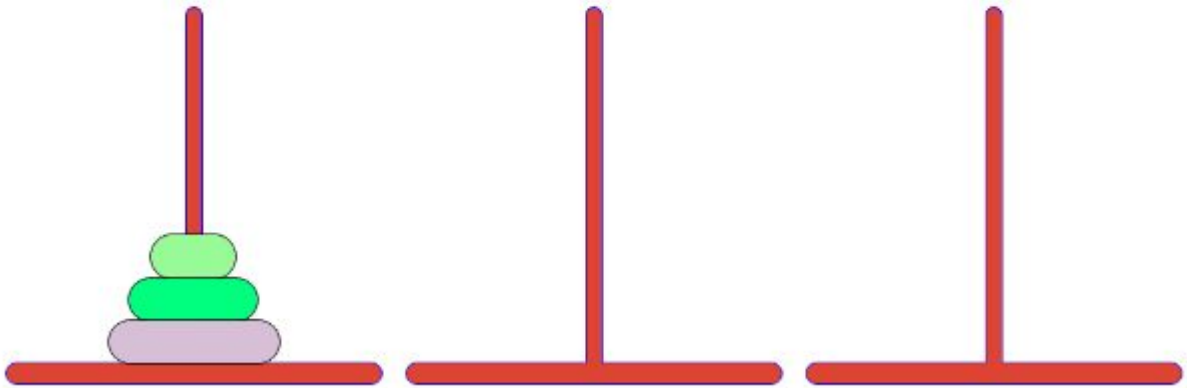
Recursion and Dynamic Programming

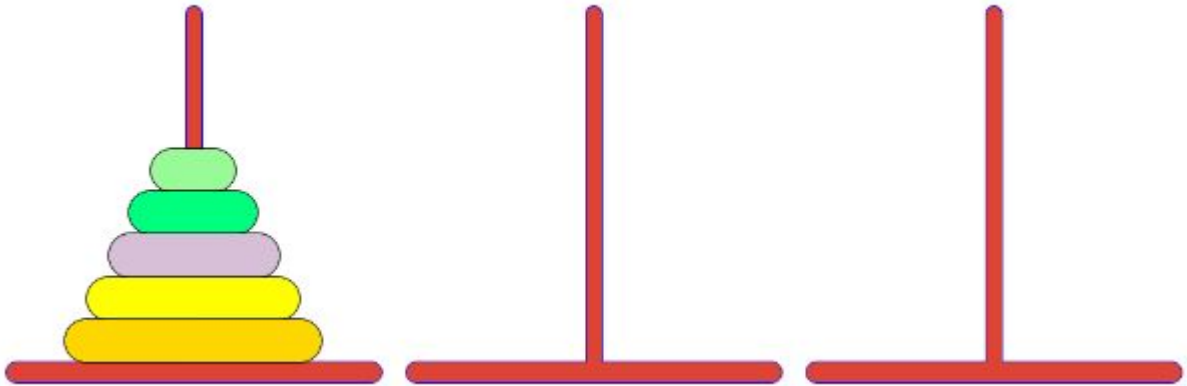
Towers of Hanoi

- Try it at https://www.mathsisfun.com/games/towerofhanoi.html
- How do you find the optimal order of moves?



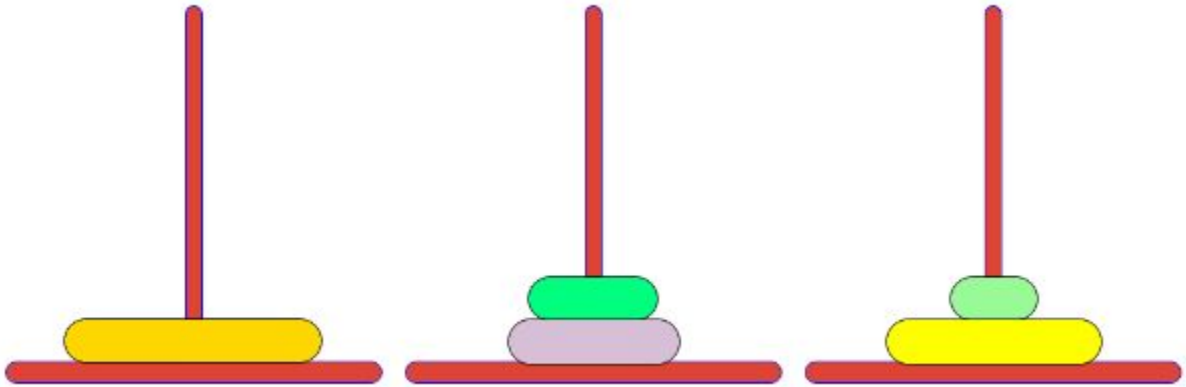
Towers of Hanoi

- Break the problem down into a recurrence
- Look at the disk on the bottom
- When can it be moved?



