

*'Tis a lesson you should heed,  
 Try, try again;  
 If at first you don't succeed,  
 Try, try again;  
 Then your courage should appear,  
 For, if you will persevere,  
 You will conquer, never fear;  
 Try, try again.*

— Thomas H. Palmer, *The Teacher's Manual: Being an Exposition of an Efficient and Economical System of Education Suited to the Wants of a Free People* (1840)

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

— Yogi Berra

### 3 Backtracking

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

#### 3.1 $n$ Queens

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

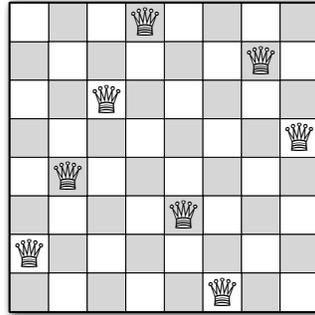
Obviously, in any solution to the  $n$ -Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array  $Q[1..n]$ , where  $Q[i]$  indicates which square in row  $i$  contains a queen, or 0 if no queen has yet been placed in row  $i$ . To find a solution,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

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

```

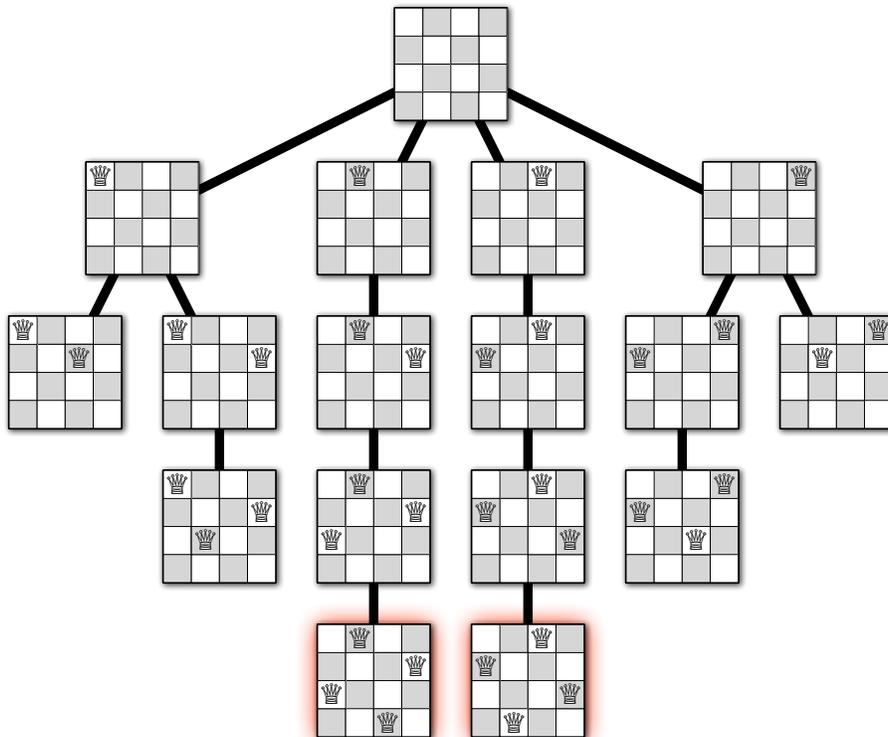
RECURSIVEQUEENS(Q[1..n], r):
  if r = n + 1
    print Q
  else
    for j ← 1 to n
      legal ← TRUE
      for i ← 1 to r - 1
        if (Q[i] = j) or (Q[i] = j + r - i) or (Q[i] = j - r + i)
          legal ← FALSE
      if legal
        Q[r] ← j
        RECURSIVEQUEENS(Q[1..n], r + 1)

