The whole is of necessity prior to the part.

Aristotle

11. Divide and Conquer

11.1. Divide and Conquer

Many recursive algorithms follow the *divide and conquer* philosophy. They **divide** the problem into smaller subproblems (of the same shape) and then **conquer** these subproblems (either because they are trivial or they are further divided). Finally the solutions of the subproblems are **combined** into a solution for the original problem.[1]

In the sequel we will demonstrate the *divide and conquer* principle by discussing *mergesort*, which works as follows: To sort a list `zs` we distinguish two cases. In the base case `zs` contains at most one element and is already sorted. In the step case we *divide* `zs` into two sublists `xs` and `ys` (such that `zs` = `xs@ys`). After sorting `xs` and `ys` we *merge* the result, such that the obtained list is also sorted. In OCaml, mergesort can be implemented as follows:

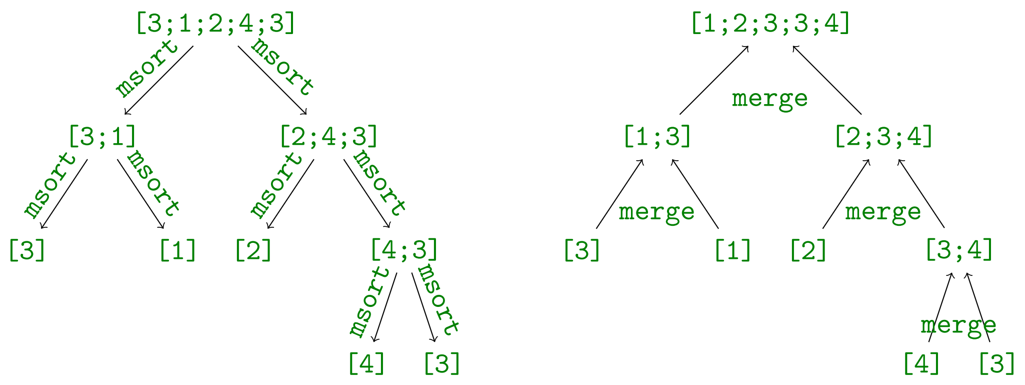
```
let rec merge xs ys = match (xs,ys) with
 | ([],ys) -> ys
 | (xs,[]) -> xs
 | (x::xs,y::ys) -> if x < y then x::(merge xs (y::ys))
                              else y::(merge (x::xs) ys)

let rec msort = function
 | [] -> []
 | [z] -> [z]
 | zs -> let (xs,ys) = Lst.split_at (Lst.length zs / 2) zs in
        merge (msort xs) (msort ys)
```

The above implementation divides the list which should be sorted (approximately) at the middle (using `split_at`) and sorts the first half and the second half separately. Finally the obtained lists are *merged* into the resulting list. The execution of `msort` can be visualized by the trees in Figure 11.1. First the recursive calls to `msort` decompose the list until all lists are singleton. This phase is top to bottom (cf. Figure 11.1(a)). In a second phase the resulting list is built by merging the already sorted ones. This phase is executed bottom to top (cf. Figure 11.1(b)).

Next we will investigate the runtime of `msort`. Note that the runtime of `msort` is mainly depending on the length of the list to sort. The actual list contents are of minor importance. Hence the runtime of `msort` is a function $T\colon \mathbb{N} \to \mathbb{N}$ where n is the length of the list zs we want to sort and $T(n)$ is the number of instructions that are executed when calling `msort` zs. To simplify matters we will consider a worst case analysis and

[1]Consequently the concept should be named *divide and conquer and combine* but since this sounds worse (and is more work to write) we prefer the shorter name.



(a) Divide (b) Combine

Figure 11.1.: Evaluation of `msort [3;1;2;4;3]`.

study the *asymptotic* runtime only.[2] We will use standard O-notation for this. Hence we are looking for a function $f\colon \mathbb{N} \to \mathbb{N}$ such that $T \in O(f)$. Sometimes we are interested in tight upper bounds on $T(n)$ and then write $T \in \Theta(f)$.

