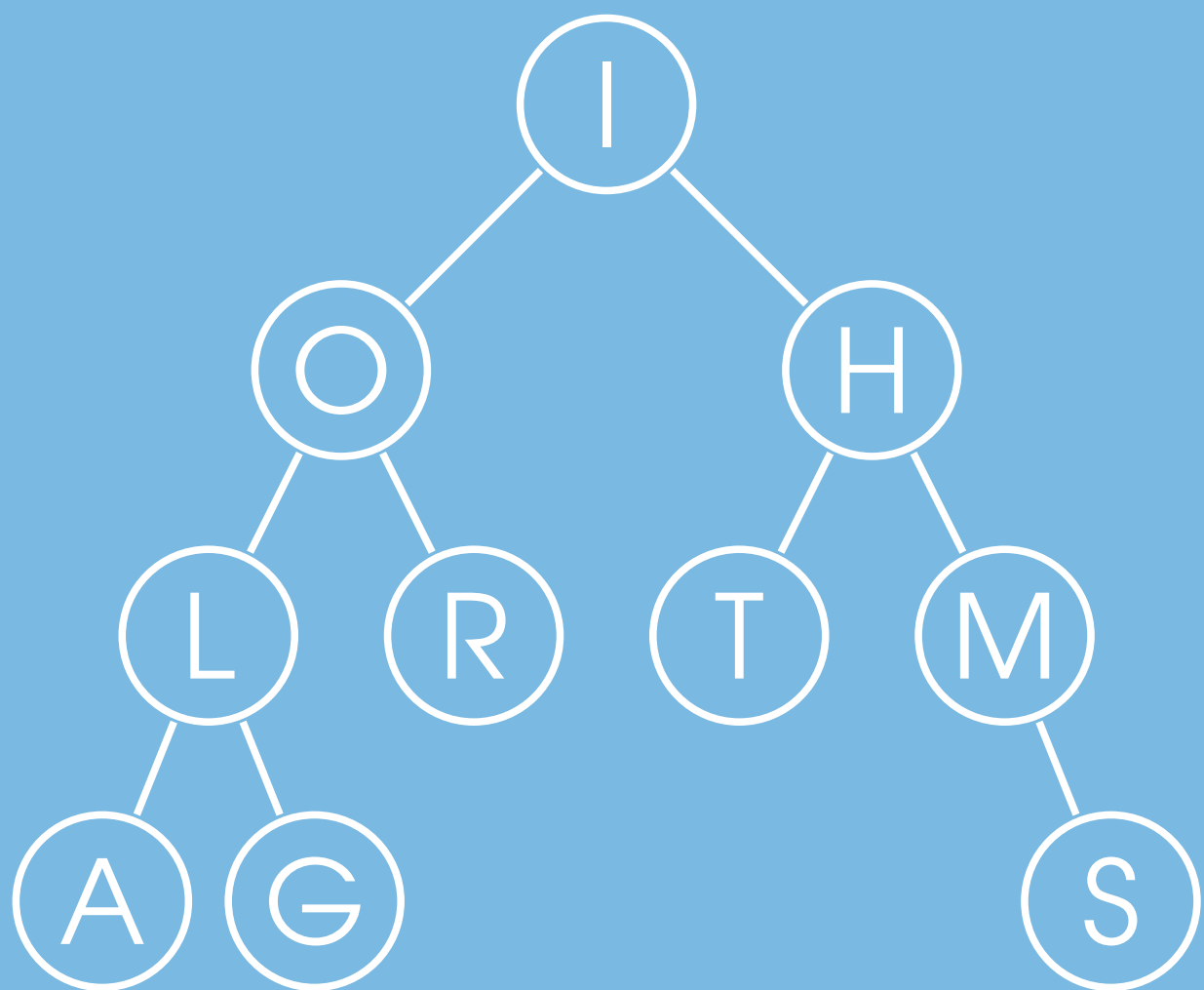


LEARNING ALGORITHMS THROUGH PROGRAMMING AND PUZZLE SOLVING



by Alexander Kulikov and Pavel Pevzner

Contents

- About This Book ix
- Programming Challenges xii
- Interactive Algorithmic Puzzles xix
- What Lies Ahead xxii
- Meet the Authors xxiii
- Meet Our Online Co-Instructors xxiv
- Acknowledgments xxv