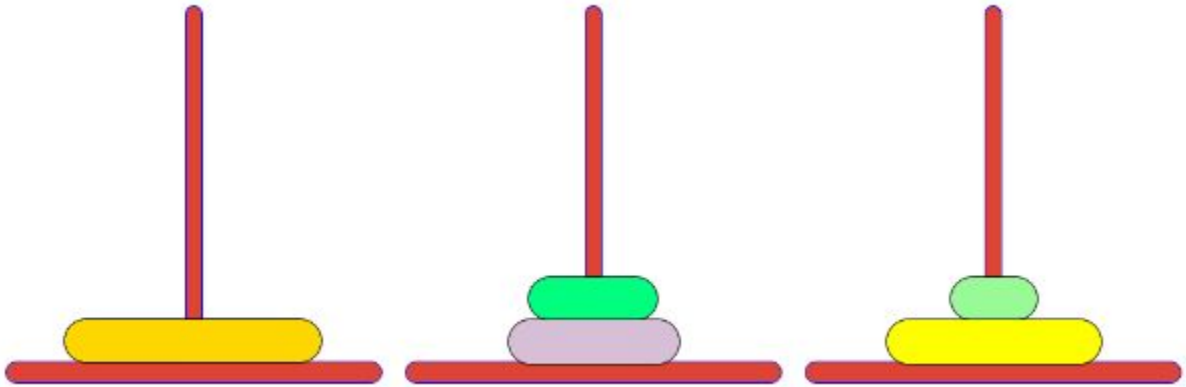
Towers of Hanoi

- First, the rules state that it can only move once *every* piece on top of it moves



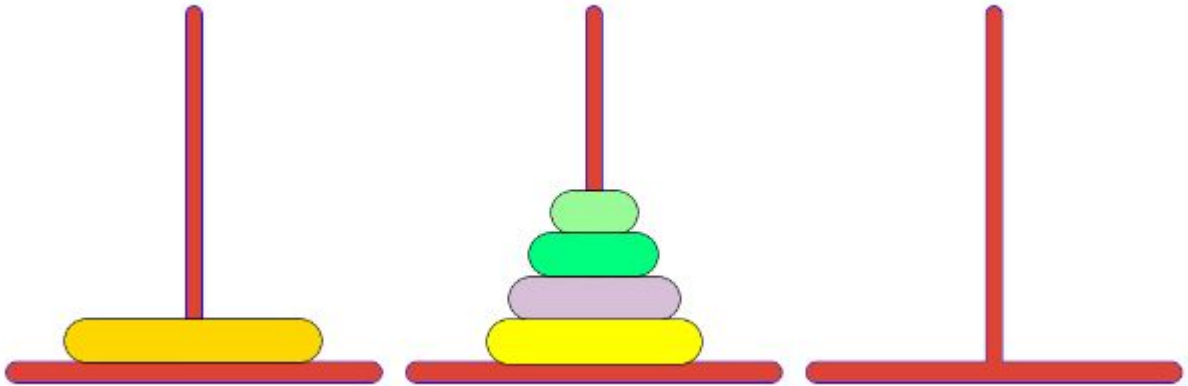
Towers of Hanoi

- First, the rules state that it can only move once *every* piece on top of it moves
- But this isn’t good enough



Towers of Hanoi

- There also needs to be a free space for it to move to
- Every disk in this space is bigger
- Since this is the largest disk the space must be empty



Towers of Hanoi

- The largest piece only moves once the top pieces are all moved to another peg
- Do you notice the recurrence?