The runtime of a divide and conquer algorithm can always be computed by summing up the runtime for each of the steps to *divide* the problem, to *conquer* the subproblems, and to *combine* the solutions. We investigate each of the steps below for our implementation of `msort`:

- *divide*: We divide the original list of length n into two lists. We used the function `split_at` for this purpose. Looking at the implementation of `split_at` we see that this part can be done in $O(n)$ time. Hence the runtime of *divide* is $O(n)$.

- *conquer*: Here we must consider two cases. If we have an empty or a singleton list, then this list is already sorted and we can just return it. Hence in this case we have constant runtime, i.e., $O(1)$. In the other case we *conquer* each of the two lists (obtained from *divide*). Note that `split_at` produced two sublists, each of length (approximately) $\frac{n}{2}$. To sort one of them we need time $T(\frac{n}{2})$ and hence for both of them we obtain a runtime of $2T(\frac{n}{2})$. Hence in this case *conquer* runs in time $2T(\frac{n}{2})$.

- *combine*: Finally we need to *combine* the two sorted lists into a single sorted list (using `merge`). Since both of these lists are of length $\frac{n}{2}$ and one element of either list is removed in each recursive call, `merge` runs in time $O(n)$.

Hence for `msort` the function T can be given as follows (for some constant c):

$$T(n) = \begin{cases} c & \text{if } n \leqslant 1 \\ 2T(\frac{n}{2}) + cn & \text{otherwise} \end{cases} \tag{11.1}$$

[2]Here *asymptotic* means that we are not interested in the exact number of operations `msort` executes but in the *order* of the performed operations.

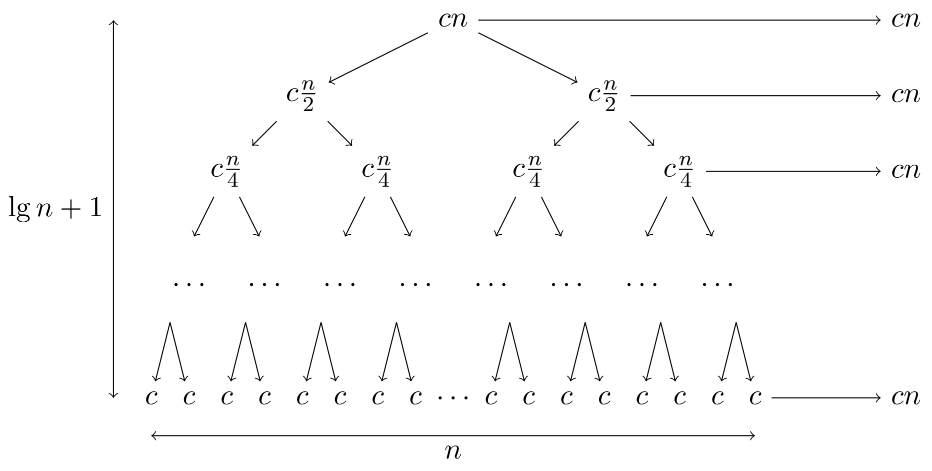


Figure 11.2.: A tree for the recurrence (11.1).

The case $n \leqslant 1$ will be called the base case (in contrast to the step case). Note that $2T(\frac{n}{2})$ is the time needed for *conquer* while cn is the accumulated time for *divide* and *combine*. Equations of the form (11.1) are called *recurrence equations* or just *recurrences*. Note that the recurrence (11.1) does not yet give a bound on the runtime of `msort`. However, we can use it to compute such a bound. We unfold recurrence (11.1) recursively and obtain a binary tree (see Figure 11.2). For ease of discussion we assume that n is a power of 2, i.e., $n = 2^k$ for some $k \in \mathbb{N}$, and hence the tree is perfect (see Section 6.1.2). Each node corresponds to a call to `msort` and we label the node by the runtime needed for the phases *divide* and *combine*. At the root node we split a list of length n into two sublists (*divide*) and then merge the sorted lists, each of length (approximately) $\frac{n}{2}$ (*combine*). Hence this node gives runtime cn. The two recursive calls (on lists of length $\frac{n}{2}$) give runtime $c\frac{n}{2}$ each, so together also yield cn. This actually holds for each *level* of the tree. Note that the bottom level of the tree has n nodes since we started with a list of length n. So how many levels are there? Or stated differently: What is the height of the tree? From Exercise 11.1 we obtain that there are $\lg n + 1$ levels and hence $T(n) \leqslant cn \times (\lg n + 1) \in O(n \lg n)$.[3]