- 1 Algorithms and Complexity 1**
 - 1.1 What Is an Algorithm? 1
 - 1.2 Pseudocode 1
 - 1.3 Problem Versus Problem Instance 1
 - 1.4 Correct Versus Incorrect Algorithms 3
 - 1.5 Fast Versus Slow Algorithms 4
 - 1.6 Big-O Notation 6

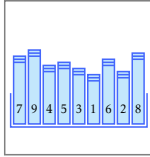
- 2 Algorithm Design Techniques 9**
 - 2.1 Exhaustive Search Algorithms 9
 - 2.2 Branch-and-Bound Algorithms 10
 - 2.3 Greedy Algorithms 10
 - 2.4 Dynamic Programming Algorithms 10
 - 2.5 Recursive Algorithms 14
 - 2.6 Divide-and-Conquer Algorithms 20
 - 2.7 Randomized Algorithms 22

- 3 Programming Challenges 27**
 - 3.1 Sum of Two Digits 28
 - 3.2 Maximum Pairwise Product 31
 - 3.2.1 Naive Algorithm 32
 - 3.2.2 Fast Algorithm 36

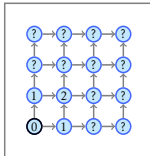
3.2.3	Testing and Debugging	37
3.2.4	Can You Tell Me What Error Have I Made?	39
3.2.5	Stress Testing	39
3.2.6	Even Faster Algorithm	43
3.2.7	A More Compact Algorithm	44
3.3	Solving a Programming Challenge in Five Easy Steps	44
3.3.1	Reading Problem Statement	44
3.3.2	Designing an Algorithm	45
3.3.3	Implementing an Algorithm	45
3.3.4	Testing and Debugging	46
3.3.5	Submitting to the Grading System	47
4	Good Programming Practices	49
4.1	Language Independent	49
4.1.1	Code Format	49
4.1.2	Code Structure	49
4.1.3	Names and Comments	51
4.1.4	Debugging	52
4.1.5	Integers and Floating Point Numbers	54
4.1.6	Strings	56
4.1.7	Ranges	56
4.2	C++ Specific	58
4.2.1	Code Format	58
4.2.2	Code Structure	58
4.2.3	Types and Constants	61
4.2.4	Classes	62
4.2.5	Containers	64
4.2.6	Integers and Floating Point Numbers	65
4.3	Python Specific	66
4.3.1	General	66
4.3.2	Code Structure	68
4.3.3	Functions	70
4.3.4	Strings	71
4.3.5	Classes	72
4.3.6	Exceptions	73

5	Algorithmic Warm Up	75
5.1	Fibonacci Number	77
5.2	Last Digit of Fibonacci Number	79
5.3	Greatest Common Divisor	81
5.4	Least Common Multiple	82
5.5	Fibonacci Number Again	83
5.6	Last Digit of the Sum of Fibonacci Numbers	85
	Solution 1: Pisano Period	86
	Solution 2: Fast Matrix Exponentiation	89
	Python Code	91
5.7	Last Digit of the Sum of Fibonacci Numbers Again	93
5.8	Last Digit of the Sum of Squares of Fibonacci Numbers	94
6	Greedy Algorithms	97
6.1	Money Change	99
	Solution: Use Largest Denomination First	100
	Python Code	101
6.2	Maximum Value of the Loot	103
6.3	Car Fueling	105
6.4	Maximum Advertisement Revenue	107
6.5	Collecting Signatures	109
	Solution: Cover Segments with Minimum Right End First	110
	Python Code	112
6.6	Maximum Number of Prizes	114
6.7	Maximum Salary	116
7	Divide-and-Conquer	119
7.1	Binary Search	121
7.2	Majority Element	124
7.3	Improving QUICKSORT	126
7.4	Number of Inversions	127
7.5	Organizing a Lottery	129
	Solution 1: Sorting All Points	130
	Solution 2: Binary Search	132
	Python Code	133
7.6	Closest Points	135

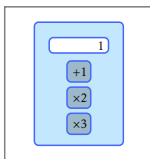
8	Dynamic Programming	141
8.1	Money Change Again	144
8.2	Primitive Calculator	145
8.3	Edit Distance	147
8.4	Longest Common Subsequence of Two Sequences	149
8.5	Longest Common Subsequence of Three Sequences	151
8.6	Maximum Amount of Gold	153
	Solution 1: Analyzing the Structure of a Solution	154
	Solution 2: Analyzing All Subsets of Bars	156
	Solution 3: Memoization	158
	Python Code	159
8.7	Partitioning Souvenirs	162
8.8	Maximum Value of an Arithmetic Expression	164
	Appendix	165
	Compiler Flags	165
	Frequently Asked Questions	166



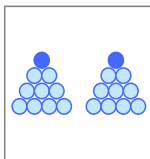
Book Sorting. Rearrange books on the shelf (in the increasing order of heights) using minimum number of swaps.



