Practice problems

Problem 1 - Connectivity

Write function `is_conncted(n, edges)`, that takes as an input a number `n` -- the number of vertices of a graph, and a list `edges` of edges -- each a pair of (non-equal) numbers between 0 and $n-1$. The function should return `True` if a graph specified by edges `edges` is connected and `False` otherwise.

Examples

```
edges = [(0, 1), (1, 2), (0, 2), (3,4)]
is_connected(5, edges) # returns False
```

The graph specified by the edges above consist of a triangle on vertices $0, 1, 2$ and a separated edge connected vertices $3, 4$. It is not connected -- there is no path from vertex 0 to 3.

```
edges = [(0, 1), (1, 2), (0, 2), (3,4), (3, 2)]
is_connected(5, edges) # returns True
```

The graph is similar to the previous one, except with additional edge connecting vertex 2 and 3. Now it is connected.

```
edges = [(0, 1), (1, 2), (0, 2), (3,4), (3, 2)]
is_connected(6, edges) # returns False
```

The graph has the same edges as the previous one, but now has 6 vertices. The vertex with index 5 is an isolated vertex (with no edges incident to it) --- so the graph is not connected.

Hint

It is easiest to solve this problem using a variant of the DFS algorithm discussed in lecture 12.

Problem 2 - Balls and bins

Write a functions `simulate_balls_and_bins(n,m)`, that takes two arguments, `n` and `m`.

We imagine having n, initially empty bins, to which we throw m (identical) balls, at random, one by one. The function should iterate for m steps, and in each iteration throw a "ball" to a random "bin" - i.e. a "bin" with a random index from 0 to $n-1$.

The function should return a list of size m; the k-th element of which should correspond to the maximum number of balls in any of the bins, after the first k balls have been thrown.

Write also a function `plot_balls_and_bins(n,m)`, that calls the previous function and plot how the maximum load of a bins changes as we throw balls.

Reminder
To generate a random integer in the range $[a, b]$, including both points (where `a` and `b` are integers), you can use the following code:

```
import random

random.randint(a, b)
```

Use

```
import matplotlib.pyplot as plt
```

to import a library used to plot things in Python.

To produce a single plot with a line connecting points given by a single list of (integer, or float) values, for example `y = [2, 2, 3, 3, 6]`, you can now just call

```
plt.plot(y)
```

(this produces a plot connecting points $(i, y[i])$).

Problem 3 - Subset sum

Write a function `subset_sum(lst, target)`, that takes two arguments: a list of positive integers, and a positive integer `target`. It should return all subsets of indices $\{0, \ldots, \text{len}(\text{lst}) - 1\}$, such that the sum of elements $\text{lst}[i]$ over this subset of indices is exactly `target`.

Example

```
subset_sum([1,2,3,4,5,6], 6) == [ [0, 1, 2], [1, 3], [0, 4], [5] ]
```

since $1 + 2 + 3 = 6$, but also $2 + 4 = 6$, $1 + 5 = 6$, and $6 = 6$.

```
subset_sum([2,2,2,2,2], 2) == [[0], [1], [2], [3], [4]]
```

Since each element by itself is equal to 2.

```
subset_sum([4,5,6,7,8,9,10], 2) == []
```

Complexity

Your solution might run in the worst-case in time $\Theta(2^n)$. Partial points will be given for a solution that always runs in time 2^n -- for example by generating all possible subsets of $\{0, \ldots, \text{len}(\text{lst}) - 1\}$ and checking for each of those if its adds up to target.

To get full points, implement a recursive (backtracking-based) solution, which has potential of running much faster on some inputs: in each stage attempt to either append or not append the k-th element to the set, and run the same procedure recursively on the remaining elements, and updated target. Observe that if the updated target is negative, there could be no solution.

For example, when the `target` is smaller than all elements in the sequence, your implementation should run in time $O(n)$.