```
⟨⟨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 subset sum 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.[6]
- For the subset sum 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 must 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 find 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 String 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:[7]

[6]Many games 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 the entire history of previous moves.

[7]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

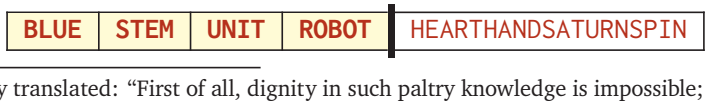
PRIMVSDIGNITASINTAMTENVISCIENTIANONPOTEST
ESSERESENIMSVNTPARVAEPROPEINSINGVLISLITTERIS
ATQVEINTERPVNCTIONIBUSVERBORVMOCCVPATAE

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*.[8] 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,[9] 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 subset sum 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:



[8] 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

[9] 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.

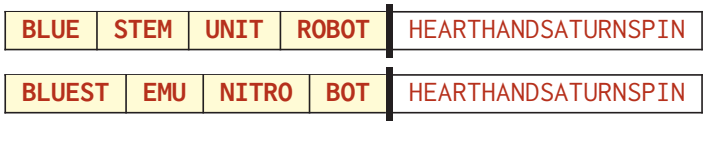
- Then *tentatively* accept HEAR as the next word, and let the Recursion Fairy make all remaining decisions.

- 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 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.[10] 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

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

assume that the algorithm actually makes every possible recursive call.[11] 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 $T(n) = 2T(n-2) + \alpha$. At this point, we can confidently guess (or derive via recursion trees) 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 subset sum 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 and/or more frequent words, and lower negative scores to longer and/or more ridiculous non-words. Our goal is to find a segmentation that maximizes the sum of the scores of the segments.

[11] 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 $+$.

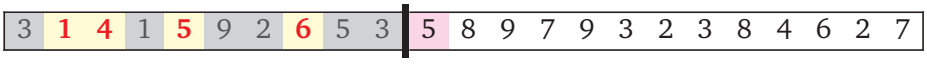
2.6 Longest Increasing Subsequence

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.

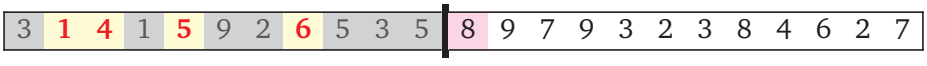
One natural approach to building this ***longest increasing subsequence*** is to *decide*, for each index j in order from 1 to n, whether or not to include $A[j]$ in the subsequence. Jumping into the middle of this decision sequence, we might imagine the following picture:



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:

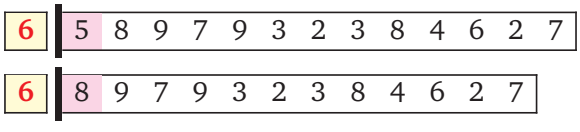


Now we *can* include 8, but it's not obvious whether we *should*. Rather than trying to be "smart", our backtracking algorithm will use simple brute force.

- 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 the subset sum problem.)

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[1] ≤ prev
      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.

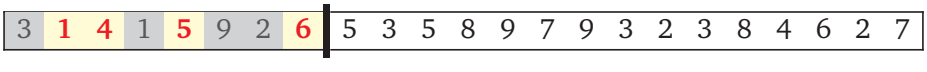
```
LIS(A[1..n]):
  A[0] ← −∞
  return LISBIGGER(0, 1)
```

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.

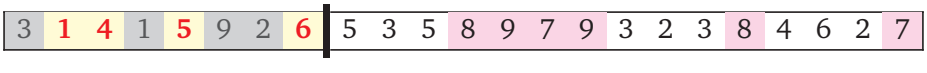
2.7 Longest Increasing Subsequence, Take 2

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 $A[i]$ is the next element of the output sequence?", we could ask directly, "Where is the next element of the output sequence, if any?"

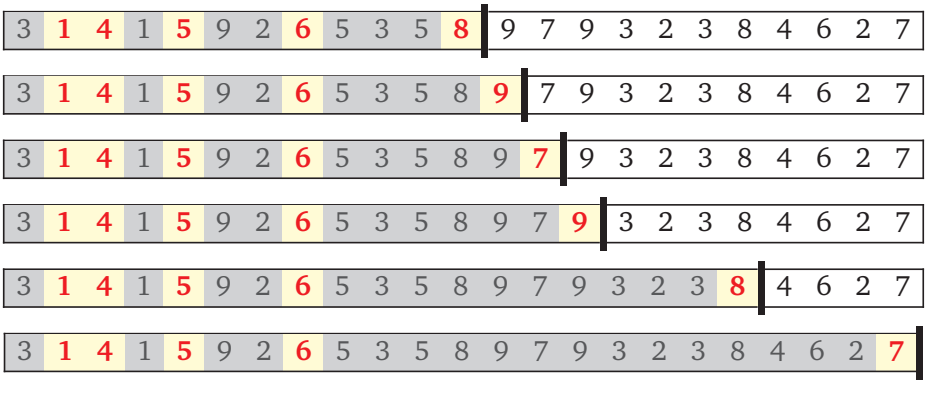
Jumping into the middle of this strategy, we might be faced with the following picture. Here we just decided to include the 6 just left of the black bar in our output sequence, and we need to decide which element to the right of the bar to include next.



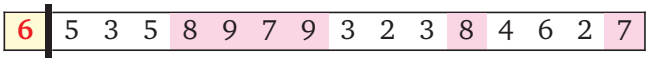
Of course, we only need to consider numbers on the right that are bigger than 6.



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 continue the decision process recursively for 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 $A[j] \geq A[i]$ 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, 1 + LISFIRST(j)}
  return 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(1)}
  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.[12] 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 $(*)$

[12] 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-from 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 SUBSET-SUM:

   (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; 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 Sedgwick in 2006. An AA-tree is a red-black tree with one additional constraint:

   - No left child is red.[13]

   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 time isn't particularly important.