Towers of Hanoi

- Move n-1 disks to the middle peg
- Move the nth disk to the final peg
- Then move n-1 disks from the middle peg to the final peg

Towers of Hanoi

- The cost to move all n disks is the cost of moving the largest disk, plus the cost of moving n-1 disks twice.
- In other words, if f(n) is the minimum number of moves for n disks, then f(n) = 2f(n-1)+1
- Also, clearly, the base case is f(1) = 1

Towers of Hanoi

- This gives us a way to calculate f(n) in n steps
- I won't show how to prove this, but it turns out that $f(n) = 2^n-1$
  - The easiest way is to guess (think about binary digits) and then prove your guess by induction

Another recurrence (and some dp)

Binary numbers without consecutive ones

- How many binary numbers are there without any ones touching?
  - e. g. 101001 counts, but 11001 doesn't
- Let g(n) be the number of numbers with n binary digits
- Look for a recurrence

Binary numbers without consecutive ones

- Each number ends with a 0 or a 1
- If it ends in a 1 then the last two digits are 01
- It can always end in a 0
- Now try to find the recurrence

Binary numbers without consecutive ones

- g(n+1) = g(n) + g(n-1)
  - g(n+1) is the number of strings of length n (with 0 appended to them), plus the number of strings of length n-1 (with 01 appended to them)
- Also, we know that the base cases are g(1) = 2, and g(2) = 3 (you can test this yourself)
- Is this sequence familiar?
  - 2,3,5,8,13,21,34...

Binary numbers without consecutive ones

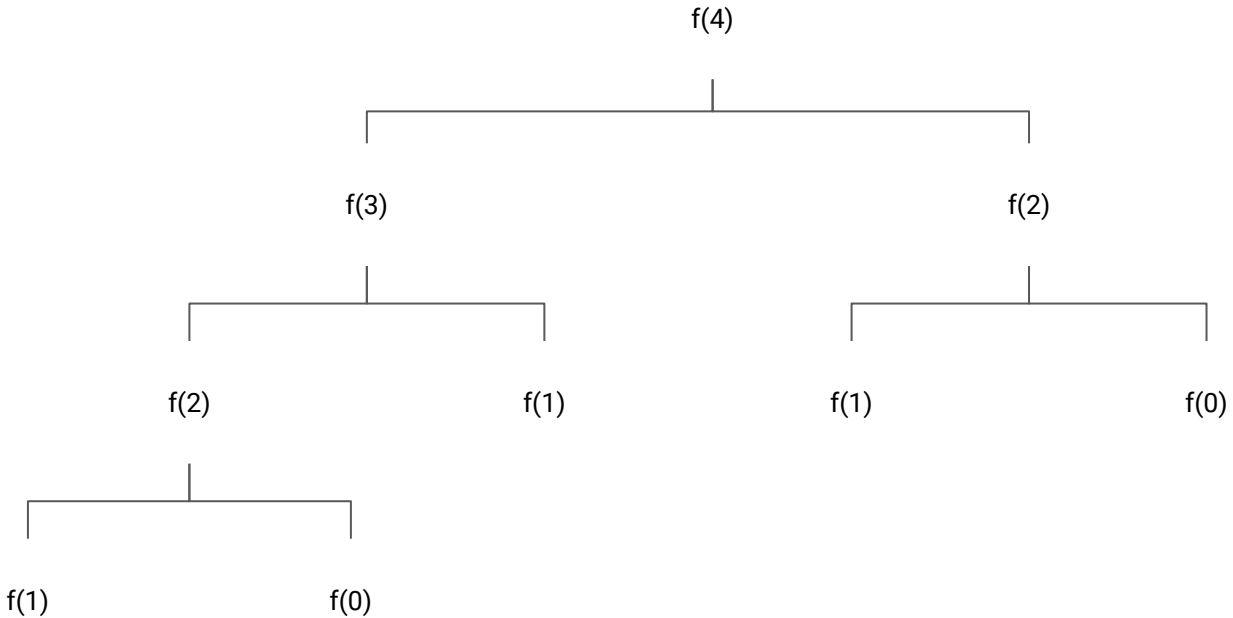
- g(n) is the (n+2)th Fibonacci number
- How do we calculate calculate this efficiently?

The stupid way

- If you just copy the formula for the Fibonacci sequence into python you get this:

```
2 def fib(n):
3     if n == 0 or n == 1: return 1
4     return fib(n-1) + fib(n-2)
```

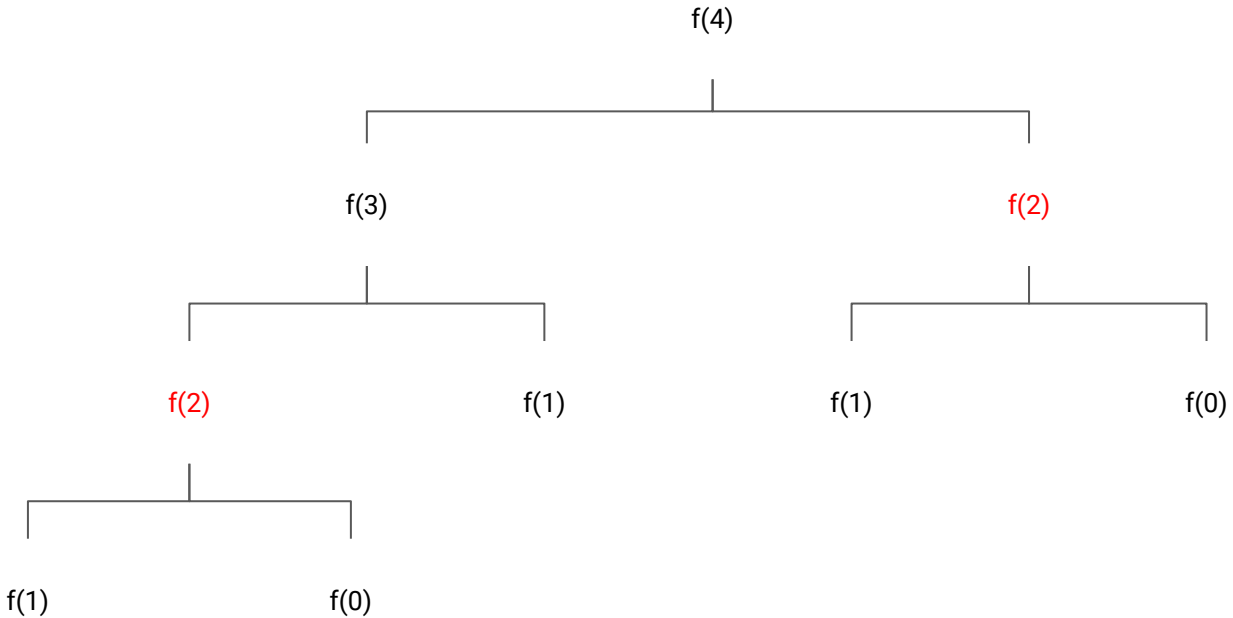
The stupid way



Why is it stupid

- Overlapping calls
- Unnecessary calculations
- Exponential time ($O(1.6^n)$)

The stupid way



Dynamic programming

- Dynamic programming is an optimization to normal recursion that works by remembering intermediate values to remove unnecessary calculations
- The point of dp in this scenario is to remove the repeated calculations
- We should never go through the process of calculating fib(n) twice for the same n

- We can ‘memoize’ results from this method
- Or we can build a list of Fibonacci numbers from the ground up

Memoization

- This is better but still not ideal
- Every function call has some overhead
- In other problems it might be more efficient because it doesn't evaluate unnecessary state
- O(n) space and time complexity

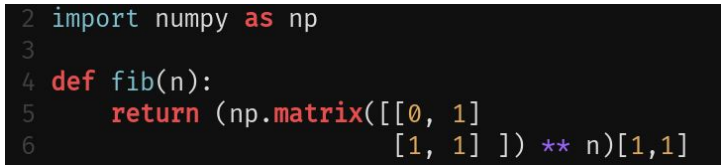
```
2  dp = [0] * 100000
3  dp[0] = 1
4  dp[1] = 1
5
6  # assume 0 <= n < 100000
7  def fib(n):
8      print(n)
9      if dp[n]: return dp[n]
10     dp[n] = fib(n-1) + fib(n-2)
11     return dp[n]
```

Bottom up

- Iteration is usually faster than recursion
- O(n) space and time complexity

```
fib = [0] * 100000
fib[0] = 1
fib[1] = 1
for i in range(2, len(fib)):
    fib[i] = fib[i-1] + fib[i-2]
```

Efficient Fibonacci (not dp)

Matrix exponentiation	Faster iterative solution
	
O(1) space O(log n) time	O(1) space O(n) time

Applications

- Dynamic programming is good for more than just Fibonacci numbers
- In this presentation I'll cover the 0-1 knapsack problem

The 0-1 knapsack problem

0-1 knapsack problem

- You have a set of objects, each with a size and a value
- Your backpack can only fit up to a certain size
- What is the maximum value you can bring?
- It’s called 0-1 because there is only one of each item
    - You have either 0 or 1 of each object in your backpack

