence that starts with the sentinel, and finally ignore the sentinel.

```
LIS(A[1..n]):
  best ← 0
  for i ← 1 to n
      best ← max{best, LISFIRST(i)}
  return best
```

```
LIS(A[1..n]):
  A[0] ← −∞
  return LISFIRST(0) − 1
```

2.8 Optimal Binary Search Trees

Our final example combines recursive backtracking with the divide-and-conquer strategy. Recall that the running time for a successful search in a binary search tree is proportional to the number of ancestors of the target node.[13] As a result, the worst-case search time is proportional to the depth of the tree. Thus, to minimize the worst-case search time, the height of the tree should be as small as possible; by this metric, the ideal tree is perfectly balanced.

In many applications of binary search trees, however, it is more important to minimize the total cost of several searches rather than the worst-case cost of a single search. If x is a more frequent search target 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 with depth $\Omega(n)$ might actually be the best choice!

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

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 $v_1, v_2, \ldots, v_n$ be the nodes of T, indexed in sorted order, so that each node v_i stores the corresponding key $A[i]$. Then ignoring constant factors, the total cost of performing all the binary searches is given by the following expression:

$$Cost(T, f[1..n]) := \sum_{i=1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } T \qquad (*)$$

Now suppose v_r is the root of T; by definition, v_r is an ancestor of every node in T. If $i < r$, then all ancestors of v_i except the root are in the left subtree of T. Similarly, if $i > r$, then all ancestors of v_i except the root are in the right subtree of T. Thus, we can partition the cost function into three parts as follows:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \cdot \#\text{ancestors of } v_i \text{ in } \mathit{left}(T) + \sum_{i=r+1}^{n} f[i] \cdot \#\text{ancestors of } v_i \text{ in } \mathit{right}(T)$$

The second and third summations look exactly like our original definition $(*)$

[13] 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.

for $Cost(T, f[1..n])$. Simple substitution now gives us a recurrence for $Cost$:

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

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_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r. Then the recursive definition of $Cost(T, f)$ immediately implies that the left subtree $left(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $right(T_{\text{opt}})$ 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 construct the rest of the optimal tree.**

More generally, let $OptCost(i, k)$ denote the total cost of the optimal search tree for the interval of frequencies $f[i..k]$. This function obeys the following recurrence.

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle\sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

The base case correctly indicates that the minimum possible cost to perform zero searches into the empty set is zero! Our original problem is to compute $OptCost(1, n)$.

This recursive definition can be translated mechanically into a recursive backtracking algorithm to compute $OptCost(1, n)$. Not surprisingly, the running time of this algorithm is exponential. In the next chapter, we'll see how to reduce the running time to polynomial, so there's not much point in computing the precise running time...

♥ Analysis

...unless you're into that sort of thing. Just for the fun of it, let's figure out how slow this backtracking algorithm actually is. The running time satisfies the recurrence

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

The $O(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$. Yeah, that's one ugly recurrence, but we can solve it using exactly the same

subtraction trick we used before. We replace the $O(\,)$ notation with an explicit constant, regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

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

Hey, that doesn't look so bad after all. The recursion tree method immediately gives us the solution $\boldsymbol{T(n) = O(3^n)}$ (or we can just guess and confirm by induction).

This analysis implies that our recursive algorithm does *not* examine all possible binary search trees! The number of binary search trees with n vertices satisfies the recurrence

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

which has the closed-form solution $N(n) = \Theta(4^n/\sqrt{n})$. (No, that's not obvious.) Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees for each root. A full enumeration of binary search trees would consider all possible *pairs* of left and right subtrees; hence the product in the recurrence for $N(n)$.

Exercises

1. Describe recursive algorithms for the following generalizations of the SubsetSum problem:

   (a) Given an array $X[1..n]$ of positive integers and an integer T, compute the *number* of subsets of X whose elements sum to T.

   (b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers and an integer T, where each $W[i]$ denotes the *weight* of the corresponding element $X[i]$, compute the *maximum weight* subset of X whose elements sum to T. If no subset of X sums to T, your algorithm should return $-\infty$.

2. Describe recursive algorithms for the following variants of the text segmentation problem. Assume that you have a subroutine IsWord that takes an

array of characters as input and returns TRUE if and only if that string is a "word".

(a) Given an array $A[1..n]$ of characters, compute the number of partitions of A into words. For example, given the string ARTISTOIL, your algorithm should return 2, for the partitions ARTIST·OIL and ART·IS·TOIL.