There are several ways to solve recurrence equations, but for the ones we meet in this course unfolding the tree will suffice. In general the runtime of a divide and conquer algorithm can be given as

$$T(n) = \begin{cases} c & \text{base case} \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{step case} \end{cases} \tag{11.2}$$

Here a is the number of subproblems that must be conquered in the step case and $\frac{n}{b}$ is the size of each of these problems. Furthermore the time needed to divide ($D(n)$) the problems and to combine ($C(n)$) the solutions must be added. For `msort` above we have $a = b = 2$, but there exist many other problems where a and b are different from

[3]By $\lg n$ we abbreviate $\log_2 n$.

two (and even different from each other). Note that $D(n) + C(n)$ tells us how costly a single function call is, whereas a gives us the number of recursive function calls we have to consider. However, the problems treated recursively are smaller ($\frac{n}{b}$), which we must take into account.

To get an even simpler form of recurrence equations we often collapse the cases for dividing problems, combining solutions, and the base case. Then a recurrence equation reads as follows:

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{11.3}$$

Here $a, b \in \mathbb{N}$ and $f\colon \mathbb{N} \to \mathbb{N}$ is an asymptotically positive function. Often we do not need to solve recurrence equations on our own, but we can use the *Master Theorem*, which tells us how the solutions look like in many cases. Basically, we compare the cost for a single function call ($f(n)$) with the number of nodes in the last level of the tree ($n^{\log_b a}$, which estimates the number of recursive calls needed if $b > 1$). If the cost for a single call is significantly smaller than the number of recursive calls ($f(n) \in O(n^{\log_b a-\epsilon})$, then the latter determines the overall complexity. This corresponds to the first case in the theorem. The reasoning for the third case is analogous. In the second case the cost of one function call approximately equals the number of recursive calls and hence the reasoning is similar as in Figure 11.2, explaining the additional factor $\lg n$.

Theorem 11.1 (Master Theorem). *Let $a \geqslant 1$, $b > 1$, and $T(n)$ as in (11.3). Then*

1. *$T(n) \in \Theta(n^{\log_b a})$ if $f(n) \in O(n^{\log_b a-\epsilon})$ for some $\epsilon > 0$.*
2. *$T(n) \in \Theta(n^{\log_b a} \lg n)$ if $f(n) \in \Theta(n^{\log_b a})$.*
3. *$T(n) \in \Theta(f(n))$ if $f(n) \in \Omega(n^{\log_b a+\epsilon})$ for $\epsilon > 0$ and $af(\frac{n}{b}) \leqslant cf(n)$ for some $c < 1$ and sufficiently large n.*

Since for `msort` we have $a = b = 2$ and $f(n) \in \Theta(n)$, the Master Theorem (case 2) applies and yields $T(n) = \Theta(n \lg n)$.

We have seen that for many divide and conquer algorithms we can easily conclude their runtime by just checking if the Master Theorem applies. But this is far from being the only benefit. Another nice fact about recursive algorithms is that proving their correctness is usually easier than when considering loops since induction can be applied. In the exercises you are asked to prove the correctness of `msort` (see Exercise 11.3). Have you ever tried to prove *bubblesort* correct?