```



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

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

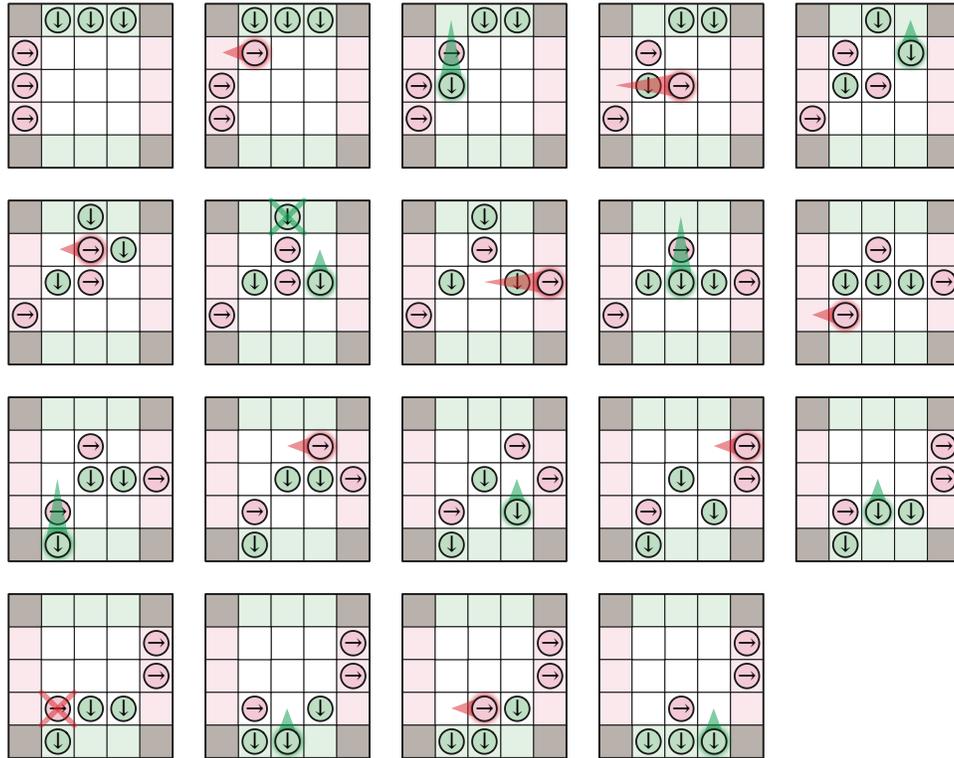


The complete recursion tree for our algorithm for the 4 queens problem.

### 3.2 Game Trees

Consider the following simple two-player game played on an  $n \times n$  square grid with a border of squares. Let's call the players Horace and Vera. 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 to the right;

symmetrically, Vera's tokens start in the top border, one in each column, and move down. 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. However, if no legal moves or jumps are available, Horace simply passes. Similarly, Vera either moves or jump 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.



Vera wins the 3 × 3 game.

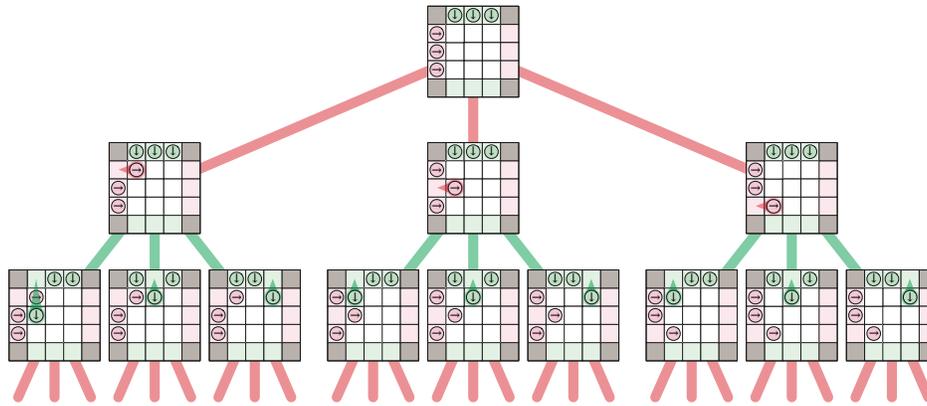
We can use a simple backtracking algorithm to determine the best move for each player at each turn. The *state* of the game consists of the locations of all the pieces and the player whose turn it is. We recursively define a game state to be *good* or *bad* as follows:

- A game state is *bad* if all the opposing player's tokens have reached their goals.
- A game state is *good* if the current player can move to a state that is bad for the opposing player.
- A configuration is *bad* if every move leads to a state that is good for the opposing player.

This recursive definition immediately suggests a recursive backtracking algorithm to determine whether a given state of the game is good or bad. Moreover, for any good state, the backtracking algorithm finds a move leading to a bad state for the opposing player. Thus, 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.

All computer game players 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<sup>1</sup> to *prune* the game tree, by ignoring states

<sup>1</sup>A heuristic is an algorithm that doesn't work.



The first two levels of the game tree.

that are obviously good or bad (or at least obviously 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.

### 3.3 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**, thanks to the subsets  $\{8, 7\}$  or  $\{7, 5, 3\}$  or  $\{6, 9\}$  or  $\{5, 10\}$ . On the other hand, if  $X = \{11, 6, 5, 1, 7, 13, 12\}$  and  $T = 15$ , the answer is **FALSE**.

There are two trivial cases. If the target value  $T$  is zero, then we can immediately return **TRUE**, because empty set is a subset of every set  $X$ , and the elements of the empty set add up to zero.<sup>2</sup> On the other hand, if  $T < 0$ , or if  $T \neq 0$  but the set  $X$  is empty, then we can immediately return **FALSE**.