For more backtracking exercises, see the next chapter!

[13] 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.

Potes enim videre in hac margine, qualiter hoc operati fuimus, scilicet quod iunximus primum numerum cum secundo, videlicet 1 cum 2; et secundum cum tercio; et tercium cum quarto; et quartum cum quinto, et sic deinceps.…

[You can see in the margin here how we have worked this; clearly, we combined the first number with the second, namely 1 with 2, and the second with the third, and the third with the fourth, and the fourth with the fifth, and so forth.…]

— Leonardo Pisano, *Liber Abaci* (1202)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

You know what a learning experience is?
A learning experience is one of those things that says,
"You know that thing you just did? Don't do that."

— Douglas Adams, *The Salmon of Doubt* (2002)

3

Dynamic Programming

3.1 Mātrāvṛtta

One of the earliest examples of recursion arose in India more than 2000 years ago, in the study of poetic meter, or prosody. Classical Sanskrit poetry distinguishes between two types of syllables (*akṣara*): *light* (*laghu*) and *heavy* (*guru*). In one class of meters, variously called *mātrāvṛtta* or *mātrāmeru* or *mātrāchandạ*, each line of poetry consists of a fixed number of "beats" (*mātrā*), where each light syllable lasts one beat and each heavy syllable lasts two beats. The formal study of *mātrā-vṛtta* dates back to the *Chandaḥśāstra*, written by the scholar Piṅgala between 600BCE and 200BCE. Piṅgala observed that there are exactly five 4-beat meters: ——, —••, •—•, ••—, and ••••. (Here each "—" represents a long syllable and each "•" represents a short syllable.)[1]

[1]In Morse code, a "dah" lasts three times as long as a "dit", but each "dit" or "dah" is followed by a pause with the same duration as a "dit". Thus, each "dit-pause" is a *laghu akṣara*, each

Although Piṅgala's text *hints* at a systematic rule for counting meters with a given number of beats,[2] it took about a millennium for that rule to be stated explicitly. In the 7th century CE, another Indian scholar named Virahāṇka wrote a commentary on Piṅgala's work, in which he observed that the number of meters with n beats is the sum of the number of meters with $(n-2)$ beats and the number of meters with $(n-1)$ beats. In more modern notation, Virahāṇka's observation implies a recurrence for the total number $M(n)$ of n-beat meters:

$$M(n) = M(n-2) + M(n-1)$$

It is not hard to see that $M(0) = 1$ (there is only one empty meter) and $M(1) = 1$ (the only one-beat meter consists of a single short syllable).

The same recurrence reappeared in Europe about 500 years after Virahāṇka, in Leonardo Pisano's 1202 treatise *Liber Abaci*, one of the most influential early European works on "algorism". In full compliance with Stigler's Law of Eponymy,[3] the modern *Fibonacci numbers* are defined using Virahāṇka's recurrence, but with different base cases:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

In particular, we have $M(n) = F_{n+1}$ for all n.

Backtracking Can Be Slow

The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them. Here is the same algorithm written in pseudocode:

"dah-pause" is a *guru akṣara*, and there are exactly five letters (M, D, R, U, and H) whose codes last four *mātrā*.

[2]The *Chandaḥśāstra* contains *two* systematic rules for listing all meters with a given number of *syllables*, which correspond roughly to writing numbers in binary from left to right (like Greeks) or from right to left (like Egyptians). The same text includes a recursive algorithm to compute 2^n (the number of meters with n syllables) by repeated squaring, and (arguably) a recursive algorithm to compute binomial coefficients (the number of meters with k short syllables and n syllables overall).

[3]*"No scientific discovery is named after its original discoverer."* In his 1980 paper that gives the law its name, the statistician Stephen Stigler jokingly claimed that this law was first proposed by sociologist Robert K. Merton. However, similar statements were previously made by Vladimir Arnol'd in the 1970's ("Discoveries are rarely attributed to the correct person."), Carl Boyer in 1968 ("Clio, the muse of history, often is fickle in attaching names to theorems!"), Alfred North Whitehead in 1917 ("Everything of importance has been said before by someone who did not discover it."), and even Stephen's father George Stigler in 1966 ("If we should ever encounter a case where a theory is named for the correct man, it will be noted."). We will see *many* other examples of Stigler's law in this book.