11.2. Dynamic Programming

Dynamic programming is a technique that prefers *recalling* over *recomputing*. To this end it stores the results of subproblems and just looks the result up instead of recomputing it. Clearly, storing the results for subproblems will need additional memory. Not all divide and conquer problems are equally well-suited for dynamic programming. Those where it is necessary to solve many (identical) subproblems frequently will benefit more than problems where all subproblems look (fairly) different. Another important issue is that the *lookup* of a result should be drastically cheaper than the *recomputation* of a result. In practice one typically uses data structures that allow a lookup in constant or logarithmic time. In the imperative world this typically holds for hash tables (lookup is possible in constant time if there is at most one element in each bucket) whereas in the functional setting often binary search trees (lookup is possible in logarithmic time if the tree is balanced) are used. Since theory is pretty independent from such implementation matters we will refer to it in the sequel just as *(lookup) table* and assume that operations such as insertion (`add`) and lookup (`find`) are sufficiently efficient. We will use the interface to a lookup table shown in Listing 11.1.

```
type ('a,'b) t
val empty: ('a,'b) t
val mem: ('a,'b) t -> 'a -> bool
val find : ('a,'b) t -> 'a -> 'b
val add : ('a,'b) t -> 'a -> 'b -> ('a,'b) t
```

Listing 11.1: `Lookup.mli`

The type of a lookup table is `('a,'b) t` where `'a` is the type for the *keys* and `'b` the type for the *values* associated with the keys. The constant `empty` returns an empty table. The function `mem t k` checks if table `t` contains a binding for the key `k`. If so, then `find t k` returns the value `v` associated to `k` in table `t`. Finally, `add t k v` adds a new binding `(k,v)` to table `t` and returns the new table.

11.2.1. Fibonacci Numbers

We have already seen that the straightforward implementation of the Fibonacci function is inefficient, since `fib n` results in approximately 2^n recursive calls. To overcome this exponential growth we will dynamically program (hence the name) a table, where for each m (here $1 \leqslant m \leqslant n$) we have two cases. If we did not yet consider m we compute `fib` m and add as a binding for m the value of `fib` m to the table. If m has already been considered, then we just look up the binding, i.e., `fib` m, in the table. To store the table linear space is needed but now it is possible to compute `fib` m in linear time. An implementation of the Fibonacci numbers using dynamic programming can be done as follows:

```
let fib_dp n =
 let rec fib t n =
  if Lookup.mem t n then t
  else if n < 2 then Lookup.add t n 1
  else
   let t = fib t (n-1) in
   let t = fib t (n-2) in
   let r = Lookup.find t (n-1) + Lookup.find t (n-2) in
   let t = Lookup.add t n r in
   t
 in Lookup.find (fib Lookup.empty n) n
```

The differences to the original implementation are as follows: The (inner) `fib` function now has an additional parameter `t`, which is a lookup table that will finally contain all Fibonacci numbers as bindings. Hence `fib` now returns a lookup table instead of a single Fibonacci number. The first thing `fib` checks is if it has already computed the Fibonacci number for the current parameter n. In this case the binding is already in the table and it can be returned without changes. If there is no binding yet for n then we have to add it. This is easy in the base case ($n < 2$). In the step case we first get the (updated) tables for the recursive calls to $n-1$ and $n-2$. Now the table contains bindings for the $(n-1)$-st and the $(n-2)$-nd Fibonacci numbers which we can just lookup in the table to compute the n-th Fibonacci number, which we finally add to the table before we return it. Since `fib Lookup.empty n` returns a table containing the first n Fibonacci numbers the last line then looks up the n-th Fibonacci number.

We have seen that dynamic programming allows to reduce exponential runtime to polynomial runtime while the additional memory needed is only linear (in the size of the input).

11.2.2. Optimal Rod Cutting

Another example demonstrating the benefit of dynamic programming is the *Optimal Rod Cutting* problem. This problem comes as an *optimization* problem. We are given a rod of length n and a table of prices p_i for $1 \leqslant i \leqslant n$ where p_i is the price for a rod of length i. The question is to maximize the profit when cutting a rod of length n into pieces. Consider the following example.

Example 11.1. Given the following table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	2	3	5	5	8	12	12	15	15	17

What is the maximum price that can be obtained when cutting a rod of length 10? Later on we will see that the answer is 20 for this instance. Note that for simplicity we are currently not interested in *how to cut the rods* to obtain this answer.