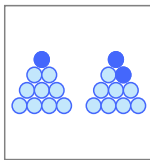
Number of Paths. Find out how many paths are there to get from the bottom left circle to any other circle and place this number inside the corresponding circle.



Antique Calculator. Find the minimum number of operations needed to get a positive integer n from the integer 1 using only three operations: add 1, multiply by 2, or multiply by 3.



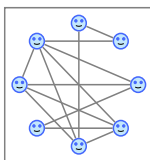
Two Rocks Game. There are two piles of ten rocks. In each turn, you and your opponent may either take one rock from a single pile, or one rock from both piles. Your opponent moves first and the player that takes the last rock wins the game. Design a winning strategy.



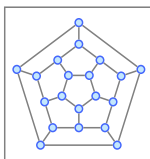
Three Rocks Game. There are two piles of ten rocks. In each turn, you and your opponent may take up to three rocks. Your opponent moves first and the player that takes the last rock wins the game. Design a winning strategy.



Map Coloring. Use minimum number of colors such that neighboring countries are assigned different colors and each country is assigned a single color.



Clique Finding. Find the largest group of mutual friends (each pair of friends is represented by an edge).



Icosian Game. Find a cycle visiting each node exactly once.

```
FASTROCKS( $n, m$ ):  
if  $n$  and  $m$  are both even:  
    return  $L$   
else:  
    return  $W$ 
```

However, though FASTROCKS is more efficient than ROCKS, it may be difficult to modify it for similar games, for example, a game in which each player can move up to three rocks at a time from the piles. This is one example where the slower algorithm is more instructive than a faster one.

Exercise Break. Play the Three Rocks game using our interactive puzzle and construct the dynamic programming table similar to the table above for this game.

2.5 Recursive Algorithms

Recursion is one of the most ubiquitous algorithmic concepts. Simply, an algorithm is recursive if it calls itself.

The *Towers of Hanoi puzzle* consists of three pegs, which we label from left to right as 1, 2, and 3, and a number of disks of decreasing radius, each with a hole in the center. The disks are initially stacked on the left peg (peg 1) so that smaller disks are on top of larger ones. The game is played by moving one disk at a time between pegs. You are only allowed to place smaller disks on top of larger ones, and any disk may go onto an empty peg. The puzzle is solved when all of the disks have been moved from peg 1 to peg 3. Try our interactive puzzle Hanoi Towers to figure out how to move all disks from one peg to another.

Towers of Hanoi Problem

Output a list of moves that solves the Towers of Hanoi.

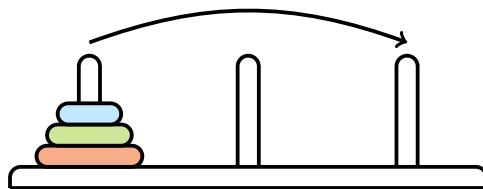
Input: An integer n .

Output: A sequence of moves that solve the n -disk Towers of Hanoi puzzle.

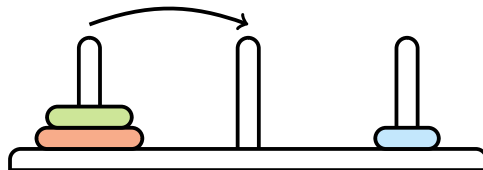
Solving the puzzle with one disk is easy: move the disk to the right peg. The two-disk puzzle is not much harder: move the small disk to the middle peg, then the large disk to the right peg, then the small disk to the right peg to rest on top of the large disk.