```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

Unfortunately, this naive recursive algorithm is horribly slow. Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. Let $T(n)$ denote the number of recursive calls to RecFibo; this function satisfies the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1,$$

which looks an awful lot like the recurrence for Fibonacci numbers themselves! Writing out the first several values of $T(n)$ suggests the closed-form solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$, which we can verify by induction (hint, hint). So computing F_n using this algorithm takes about twice as long as just counting to F_n. Methods beyond the scope of this book[4] imply that $F_n = \Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803$ is the so-called *golden ratio*. In short, the running time of this recursive algorithm is exponential in n.

We can actually see this exponential growth directly as follows. Think of the recursion tree for RecFibo as a binary tree of additions, with only 0s and 1s at the leaves. Since the eventual output is F_n, exactly F_n of the leaves must have value 1; these leaves represent the calls to RecRibo(1). An easy inductive argument (hint, hint) implies that RecFibo(0) is called exactly F_{n-1} times. (If we just want an asymptotic bound, it's enough to observe that the number of calls to RecFibo(0) is at most the number of calls to RecFibo(1).) Thus, the recursion tree has exactly $F_n + F_{n-1} = F_{n+1} = O(F_n)$ leaves, and therefore, because it's a full binary tree, $2F_{n+1} - 1 = O(F_n)$ nodes altogether.

Memo(r)ization: Remember Everything

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecFibo(n) results in one recursive call to RecFibo($n-1$), two recursive calls to RecFibo($n-2$), three recursive calls to RecFibo($n-3$), five recursive calls to RecFibo($n-4$), and in general F_{k-1} recursive calls to RecFibo($n-k$) for any integer $0 \le k < n$. Each call is recomputing some Fibonacci number from scratch.

We can speed up our recursive algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later.

[4] See http://algorithms.wtf for notes on solving recurrences.