We will solve the optimal rod cutting problem recursively. Let r_i (for $1 \leqslant i \leqslant n$) be the optimal solution for a rod of length i. Then

$$r_n = \max_{1 \leqslant i < n} \{p_n, p_i + r_{n-i}\} \tag{11.4}$$

This formula says that for an optimal solution we either do not cut at all (p_n) or take the maximum sum of p_i (which is not cut further) and r_{n-i} (which is an optimal solution for a smaller problem). We can implement the above formula as follows:[4]

```
let price ps i = if i <= Lst.length ps then Lst.nth ps (i-1) else 0

let rec cut ps n =
 if n = 0 then 0
```

[4]Note that the original problem is restricted to rods for which a price is specified. The implementation is slightly more general by assigning a price of zero to rods which are too long, i.e., ones that we cannot sell.

```
  else
   let f i q = max q (price ps i + cut ps (n-i)) in
   Lst.foldr f (price ps n) (IntLst.range 1 n)

let ps = [2;3;5;5;8;12;12;15;15;17]
```

Here `price ps i` returns the price for a rod of length `i` and `cut` implements the formula (11.4) for a price list `ps`. Now `cut ps 10` = `20` can be computed in almost no time but the call `cut ps 30` already seems to take forever. The reason is that the same problems are solved again and again. We can remove the bottleneck similar as for the Fibonacci numbers by dynamic programming. The result looks as follows:

```
let cut_dp ps n =
 let rec cut t ps n =
  if Lookup.mem t n then t
  else if n = 0 then Lookup.add t n 0
  else
   let t = Lst.foldr (fun i t -> cut t ps (n-i)) t (IntLst.range 1 n) in
   let f i q = max q (price ps i + Lookup.find t (n-i)) in
   let r = Lst.foldr f (price ps n) (IntLst.range 1 n) in
   Lookup.add t n r
in Lookup.find (cut Lookup.empty ps n) n
```

The (inner) `cut` function is almost the same as before. The main difference is that we first update the lookup table before computing the optimal value r for a rod of length n.

We observe that `rod_dp 30` = `60` is now computed almost instantaneously. While `rod_dp` is significantly faster than `rod` it might not be satisfactory because an optimal solution does not yet tell us *how* the rod should be cut. Adding this information is the task of Exercise 11.10.

11.2.3. Beans and Bowls

Consider a bowl containing black and white beans. We may replace beans by the following laws:

1. Replace two black beans by a single white bean.
2. Replace two white beans by a single black bean.
3. Replace a black and a white bean by a single white bean.

These laws can be written more concisely as *rewrite rules*[5]

$$\bullet\bullet \longrightarrow \circ \qquad \circ\circ \longrightarrow \bullet \qquad \bullet\circ \longrightarrow \circ \qquad \circ\bullet \longrightarrow \circ$$

Now we face the following question: Starting with a bowl containing 150 black and 75 white beans can the color of the last bean be predicted? In other words the question is if we always end up with either a white or a black bean (independent from *how* we replace the beans).

[5]Here we use two rewrite rules for the last law to show that the order of the beans does not matter.

First we tackle the problem with a naive algorithm that just tries all possibilities. We write a function `beans : int -> int -> (bool * bool)` which takes two integers (number of black and white beans, respectively) and returns a pair of booleans. Here the first component is true if and only if it is possible to end up with a black bean and the second component indicates if we can end up with a white bean (it might be that both possibilities come true). The function can then be implemented as follows:

```
let rec beans b w =
 if b = 1 && w = 0 then (true,false)
 else if b = 0 && w = 1 then (false,true)
 else if b < 0 || w < 0 then (false, false)
 else
  (beans (b-2) (w+1)) <||>
  (beans (b+1) (w-2)) <||>
  (beans (b-1) w)
```

Here the operator `<||>` is a logical or on pairs, defined as

```
let (<||>) (b1,w1) (b2,w2) = (b1 || b2,w1 || w2)
```

However, while e.g., `beans 5 5` shows that for some starting configurations it is possible to end up with a black or a white bean the call `beans 150 75` does not terminate within reasonable time. The reason is that similar instances of the problem are considered over and over again (see Exercise 11.11). To overcome this bottleneck we use dynamic programming. In the function `beans_dp` we first fill the table (with the inner `beans` function) and then we can access the result for a concrete configuration.

```
module L = Lookup

let beans_dp b w =
 let rec beans t b w =
  if L.mem t (b,w) then t else
  if b = 1 && w = 0 then L.add t (b,w) (true,false)
  else if b = 0 && w = 1 then L.add t (b,w) (false,true)
  else if b < 0 || w < 0 then L.add t (b,w) (false, false)
  else
   let t = beans t (b-2) (w+1) in
   let t = beans t (b+1) (w-2) in
   let t = beans t (b-1) w in
   let r = L.find t (b-2,w+1) <||> L.find t (b+1,w-2) <||> L.find t (b-1,w) in
   L.add t (b,w) r
 in L.find (beans L.empty b w) (b,w)
```

And indeed, if we start with 150 black and 75 white beans the outcome can either be a black or a white bean. Note that the call `beans_dp 150 75` does not return immediately (On a contemporary laptop it takes about a second). To understand this (shockingly slow) execution we first take a look at the search space. The call `beans_dp 150 75` requires 19,138 entries in the lookup table and from Exercise 11.13 we obtain that in general the lookup table (computed for the call `beans_dp` m n) is quadratic in $m + n$. Hence the runtime of `beans_dp` will be (at least) quadratic. The overall runtime also depends on the implementation of the `Lookup` module. Ours is based on binary search trees, where `mem` and `find` have logarithmic runtime (if the tree is balanced, otherwise the runtime is linear). Hence the overall runtime of `beans_dp` will be between $O(n^2 \cdot \log n)$ and $O(n^3)$.

11.3. Chapter Notes

This chapter builds on Chapter 4 (divide and conquer) and Chapter 15 (dynamic programming) from [4].

While *divide and conquer* techniques are not restricted to functional programming they often appear in this programming paradigm due to the heavy use of recursion. The principle itself is much older, however. Already in 1805 Carl Friedrich Gauss presented a fast Fourier transformation algorithm where the problem is divided into smaller subproblems whose solutions are combined. Recurrence equations have already been studied by Fibonacci in the 13th century.

While *dynamic programming* is a simple idea itself, it took until the 1950's to properly study the underlying mathematics. The idea of dynamic programming sometimes is also referred to by the name *memoization*. Note that some other (functional) programming languages have memoization already built in (e.g. Haskell).

There are many other real world problems where an efficient implementation is possible using dynamic programming. We mention an important one which is finding longest common subsequences of two strings. This is used in DNA analysis to determine the similarity of two genes. Other problems that benefit from dynamic programming are discussed in the exercises.

11.4. Exercises

Exercise 11.1. Consider a perfect binary tree where the last level has n nodes. Show that the tree has $\lg n + 1$ levels, i.e., height $\lg n + 1$.

Hint: How many nodes does the tree have? Lemma 6.5 might be helpful.

Exercise 11.2. Prove the following claim by induction:

If xs and ys are sorted lists then `merge` xs ys is a sorted list.

Hint: Which kind of induction is useful?

Exercise 11.3. Show by structural induction on lists that for all lists zs the result of `msort` zs is a sorted list.

Hint: You can assume the claim in Exercise 11.2.

Exercise 11.4. Consider a different implementation of `msort` where the list `zs` is split differently, i.e., `(xs,ys) = (Lst.hd zs, Lst.tl zs)`. Is the (worst case) runtime affected by this change?

Exercise 11.5. Write a function `qsort`, which implements *quicksort*.

Hint: For a non-empty list select the head element as pivot.

Exercise 11.6. Consider the Fibonacci numbers (see Definition 7.1).

1. Compute a recurrence equation for the Fibonacci numbers.
2. Does the Master Theorem apply to the recurrence from item 1?

Exercise 11.7. Consider the following implementation of *insert* sort.

```
let rec insert x = function
 | [] -> [x]
 | y::ys -> if x < y then x::y::ys else y::(insert x ys)

let rec isort = function
 | [] -> []
 | x::xs -> insert x (isort xs)
```

1. Compute the recurrence equation for `isort`.
2. Solve the recurrence by unfolding it into a tree. Conclude an upper bound for the runtime of `isort`.
3. Does the Master Theorem apply to the recurrence from item 1?

Exercise 11.8. Give a recurrence where the Master Theorem does not apply.
Hint: Find (un)suitable values for a, b, and $f(n)$.

Exercise 11.9. Consider the Optimal Rod Cutting problem. A *greedy* strategy chooses a j such that $p_j + p_{n-j}$ is maximal. Then it performs the recursive calls on the shorter rods (of lengths j and $n-j$) such as in the following function:

```
let rec cut_greedy ps n =
 if n = 0 then 0
 else
  let f i (l,q) =
   let p = price ps i + price ps (n-i) in
   if p > q then (i,p) else (l,q)
  in
  let (j,_) = Lst.foldr f (n,price ps n) (IntLst.range 1 n) in
  if j = n then price ps n else cut_greedy ps j + cut_greedy ps (n-j)
```

1. Show that the greedy strategy does not necessarily yield an optimal solution.
2. What is the runtime of `cut_greedy`?

Exercise 11.10. Extend the function `cut_rod` such that it also returns *how* an optimal solution can be obtained by indicating how the rod must be cut.
Hint: Add this information to the lookup table (it might now contain triples (r_n, l, r) where r_n is the optimal solution to a rod of length n and l and r are the lengths of the left and right rod, respectively).

Exercise 11.11. Consider beans & bowls.

1. Give an upper bound on the number of recursive calls emerging from `beans` m n (in terms of m and n).
2. Draw the recursive calls emerging from `beans 3 3` as a tree. The nodes are labeled `beans` m n for different values of m and n and there is an edge from node `beans` m n to node `beans` m' n' if the former recursively calls the latter.

   Hint: Share identical nodes in the tree.
3. Use the tree from (2) to compute the number of (recursive) calls to `beans` (starting from `beans 3 3`). Check your computation by adding a counter to `beans`.
4. Use the tree from (2) to compute the number of (recursive) calls to `beans` (starting from `beans_dp 3 3`). Check your computation by adding a counter to `beans` (inside `beans_dp`).

Exercise 11.12. Extend `beans_dp` such that it returns the sequence of choices that yield a single black/white bean.

Exercise 11.13. Give an upper bound (in terms of m and n) on the size of the lookup table generated for the call `beans_dp` m n. Conclude that the size of the lookup table is quadratic in $m + n$. Compare your upper bound with the exact size of the lookup table for $m = 150$ and $n = 75$.
Hint: Note that $m \times n$ is not a correct upper bound. Why?

Exercise 11.14. Consider Post's Correspondence Problem (PCP). Here PCPs are given as a list of pairs where each pair contains the corresponding words, i.e., the words at the same indices.

```
let pcp0 = [([0;1],[0]);([0],[1;0])]
let pcp1 = [([0;1],[0]);([1;1],[1;0])] (*no solution*)
let pcp2 = [([0;1],[0]);([0;1],[1;1;0]);([0],[1;0;0]);([1;0;0],[1;0])]
let pcp3 = [([1;0;1],[1]);([0],[0;1]);([1;0],[1;1;0]);([1;1],[1;0])]
let pcp4 = [([0;1],[0]);([0;0],[1;0])] (*no solution*)
```

The following implementation tries to determine if a PCP has a solution or not by testing all possibilities in a breadth first search. To save memory, common prefixes of two words are removed using the function `trim`.

```
let rec trim = function
 | (x::xs,y::ys) when x = y -> trim (xs,ys)
 | d -> d

let extend (w1,w2) (d1,d2) = trim (w1@d1,w2@d2)

let solve ds =
 let rec solve = function
  | [] -> false
  | ([],[])::_ -> true
  | (x::_,y::_)::ws when x <> y -> solve ws
  | w::ws -> solve (ws@Lst.map (extend w) ds)
 in solve (Lst.map trim ds)
```

1. Write a function `solve_dp`, which uses dynamic programming to avoid considering the same problems again.
2. Can you give an upper bound on the additional memory needed for the lookup table?
3. Write a function `solve_dps`, which returns (the indices of) a solution.