The three-disk puzzle is somewhat harder, but the following sequence of seven moves solves it:

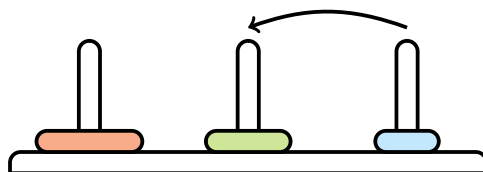
1. Move disk from peg 1 to peg 3



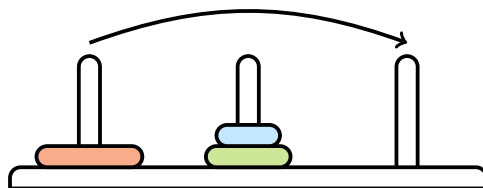
2. Move disk from peg 1 to peg 2



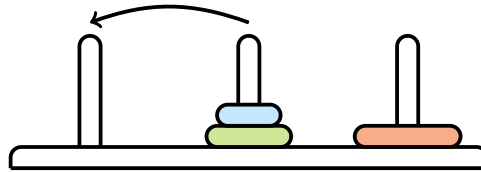
3. Move disk from peg 3 to peg 2



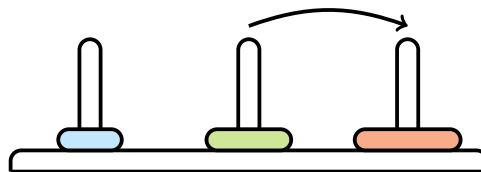
4. Move disk from peg 1 to peg 3



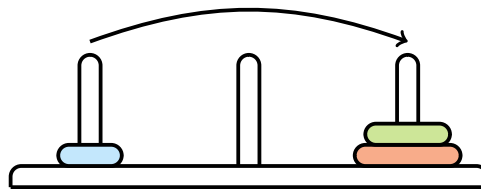
5. Move disk from peg 2 to peg 1



6. Move disk from peg 2 to peg 3



7. Move disk from peg 1 to peg 3



Now we will figure out how many steps are required to solve a four-disk puzzle. You cannot complete this game without moving the largest disk. However, in order to move the largest disk, we first had to move all the smaller disks to an empty peg. If we had four disks instead of three, then we would first have to move the top three to an empty peg (7 moves), then move the largest disk (1 move), then again move the three disks from their temporary peg to rest on top of the largest disk (another 7 moves). The whole procedure will take $7 + 1 + 7 = 15$ moves.

More generally, to move a stack of size n from the left to the right peg, you first need to move a stack of size $n - 1$ from the left to the middle peg, and then from the middle peg to the right peg once you have moved the n -th disk to the right peg. To move a stack of size $n - 1$ from the middle to the right, you first need to move a stack of size $n - 2$ from the middle to the left, then move the $(n - 1)$ -th disk to the right, and then move the stack of size $n - 2$ from the left to the right peg, and so on.

At first glance, the Towers of Hanoi Problem looks difficult. However, the following *recursive algorithm* solves the Towers of Hanoi Problem with just 9 lines!

```
HANOITOWERS( $n$ ,  $fromPeg$ ,  $toPeg$ )
if  $n = 1$ :
    output "Move disk from peg  $fromPeg$  to peg  $toPeg$ "
    return
 $unusedPeg \leftarrow 6 - fromPeg - toPeg$ 
HANOITOWERS( $n - 1$ ,  $fromPeg$ ,  $unusedPeg$ )
output "Move disk from peg  $fromPeg$  to peg  $toPeg$ "
HANOITOWERS( $n - 1$ ,  $unusedPeg$ ,  $toPeg$ )
```

The variables $fromPeg$, $toPeg$, and $unusedPeg$ refer to the three different pegs so that $HANOITOWERS(n, 1, 3)$ moves n disks from the first peg to the third peg. The variable $unusedPeg$ represents which of the three pegs can serve as a temporary destination for the first $n - 1$ disks. Note that $fromPeg + toPeg + unusedPeg$ is always equal to $1 + 2 + 3 = 6$, so the value of the variable $unusedPeg$ can be computed as $6 - fromPeg - toPeg$. Table below shows the result of $6 - fromPeg - toPeg$ for all possible values of $fromPeg$ and $toPeg$.