(b) Given two arrays $A[1..n]$ and $B[1..n]$ of characters, decide whether A and B can be partitioned into words at the same indices. For example, the strings BOTHEARTHANDSATURNSPIN and PINSTARTRAPSANDRAGSLAP can be partitioned into words at the same indices as follows:

BOT·HEART·HAND·SAT·URNS·PIN
PIN·START·RAPS·AND·RAGS·LAP

(c) Given two arrays $A[1..n]$ and $B[1..n]$ of characters, compute the number of different ways that A and B can be partitioned into words at the same indices.

3. An *addition chain* for an integer n is an increasing sequence of integers that starts with 1 and ends with n, such that each entry after the first is the sum of two earlier entries. More formally, the integer sequence $x_0 < x_1 < x_2 < \cdots < x_\ell$ is an addition chain for n if and only if

- $x_0 = 1$,
- $x_\ell = n$, and
- for every index $k > 0$, there are indices $i \le j < k$ such that $x_k = x_i + x_j$.

The ℓength of an addition chain is the number of elements minus 1; we don't bother to count the first entry. For example, $\langle 1, 2, 3, 5, 10, 20, 23, 46, 92, 184, 187, 374\rangle$ is an addition chain for 374 of length 11.

(a) Describe a recursive backtracking algorithm to compute a minimum-length addition chain for a given positive integer n. ***Don't*** analyze or optimize your algorithm's running time, except to satisfy your own curiosity. A correct algorithm whose running time is exponential in n is sufficient for full credit. *[Hint: This problem is a lot more like n Queens than text segmentation.]*

♥(b) Describe a recursive backtracking algorithm to compute a minimum-length addition chain for a given positive integer n *in time that is sub-exponential in n*. *[Hint: You may find the results of certain Egyptian rope-fasteners, Indus-River prosodists, and Russian peasants helpful.]*

4. (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 $X[1..n]$ of numbers ***bitonic*** if there is an index i with $1 < i < n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Give a simple recursive definition for the function $lbs(A)$, which gives the length of the longest bitonic subsequence of an arbitrary array A of integers.

(d) 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.

(e) 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.

(f) Call a sequence $X[1..n]$ *convex* 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 convex subsequence of an arbitrary array A of integers.

5. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \le n$.

(a) Describe a recursive backtracking algorithm to determine whether X is a subsequence of Y. For example, the string PPAP is a subsequence of the string PENPINEAPPLEAPPLEPEN.

(b) Describe a recursive backtracking algorithm to find the smallest number of symbols that can be removed from Y so that X is no longer a subsequence. Equivalently, your algorithm should find the longest subsequence of Y that is *not* a supersequence of X. For example, after removing removing two symbols from the string PENPINEAPPLEAPPLEPEN, the string PPAP is no longer a subsequence.

♥(c) Describe a recursive backtracking algorithm to determine whether X occurs as two *disjoint* subsequences of Y. For example, the string PPAP appears as two disjoint subsequences in the string PENPINEAPPLEAPPLEPEN.

Don't analyze the running times of your algorithms, except to satisfy your own curiosity. All three algorithms run in exponential time; we'll improve that later, so the precise running time isn't particularly important.

6. This problem asks you to design backtracking algorithms to find the cost of an optimal binary search tree that satisfies additional balance constraints. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. This is exactly the same cost function as described in Section 2.8. But now your task is to compute an optimal tree that satisfies some additional constraints.

   (a) ***AVL trees*** were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most one.

   Describe a recursive backtracking algorithm to construct an optimal AVL tree for a given set of search keys and frequencies.

   (b) ***Symmetric binary B-trees*** are another self-balancing binary trees, first described by Rudolf Bayer in 1972; these are better known by the name ***red-black trees***, after a somewhat simpler reformulation by Leo Guibas and Bob Sedgwick in 1978. A red-black tree is a binary search tree with the following additional constraints:

   - Every node is either red or black.
   - Every red node has a black parent.
   - Every root-to-leaf path contains the same number of black nodes.

   Describe a recursive backtracking algorithm to construct an optimal red-black tree for a given set of search keys and frequencies.

   (c) ***AA trees*** were proposed by proposed by Arne Andersson in 1993 and slightly simplified (and named) by Mark Allen Weiss in 2000. AA trees are also known as *left-leaning red-black trees*, after a symmetric reformulation (with different rebalancing algorithms) by Bob Sedgewick in 2006. An AA tree is a red-black tree with one additional constraint:

   - No left child is red.[14]

   Describe a recursive backtracking algorithm to construct an optimal AA tree for a given set of search keys and frequencies.

   Don't analyze the running times of your algorithms, except to satisfy your own curiosity. All three algorithms run in exponential time; we'll improve that later, so the precise running times aren't particularly important.

For more backtracking exercises, see the next chapter!

[14] Sedgwick's reformulation requires that no *right* child is red. Whatever. Andersson and Sedgwick are strangely silent on which end of the egg to eat first.