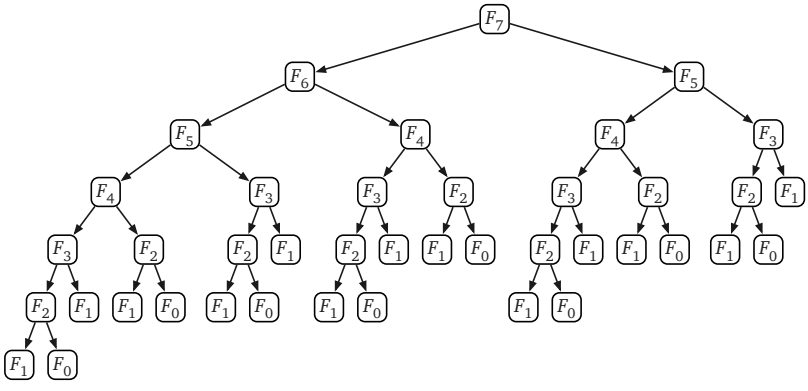


Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

This optimization technique, now known as *memoization*, is usually credited to Donald Michie in 1967, but essentially the same technique was proposed in 1959 by Arthur Samuel.[5]

```
MemFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

Memoization clearly decreases the running time of the algorithm, but by how much? If we actually trace through the recursive calls made by MemFibo, we find that the array $F[\,]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. This pattern can be verified by induction: Each entry $F[i]$ is filled only after its predecessor $F[i-1]$. If we ignore the time spent in recursive calls, it requires only constant time to evaluate the recurrence for each Fibonacci number F_i. But by design, the recurrence for F_i is evaluated only once for each index i. We conclude that MemFibo performs only $O(n)$ additions, an *exponential* improvement over the naïve recursive algorithm!

[5]Michie proposed that programming languages should support an abstraction he called a "memo function", consisting of both a standard function ("rule") and a dictionary ("rote"), instead of separately supporting arrays and functions. Whenever a memo function computes a function value for the first time, it "memorises" (yes, with an R) that value into its dictionary. Michie was inspired by Samuel's use of "rote learning" to speed up the recursive evaluation of checkers game trees; Michie describes his more general proposal as "enabling the programmer to 'Samuelize' any functions he pleases." (As far as I can tell, Michie never actually used the term "memoisation".) Memoization was used even earlier by Claude Shannon's maze-solving robot "Theseus", which he designed and constructed in 1950.

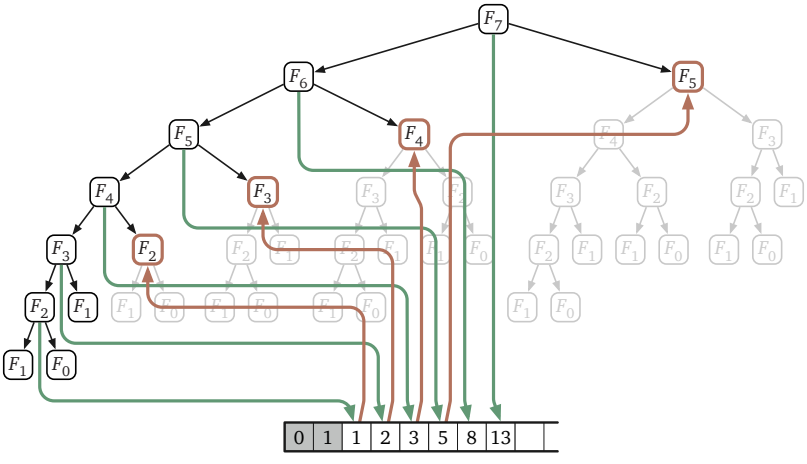


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Dynamic Programming: Fill Deliberately

Once we see how the array $F[\,]$ is filled, we can replace the memoized recurrence with a simple for-loop that *intentionally* fills the array in order, instead of relying on a more complicated recursive algorithm to do it for us accidentally.

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i−1] + F[i−2]
  return F[n]
```

Now the time analysis is immediate: ITERFIBO clearly uses $\boldsymbol{O(n)}$ ***additions*** and stores $O(n)$ *integers*.

This is our first explicit ***dynamic programming*** algorithm. The dynamic programming paradigm was formalized and popularized by Richard Bellman in the mid-1950s, while working at the RAND Corporation, although he was far from the first to use the technique. In particular, this iterative algorithm for Fibonacci numbers was already proposed by Virahāṇka and later Sanskrit prosodists in the 12th century, and again by Fibonacci at the turn of the 13th century![6]

[6]More general dynamic programming techniques were independently deployed several times in the late 1930s and early 1940s. For example, Pierre Massé used dynamic programming algorithms to optimize the operation of hydroelectric dams in France during the Vichy regime. John von Neumann and Oskar Morgenstern developed dynamic programming algorithms to determine the winner of any two-player game with perfect information (for example, checkers). Alan Turing and his cohorts used similar methods as part of their code-breaking efforts at

Many years after the fact, Bellman claimed that he deliberately chose the name "dynamic programming" to hide the mathematical character of his work from his military bosses, who were actively hostile toward anything resembling mathematical research.[7] The word "programming" does not refer to writing code, but rather to the older sense of *planning* or *scheduling*, typically by filling in a table. For example, sports programs and theater programs are schedules of important events (with ads); television programming involves filling each available time slot with a show (and ads); degree programs are schedules of classes to be taken (with ads). The Air Force funded Bellman and others to develop methods for constructing training and logistics schedules, or as they called them, "programs". The word "dynamic" was not only a reference to the multistage, time-varying processes that Bellman and his colleagues were attempting to optimize, but also a marketing buzzword that would resonate with the Futuristic Can-Do Zeitgeist™ of post-WW II America.[8] Thanks in part to Bellman's proselytizing, dynamic programming is now a standard tool for multistage planning in economics, robotics, control theory, and several other disciplines.