$fromPeg$	$toPeg$	$unusedPeg$
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

After computing $unusedPeg$ as $6 - fromPeg - toPeg$, the statements

```
HANOITOWERS( $n - 1$ ,  $fromPeg$ ,  $unusedPeg$ )
output "Move disk from peg  $fromPeg$  to peg  $toPeg$ "
HANOITOWERS( $n - 1$ ,  $unusedPeg$ ,  $toPeg$ )
```

solve the smaller problem of moving the stack of size $n - 1$ first to the temporary space, moving the largest disk, and then moving the $n - 1$ remaining disks to the final destination. Note that we do not have to specify

which disk the player should move from *fromPeg* to *toPeg*: it is always the top disk currently residing on *fromPeg* that gets moved.

Although the Hanoi Tower solution can be expressed in just 9 lines of pseudocode, it requires a surprisingly long time to run. To solve a five-disk tower requires 31 moves, but to solve a hundred-disk tower would require more moves than there are atoms on Earth. The fast growth of the number of moves that HANOITOWERS requires is easy to see by noticing that every time HANOITOWERS($n, 1, 3$) is called, it calls itself twice for $n - 1$, which in turn triggers four calls for $n - 2$, and so on.

We can illustrate this situation in a *recursion tree*, which is shown in Figure 2.2. A call to HANOITOWERS(4,1,3) results in calls HANOITOWERS(3,1,2) and HANOITOWERS(3,2,3); each of these results in calls to HANOITOWERS(2,1,3), HANOITOWERS(2,3,2) and HANOITOWERS(2,2,1), HANOITOWERS(2,1,3), and so on. Each call to the subroutine HANOITOWERS requires some amount of time, so we would like to know how much time the algorithm will take.

To calculate the running time of HANOITOWERS of size n , we denote the number of disk moves that HANOITOWERS(n) performs as $T(n)$ and notice that the following equation holds:

$$T(n) = 2 \cdot T(n - 1) + 1.$$

Starting from $T(1) = 1$, this recurrence relation produces the sequence:

$$1, 3, 7, 15, 31, 63,$$

and so on. We can compute $T(n)$ by adding 1 to both sides and noticing

$$T(n) + 1 = 2 \cdot T(n - 1) + 1 + 1 = 2 \cdot (T(n - 1) + 1).$$

If we introduce a new variable, $U(n) = T(n) + 1$, then $U(n) = 2 \cdot U(n - 1)$. Thus, we have changed the problem to the following recurrence relation.

$$U(n) = 2 \cdot U(n - 1).$$

Starting from $U(1) = 2$, this gives rise to the sequence

$$2, 4, 8, 16, 32, 64, \dots$$

implying that at $U(n) = 2^n$ and $T(n) = U(n) - 1 = 2^n - 1$. Thus, HANOITOWERS(n) is an exponential algorithm.

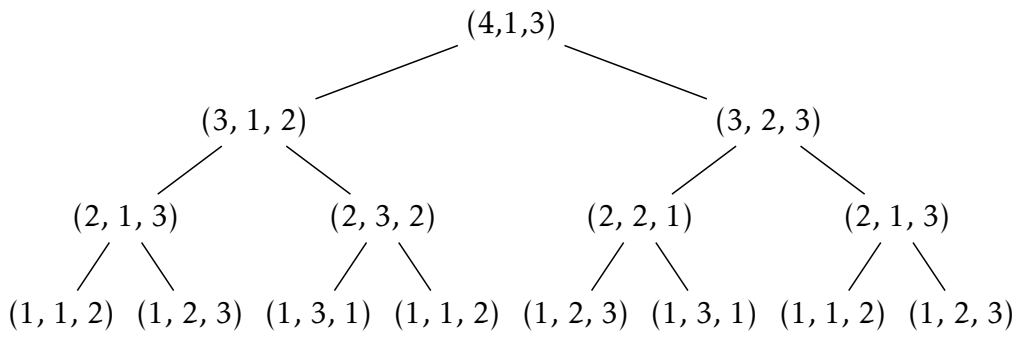
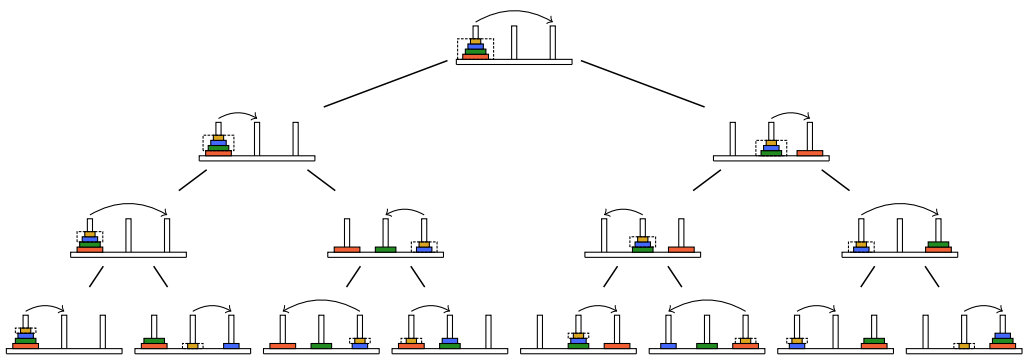