0-1 knapsack problem

For example:

```
[ {size: 9, value: 15},
  {size: 4, value: 5},
  {size: 5, value: 6},
  {size: 3, value: 5} ]
```

If your knapsack has size 12, then the optimal solution is to take the first and last items, for a value of 20

0-1 knapsack problem

- Find a recurrence
- Let f(n, b) be the maximum value you can make with size exactly b while only considering the first n items
- f(n, b) = max(f(n-1, b), f(n-1, b-size[n]) + value[n])
- Store all values of f(n, b) with b <= the size of your bag
- Find the maximum
- Base case: f(0, b) = 0 for all b

0-1 knapsack problem

Let’s evaluate this recurrence using the ‘ground up’ strategy with knapsack size = 12 and this input:

```
[ {size: 4, value: 7},
  {size: 4, value: 5},
  {size: 5, value: 6},
  {size: 3, value: 5},
  {size: 2, value: 2} ]
```

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1													
n=2													
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0									
n=2													
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7								
n=2													
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2													
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0+5	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0									
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7								
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7+5	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7					
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7+5	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12				
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3													
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0									
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7					
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12				
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7+6	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12				
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4													
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0										
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3									
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3	7	7	7						
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3	7	7	7	12	12				
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3	7	7	7	12	12	13	13		
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3	7	7	7	12	12	13	13	17	
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3	7	7	7	12	12	13	13	17	18
n=5													

```
{size: 4, value: 7},
{size: 4, value: 5},
{size: 5, value: 6},
{size: 3, value: 5},
{size: 2, value: 2}
```

	b=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	0	0	0	0	0	0	0	0	0	0	0	0	0
n=1	0	0	0	0	7	7	7	7	7	7	7	7	7
n=2	0	0	0	0	7	7	7	7	12	12	12	12	12
n=3	0	0	0	0	7	7	7	7	12	13	13	13	13
n=4	0	0	0	3	7	7	7	12	12	13	13	17	18
n=5	0	0	2	3	7	7	9	12	12	14	14	17	18

0-1 knapsack problem

- The answer is 18!
- This version of the algorithm doesn’t return this, but it’s optimal to choose items 1, 2, and 4

Implementation

C++

```cpp
const int MN = 5005;
int n, m; // n items, max weight is m
pair<int, int> arr[MN]; // {weight, value}
int dp[MN][MN]; // dp[i][j] represents that if you use the first i items to get a weight of j you can
have a value of at most dp[i][j]

int knapsack(){
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= m; j++) dp[i+1][j] = dp[i][j];
        for (int j = arr[i].first; j <= m; j++){
            dp[i+1][j] = max(dp[i+1][j], dp[i][j-arr[i].first] + arr[i].second);
        }
    }
    return dp[n][m];
}
```

Implementation

Java

```java
int n, m;
int[] weights, vals;
int[][] dp;

int knapsack(){
    dp = new int[n+1][m+1];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= m; j++) dp[i+1][j] = dp[i][j];
        for (int j = weights[i]; j <= m; j++) {
            dp[i+1][j] = Math.max(dp[i+1][j], dp[i][j-weights[i]] + vals[i]);
        }
    }
    return dp[n][m];
}
```

Implementation

Python

```python
# arr is an array of tuples of size 2
def knapsack(arr, n, m):
    dp = [[0]*(m+1) for i in range(n+1)]
    for i in range(n):
        for j in range(m+1):
            dp[i+1][j] = dp[i][j]
        for j in range(arr[i][0], m+1):
            dp[i+1][j] = max(dp[i+1][j], dp[i][j-arr[i][0]] + arr[i][1])
    return dp[n][m]
```

Practice

- Find a recurrence for the towers of hanoi where you're only allowed to move disks to adjacent towers.
- https://cses.fi/problemset/task/2205 (recursion practice)
- https://cses.fi/problemset/task/2165 (towers of hanoi)
- https://cses.fi/problemset/task/1636 (coin change dp)
- https://dmoj.ca/problem/ccc07j5 (dp)
- https://dmoj.ca/problem/ccc15j5 (dp problem)
- https://dmoj.ca/problem/coci18c6p3 (backpack dp)
- https://projecteuler.net/problem=770 (recurrence/dp)
- https://dmoj.ca/problem/dpy (dp problem)
- https://codeforces.com/problemset/problem/1593/F (dp with more dimensions)