For the general case, consider an arbitrary element  $x \in X$ . (We've already handled the case where  $X$  is empty.) There is a subset of  $X$  that sums to  $T$  if and only if one of the following statements is true:

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

In the first case, there must be a subset of  $X \setminus \{x\}$  that sums to  $T - x$ ; in the second case, there must be a subset of  $X \setminus \{x\}$  that sums to  $T$ . So we can solve **SUBSETSUM**( $X, T$ ) by reducing it to two simpler instances: **SUBSETSUM**( $X \setminus \{x\}, T - x$ ) and **SUBSETSUM**( $X \setminus \{x\}, T$ ). Here's how the resulting recursive algorithm might look if  $X$  is stored in an array.

```

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

```

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

<sup>2</sup>There's no base case like the vacuous base case!

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.

The running time  $T(n)$  clearly satisfies the recurrence  $T(n) \leq 2T(n-1) + O(1)$ , which we can solve using either recursion trees or annihilators (or just guessing) to obtain the upper bound  $T(n) = O(2^n)$ . In the worst case, the recursion tree for this algorithm is a complete binary tree with depth  $n$ .

Here is a similar recursive algorithm that actually *constructs* a subset of  $X$  that sums to  $T$ , if one exists. This algorithm also runs in  $O(2^n)$  time.

```

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

```

### 3.4 Longest Increasing Subsequence

Now suppose we are given a sequence of integers, and we want to find the longest subsequence whose elements are in increasing order. More concretely, the input is an array  $A[1..n]$  of integers, and we want to find the longest sequence of indices  $1 \leq i_1 < i_2 < \dots < i_k \leq n$  such that  $A[i_j] < A[i_{j+1}]$  for all  $j$ .

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

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

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

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

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

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

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

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

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

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

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

The running time of this algorithm satisfies the recurrence  $T(n) \leq 2T(n-1) + O(1)$ , 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.

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

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

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

The FILTER subroutine clearly runs in  $O(n)$  time, so the running time of LIS satisfies the recurrence  $T(n) \leq 2T(n-1) + O(n)$ , which solves to  $T(n) \leq O(2^n)$  by the annihilator method. This upper bound pessimistically assumes that FILTER never actually removes any elements; indeed, if the input sequence is sorted in increasing order, this assumption is correct.

### 3.5 Optimal Binary Search Trees

Our next 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.<sup>3</sup> 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 ‘popular’ 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 of 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  $v_i$  denote the node that stores  $A[i]$ , and let  $r$  be the index of the root node. Ignoring constant factors, the cost of searching for  $A[i]$  is the number of nodes on the path from the root  $v_r$  to  $v_i$ . Thus, the total cost of performing all the binary searches is given by the following expression:

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

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

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

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

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

<sup>3</sup>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  $v$ .

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

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

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

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

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

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

The  $\Theta(n)$  term comes from computing the total number of searches  $\sum_{i=1}^n f[i]$ .

Yeah, that's one ugly recurrence, but it's actually easier to solve than it looks. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for  $T(n-1)$  to get rid of the summation, and then regroup again.

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

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

Let me emphasize that this recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with  $n$  nodes satisfies the recurrence

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

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

## Exercises

1. (a) Let  $A[1..m]$  and  $B[1..n]$  be two arbitrary arrays. A *common subsequence* of  $A$  and  $B$  is both a subsequence of  $A$  and a subsequence of  $B$ . Give a simple recursive definition for the function  $lcs(A, B)$ , which gives the length of the *longest* common subsequence of  $A$  and  $B$ .
- (b) Let  $A[1..m]$  and  $B[1..n]$  be two arbitrary arrays. A *common supersequence* of  $A$  and  $B$  is another sequence that contains both  $A$  and  $B$  as subsequences. Give a simple recursive definition for the function  $scs(A, B)$ , which gives the length of the *shortest* common supersequence of  $A$  and  $B$ .
- (c) Call a sequence  $X[1..n]$  *oscillating* if  $X[i] < X[i + 1]$  for all even  $i$ , and  $X[i] > X[i + 1]$  for all odd  $i$ . Give a simple recursive definition for the function  $los(A)$ , which gives the length of the longest oscillating subsequence of an arbitrary array  $A$  of integers.
- (d) Give a simple recursive definition for the function  $sos(A)$ , which gives the length of the shortest oscillating supersequence of an arbitrary array  $A$  of integers.
- (e) Call a sequence  $X[1..n]$  *accelerating* if  $2 \cdot X[i] < X[i - 1] + X[i + 1]$  for all  $i$ . Give a simple recursive definition for the function  $lxs(A)$ , which gives the length of the longest accelerating subsequence of an arbitrary array  $A$  of integers.

**For more backtracking exercises, see the next two lecture notes!**