Figure 2.2: The recursion tree for a call to `HANOITOWERS(4,1,3)`, which solves the Towers of Hanoi problem of size 4. At each point in the tree, (i, j, k) stands for `HANOITOWERS(i, j, k)`.

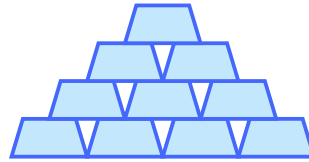
8.6 Maximum Amount of Gold

Maximum Amount of Gold Problem

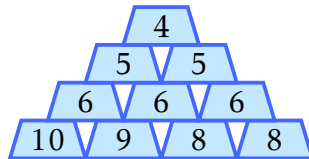
Given a set of gold bars of various weights and a backpack that can hold at most W pounds, place as much gold as possible into the backpack.

Input: A set of n gold bars of integer weights w_1, \dots, w_n and a backpack that can hold at most W pounds.

Output: A subset of gold bars of maximum total weight not exceeding W .



You found a set of gold bars and your goal is to pack as much gold as possible into your backpack that has capacity W , i.e., it may hold at most W pounds. There is just one copy of each bar and for each bar you can either take it or not (you cannot take a fraction of a bar). Although all bars appear to be identical in the figure above, their weights vary as illustrated in the figure below.



A natural greedy strategy is to grab the heaviest bar that still fits into the remaining capacity of the backpack and iterate. For the set of bars shown above and a backpack of capacity 20, the greedy algorithm would select gold bars of weights 10 and 9. But an optimal solution, containing bars of weights 4, 6, and 10, has a larger weight!

Input format. The first line of the input contains an integer W (capacity of the backpack) and the number n of gold bars. The next line contains n integers w_1, \dots, w_n defining the weights of the gold bars.

Output format. The maximum weight of gold bars that fits into a backpack of capacity W .

Constraints. $1 \leq W \leq 10^4$; $1 \leq n \leq 300$; $0 \leq w_1, \dots, w_n \leq 10^5$.

Sample.

Input:

```
10 3
1 4 8
```

Output:

```
9
```

The sum of the weights of the first and the last bar is equal to 9.

Solution 1: Analyzing the Structure of a Solution

Instead of solving the original problem, we will check whether it is possible to fully pack our backpack with the gold bars: given n gold bars of weights w_0, \dots, w_{n-1} (we switched to the 0-based indexing) and an integer W , is it possible to select a subset of them of the total weight W ?

Exercise Break. Show how to use the solutions to this problem to solve the Maximum Amount of Gold Problem.

Assume that it is possible to fully pack the backpack: there exists a set $S \subseteq \{w_0, \dots, w_{n-1}\}$ of total weight W . Does it include the last bar of weight w_{n-1} ?

Case 1: If $w_{n-1} \notin S$, then a backpack of capacity W can be fully packed using the first $n - 1$ bars.

Case 2: If $w_{n-1} \in S$, then we can remove the bar of weight w_{n-1} from the backpack and the remaining bars will have weight $W - w_{n-1}$. Therefore, a backpack of capacity $W - w_{n-1}$ can be fully packed with the first $n - 1$ gold bars.

In both cases, we reduced the problem to essentially the same problem with smaller number of items and possibly smaller backpack capacity. We thus consider the variable $pack(w, i)$ equal to `true` if it is possible to fully pack a backpack of capacity w using the first i bars, and `false`, otherwise. The analysis of the two cases above leads to the following recurrence relation for $i > 0$,

$$pack(w, i) = pack(w, i - 1) \text{ or } pack(w - w_{i-1}, i - 1).$$