Don't Remember Everything After All

In many dynamic programming algorithms, it is not necessary to retain *all* intermediate results through the entire computation. For example, we can significantly reduce the space requirements of our algorithm IterFibo by maintaining only the two newest elements of the array:

Bletchley Park. Both Massé's work and von Neumann and Mergenstern's work were first published in 1944, six years before Bellman coined the phrase "dynamic programming". The details of Turing's "Banburismus" were kept secret until the mid-1980s.

[7]Charles Erwin Wilson became Secretary of Defense in January 1953, after a dozen years as the president of General Motors. "Engine Charlie" reorganized the Department of Defense and significantly decreased its budget in his first year in office, with the explicit goal of running the Department much more like an industrial corporation. Bellman described Wilson in his 1984 autobiography as follows:

> We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word "*research*". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term "*research*" in his presence. You can imagine how he felt, then, about the term "*mathematical*". . . . I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?

However, Bellman's first published use of the term "dynamic programming" already appeared in 1952, several months before Wilson took office, so this story is at least *slightly* embellished.

[8]. . . and just possibly a riff on the iconic brand name "Dynamic-Tension" for Charles Atlas's famous series of exercises, which Charles Roman coined in 1928. Hero of the Beach!

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(This algorithm uses the non-standard but consistent base case $F_{-1} = 1$ so that ITERFIBO2(0) returns the correct value 0.) Although saving space can be absolutely crucial in practice, we won't focus on space issues in this book.

♥3.2 Aside: Even Faster Fibonacci Numbers

Although the previous algorithm is simple and attractive, it is *not* the fastest algorithm to compute Fibonacci numbers. We can derive a faster algorithm by exploiting the following matrix reformulation of the Fibonacci recurrence:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ has exactly the same effect as one iteration of the inner loop of ITERFIBO2. It follows that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

So if we want the nth Fibonacci number, we only need to compute the nth power of the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$. If we use repeated squaring,[9] computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, each of which reduces to a constant number of integer multiplications and additions. Thus, we can compute F_n in only ***O*(log *n*) *integer arithmetic operations***.

We can achieve the same speedup using the identity $F_n = F_m F_{n-m-1} + F_{m+1} F_{n-m}$, which holds (by induction!) for all integers m and n. In particular, this identity implies the following mutual recurrence for pairs of adjacent Fibonacci numbers:

$$F_{2n-1} = F_{n-1}^2 + F_n^2$$
$$F_{2n} = F_n(F_{n-1} + F_{n+1}) = F_n(2F_{n-1} + F_n)$$

[9] as suggested by Piṅgala for powers of 2 elsewhere in *Chandaḥśāstra*

(We can also derive this mutual recurrence directly from the matrix-squaring algorithm.) These recurrences translate directly into the following algorithm:

```
⟨⟨Compute the pair F_{n-1}, F_n⟩⟩
FastRecFibo(n):
  if n = 1
      return 0, 1
  m ← ⌊n/2⌋
  hprv, hcur ← FastRecFibo(m)      ⟨⟨F_{m-1}, F_m⟩⟩
  prev ← hprv² + hcur²             ⟨⟨F_{2m-1}⟩⟩
  curr ← hcur · (2 · hprv + hcur)  ⟨⟨F_{2m}⟩⟩
  next ← prev + curr               ⟨⟨F_{2m+1}⟩⟩
  if n is even
      return prev, curr
  else
      return curr, next
```

Our standard recursion tree technique implies that this algorithm performs only $O(\log n)$ integer arithmetic operations.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

Whoa! Not so fast!

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n \log_{10} \phi \approx n/5$ decimal digits long, or $n \log_2 \phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic *time* — we need $\Omega(n)$ time just to write down the answer!

The way out of this apparent paradox is to observe that ***we can't perform arbitrary-precision arithmetic in constant time***. Let $M(n)$ denote the time required to multiply two n-digit numbers. The running time of FastRecFibo satisfies the recurrence $T(n) = T(\lfloor n/2 \rfloor) + M(n)$, which solves to $T(n) = \boldsymbol{O(M(n))}$ via recursion trees. The fastest integer multiplication algorithm known (as of 2018) runs in time $O(n \log n \, 4^{\log^* n})$, so that is also the running time of the fastest algorithm known (as of 2018) to compute Fibonacci numbers.

Is this algorithm slower than our "linear-time" iterative algorithms? Actually, no! Addition isn't free, either! Adding two n-digit numbers takes $O(n)$ time, so the running time of the iterative algorithms IterFibo and IterFibo2 is $\boldsymbol{O(n^2)}$. (Do you see why?) So FastRecFibo is significantly faster than the iterative algorithms, just not *exponentially* faster.

In the original recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct