```python
def findMissing(a):
    if a[0] != 0:
        return 0
    elif a[-1] == len(a) -1:
        return len(a)
    else :
        start = 0
        stop = len(a) - 1
        assert a[start] == start and a[stop] == stop + 1
        while stop > start + 1 :
            mid = (stop + start) / 2
            if a[mid] == mid :
                start = mid
            else :
                assert a[mid] == mid + 1
                stop = mid
        return stop
```

```
start = 0
stop = len(a) - 1
while stop > start + 1 :
    mid = (stop + start) / 2
    if a[mid] == mid :
        start = mid
    else :
        stop = mid
```

After i iterations,

(a) $a[start] = start$

(b) $a[stop] = stop + 1$

(c) $stop - start = \frac{n}{2^i}$

Initialization. After 0 iterations, (a) and (b) are true by the conditions of the outer if/else chain. Moreover,

$$stop - start = n = \frac{n}{1} = \frac{n}{2^0} = \frac{n}{2^i}$$

Maintenance. Suppose the invariant holds after i iterations for some $i \geq 0$. By the precondition of the function, $a[mid] = mid$ or $a[mid] = mid + 1$. Either way, the change to $start$ and $stop$ preserves the invariant.
Moreover,

$$mid - start = \frac{start + stop}{2} - start = \frac{stop - start}{2} = \frac{\frac{n}{2^i}}{2} = \frac{n}{2^{i+1}}$$

```
start = 0
stop = len(a) - 1
while stop > start + 1 :
    mid = (stop + start) / 2
    if a[mid] == mid :
        start = mid
    else :
        stop = mid
```

After i iterations,

(a) $a[start] = start$

(b) $a[stop] = stop + 1$

(c) $stop - start = \frac{n}{2^i}$

Termination. (*Informally*) The size of the range $[start, stop)$ decreases by half each time, so after $\lg n$ iterations the range has size one and the loop stops.

(*Formally*) After $\lg n$ iterations, $stop - start = \frac{n}{2^i} = \frac{n}{2^{\lg n}} = \frac{n}{n} = 1$, and the guard failes.

After the loop terminates, $stop = start + 1$. The loop invariant indicates that $a[start] = start$ but $a[stop] = stop + 1$. Hence $stop$ is the correct result. $\square$

You are playing a computer game in which the hero must pass through a series of rooms and halls collecting treasure. There are $2n$ rooms (in pairs) and $n-1$ halls interspersed between the pairs. Each room has a one-way door to the next hall, and each hall has two one-way doors to the rooms of the next pair. The hero must, therefore, pass through exactly one room in each pair.

Each room has a certain amount of treasure, $T_{i,j}$. Halls do not have treasure, but they each have a guardian who demands payment to let the hero cross diagonally through the hall. So, to move from $T_{i-1,0}$ to $T_{i,0}$ is free, but to move from $T_{i-1,0}$ to $T_{i,1}$ costs P_i.

Devise and implement an algorithm to find the route that yields the most treasure. Analyze its efficiency.

10		5	3
	6		
20		4	2
	8		
9		7	1
	3		
5		12	0

Let

- $T_{i,j}$ be the amount of treasure in room i, j. (Given)
- P_i be the penalty for crossing the hall between the ith and $i+1$st pair of rooms. (Given)
- $C_{i,j}$ be the most treasure than can be obtained on any route ending at room i, j. ("Scratch work")
- $D_{i,j}$ be the direction the hero should come from in order to get to room i, j with the most treasure. ("Scratch work")
- R be the route the hero should take, as a list indicating which side of the hall the hero should be on. (Solution to be returned)

Throughout, variable i ranges over $[0, n)$ and j ranges over $[0, 2)$.

$$C_{i,j} = \begin{cases} T_{i,j} & \text{if } i = 0 \\ T_{i,j} + \max(C_{i-1,j}, C_{i-1,j+1\%2} - P_{i-1}) & \text{otherwise} \end{cases}$$

DP goals in CSCI 345:

- Know what DP is and to what sort of problems it applies
- Be able to code up a table-populating algorithm when the recursive characterization is given to you.

DP goals in CSCI 445:

- Be able to derive a recursive characterization to a given problem.
- Be able to code up a table-populating algorithm *and* an algorithm to reconstruct the optimal solution using the recursive characterization you have derived.

The rod-cutting problem (CLRS pg 360):

Given a table of prices for rods of different lengths and a rod (that is, a length), what is the most valuable way to cut up the rod into smaller rods?

Problem instance in the book:

length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	30
density	1	2.5	2.66	2.25	2	2.83	2.43	2.5	2.66	3

Problem instance changed slightly:

length	1	2	3	4	5	6	7	8	9	10
price	1	5	8	9	10	17	17	20	24	29
density	1	2.5	2.66	2.25	2	2.83	2.43	2.5	2.66	2.9

Consider a given rod of length 14. How should we cut it?

Using the greedy strategy (price-densest first), we would do

$$\begin{array}{ccccccc} 10 & & 3 & & 1 & & \\ 29 & + & 8 & + & 1 & = & 38 \end{array}$$

But a better cutting is

$$\begin{array}{ccccccc} 6 & & 6 & & 2 & & \\ 17 & + & 17 & + & 5 & = & 39 \end{array}$$

Representation of the problem, and of an instance of the problem:

- n is the rod length. (Given)
- p is an array of prices, p_i (or $p[i]$) the price for a rod of length i. (Given)
- $i_1, i_2, \ldots i_k$ is a way to cut up the rod, where
  - k is the number of pieces the rod is cut into.
  - i_ℓ is the length of a piece, where $1 \leq \ell \leq k$
  - $i_1 + i_2 + \cdots + i_k = n$
  - $1 \leq k \leq n$
  - $k = 1$ indicates no cuts at all
  - $k = n$ indicates cutting the rod into n pieces of unit length

  In the previous example, $i_1 = 6, i_2 = 6, i_3 = 2$.
- r_n is the (best?) revenue for cutting a rod of length n, is calculated as

$$r_n = \sum_{\ell=1}^{k} p[i[\ell]] = \sum_{\ell=1}^{k} p_{i_\ell}$$

- The solution is an array i of length k that maximizes r. (Solution to be returned)

An alternate formulation/representation is based on the position of cuts relative to the end of the original rod.

$i_1 = 6$	$i_2 = 6$	$i_3 = 2$	
0	6	12	$n = 14$
j_0	j_1	j_2	j_3

$$j_\ell = \sum_{m=1}^{\ell} i_m = j_{\ell-1} + i_\ell$$

From pg 362: We characterize the optimal substructure as

$$
\begin{aligned}
r_n = \max(\quad & p_n \\
& r_1 + r_{n-1} \\
& r_2 + r_{n-1} \\
& \vdots \\
& r_x + r_{n-x} \\
& \vdots \\
& r_{n-1} + r_1)
\end{aligned}
$$

From pg 363: The naïve recursive version and why it's bad.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n$$

Verifying this using the substitution method (see Ex 15.1-1):

$$\begin{aligned}
T(n) &= 1 + \sum_{j=0}^{n-1} 2^j \\
&= 1 + 1 + 2 + 4 + 8 + \cdots + 2^{n-2} + 2^{n-1} \\
&= T(n-1) + 2^{n-1} \\
&= 2^{n-1} + 2^{n-1} \\
&= 2 \cdot 2^{n-1} \\
&= 2^n
\end{aligned}$$

Why dynamic programming:

- Dynamic programming applies to optimization problems that have overlapping subproblems.
- Dynamic programming avoid the bad running time of brute-force ("naïvely recursive") solutions by recording previously computed results in a table (*memoization*)

The anatomy of the dynamic programming approach from the programmer's perspective (compare CLRS pg 359):

- *Characterize the substructure*: Determine what the subproblems are and how they relate to the larger problem. (Determine the meaning of the tables.)
- Recursively define the problem.
- Devise an algorithm to populate the tables of subproblem solutions. (Find *how good* the best way is.)
- Devise an algorithms to reconstruct a solution from the tables. (Find *the best way*.)