dynprog_exercises

Dynamic Programming exercises

1. Improvement to Floyd Warshall code

See the three tasks in the final code provided. Tasks 1 and 2 are self-explanatory. Task 3 is intended for efficiency and it's more general.

The reconstruction of the optimal solution in dynamic programming codes is typically done by back-tracking the optimal decisions, from the last one to the first one. In the Floyd-Warshall algorithm, we used the `insert` method when building the solutions.

This way of doing things is not optimal: each insertion operation in a list is $O(n)$. (And inserting things at the beginning is the worst possible case.) Thus, creating a list in this way is $O(n^2)$.

Appending items at the end, on the other hand, is $O(1)$ (amortized). Therefore, a better way to do things is to reconstruct the solutions in reverse, using `append`, and then reverse the result at the end, in-place (work out the overall complexity of this procedure and convince yourself that it is convenient to do things this way).

Your task 3 is to perform this optimization in the F-W code. This obviously means that you start from just `[j]`, append the intermediate nodes, and finally append also node `i`; then reverse everything. Of course, be sure to test what you did with the test function that we wrote.

Note: remember that there is a big difference between using the `append` method for lists and using concatenation with `+`. In case you were considering using the latter: don't. (Also, if this is not immediately clear to you, it's a good time to look it up and make sure you get this.)

Tip: As a preliminary step, create an alternative version of the code, then run both the new version and the old one, and make sure that they always give the same result, by extensive testing. This is the correct way to do things when optimizing code. This is how you proceed: you rename the variables that represent the reconstructed configuration, then write a new piece of code that computes them in a more efficient way, and put some assertions afterwards that ensure that the new version and the old one are equal. When everything works, you can remove the old code and the assertions.

2. Dynamic programming and recursion

As we've seen, dynamic programming is basically recursion + memoization, usually performed bottom-up instead of top-down, and it's useful when 1) the recursion tree is exponential in size, and 2) you can re-use sub-problems that reappear often in the recursion tree, so that in the end only a polynomial number of operations is required.

In this exercise, the idea is to just take the simplest possible case of a cost computation, even if it's not exponential, and do it with recursion and dynamic programming.

Our problem is extremely simple: compute the minimum over a given list. Suppose that each element of the list represents a cost. We'll try to work out what the dynamic programming approach looks like in this case.

First we need a recursive formula. One straightforward way to write the solution using recursion is to define the minimum of the whole list as the minimum between a) the first element and b) the minimum over the rest of the list. The base case is when the list is empty, and than the minimum is ∞ by default. Write a function `minimum_rec1` which does this.

Let's now make it a little bit more similar to dynamic programming, by writing the recursion backwards instead. In this analogy, the "state" of the recursion is given by how many elements of the list we have seen so far, let's call that `k`. Then it's obvious that the minimum after we've seen `k` elements (with `k>0`) is either the minimum that we had seen among the first `k-1` elements or the last element that we've seen. In formulas:

1. $c(0) = \infty$
2. $c(k) = \min\left(c(k-1), l[k-1]\right) \quad \text{if } \ k > 0$

Note that we are using Python 0-based indexing for `l`.

So now you should write a recursive version of the algorithm, in a function `minimum_rec2`, based on the above formula (it's a trivial modification of the previous recursive version, you just single-out the last element of the list rather than the first).

Extra hint if you're confused at this point: The function $c(k)$ above is computing `minimum_rec2(l[:k])`).

Then, you should write the dynamic programming version of this, in a function `minimum_dp1`. This is really easy, it's just a `for` loop over `k`. Notice however that the loop proceeds *forwards*, in the opposite sense of the recursive relation! If you think in terms of the interpretation of `k` given above it's easy to understand why.

Note: I strongly advise that while you write these algorithms, you keep around some `test` function that generates random data (or takes data provided by the user) and checks that all these algorithms give the same result (you can also compare with the standard functions provided by Python and numpy).

In dynamic programming, as we've seen repeatedly, it is often useful to keep track of some auxiliary quantity that allows us to reconstruct the optimal decision. In general, all we need to do is this: whenever we compute a minimum, we should remember which choice we took. In this particular case, we need to keep track, at each `k`, of whether we decided to accept the new element as the new minimum or not.

So you should write an algorithm `minimum_dp2` that also returns the position of the minimum, not just its value. (For an empty list, just return the position `-1`.) There are actually a couple of ways to do this. One is more standard, in which you remember every decision that you took. Another is more similar to the Floyd-Warshall case, in which you keep an "incumbent" and only update it in some cases. The former is more easily derived from the general dynamic programming scheme, the latter is more efficient (and, in practice, simpler, at least in this case).

Finally, you can write a recursive version of the algorithm, call it `minimum_rec3`, which also does that, i.e. returns the position of the minimum. Note that this will be rather easy to do if you base it on `minimum_rec2`, but if you want to do it based on `minimum_rec1` there is an additional complication. Can you see why? Try to do both.

Having done all of this, consider now:

1. The time complexity of the two algorithms
2. The space complexity of the two algorithms

Hint: the second point requires you to remember how recursion and function frames work in Python. If you don't remember what frames are, they are covered in section 3.9 of the Think Python book.

Finally, consider whether it's useful to memoize the recursive versions or not in this case.

3. Shortest path on a weighted, acyclic, layered graph

In short: implement the dynamic programming code for the shortest path problem described in the accompanying "dir_graph.pdf" notes. This is the same problem that we have already seen as our first example of a dynamic programming algorithm, but slightly simpler: we assume that the input and output layers have only one node each. The notes in "dir-graph.pdf" also define the problem, how to solve it with dynamic programming, and demonstrate the solution on an example. We will use the same example to test our code. Be aware that this is a long exercise. But it can be done in stages. Be sure to test each step thoroughly.

As a side note, this problem offers what is probably the easiest non-trivial example of a dynamic programming algorithm, because the computational graph of dynamic programming coincides with the underlying graph that you are studying.

However, the structure of the graph itself is not trivial: there are missing links, and exploiting its layered structure is essential to the algorithm. So the difficulty here is actually creating and manipulating the graph. As usual, there are several possible representations. The one I suggest is as follows: a list of rectangular matrices. Let's call `g` this list. The length of the list will be `L-1`, where `L` is the total number of layers in the graph. Each entry of the list will contain the connections weights between one layer of the graph and the next. For example, the matrix at `g[0]` will contain all the connections between layer `0` and layer `1`; the matrix at `g[1]` those between layer `1` and layer `2`, and so on. Each matrix `g[l]` will have as many rows as there are nodes in layer `l`, and as many columns as there are nodes in layer `l+1`. Since our example graph has only one node at layer `0`, then `g[0]` will only have 1 row. Since we assume that our graphs have only one node in the last layer, then `g[-1]` will only have 1 column. Also, in between, the number of columns of `g[l]` must match the number of rows of `g[l+1]`.

Here is the graph of the notes, represented in this way:

```
test_g1 = [
        np.array([
                [   6.0,    8.0,   13.0],
                ]),
        np.array([
                [   9.0,   15.0, np.inf],
                [   8.0,   10.0,   12.0],
                [np.inf,    8.0,    7.0]
                ]),
        np.array([
                [  15.0, np.inf],
                [  20.0,    8.0],
                [np.inf,    7.0],
                ]),
        np.array([
                [   3.0],
                [   4.0]
                ])
        ]
```

Your first task: write a function `check_graph` that takes an input `g` and checks that it obeys all of the above constraints (i.e. that it's a non empty list in which the elements are 2-d arrays, and such that their shapes have the correct relationships).

Remember to make sure that the above function (as well as all the other ones below) also work for the extreme case of a trivial graph with just two layers, each with a single node, and thus with a single link connecting them. That kind of graph would look like this:

```
test_g0 = [np.array([[3.3]])]
```

Another useful function to have for this kind of representation is one that computes the overall number of nodes. Write it. (You can do it in one line of code, if you use comprehensions.)

Now observe that, in the original picture in the notes, the nodes were numbered from `0` to `9`. In this representation, however, the nodes are naturally represented by two integers: the layer number `l` and the inter-layer index `j`. For example, node `2` in the picture becomes node `(1,1)` in this representation. Here is the full mapping for the example:

```
0 ⟹ (0, 0)        5 ⟹ (2, 1)
1 ⟹ (1, 0)        6 ⟹ (2, 2)
2 ⟹ (1, 1)        7 ⟹ (3, 0)
3 ⟹ (1, 2)        8 ⟹ (3, 1)
4 ⟹ (2, 0)        9 ⟹ (4, 0)
```

So here is one more, quite non-trivial, task (you can skip it for now and proceed, but it's useful to do this): write two functions that, given a graph, convert from one form to the other. When you are done, you should also write an auxiliary test function that, for each node in the graph, checks that converting from the single-index representation to the (layer+inter-index) representation and back returns the original node. And also, that for each valid combination (layer+inter-index), you can compute the single-index and go back. This will likely take some time to figure out.

By the way, the above two functions should also produce errors in case of out-of-bounds inputs.

Having done all that, you may start considering that you now have quite a few specialised functions for this representation. Perhaps it's time to put it all in a `class`, and make those functions be class methods.

Regardless of whether you use a class or not, the next step is to actually write the dynamic programming algorithm as explained in the "dir_graph.pdf" notes. As for the previous codes, start by writing the forward pass only, that just performs the optimal cost computation; don't bother with the backtracking part yet. Observe that the recursive relationship is:

- $c(0) = 0$
- $c(i) = \min_{j \in pred(i)} \{c(j) + w_{ij}\} \quad \text{if } i > 0$

This is written in terms of the single-indices nodes though, and it does not make exceedingly clear that you should loop over the layers first, and inside the layers next.

So your first possibility, if you wrote the index conversion functions mentioned above, is this: pre-allocate all of the matrix `c` with as many nodes as the graph, then use two loops (one over the layers and one inside the layer's nodes), and use the above formula, leveraging the index conversion functions. If you didn't write them, go to the next step.

The second (and, actually, simpler) possibility is to create a structure for the cost which mimics the graph, i.e. a list of numpy 1-d vectors. Each element of the list corresponds to a layer, each entry in the vector corresponds to a node in that layer. Notice that you can not use a 2-d array since all the vectors have variable length: it's not a matrix! You should create a function that returns a list like that, for a given graph. You should initialize it all to infinity, except for the first element, which should be zero. Of course, you can make it a class method, if you chose to use classes.

With this new cost structure, let's re-write the recursive relations. This time, we use pairs of indices, and bracket notation to make it more similar to Python's:

- $c[0][0] = 0$
- $c[l][j] = \min_{k \in \ell(l-1)} \left\{ c[l-1][k] + g^{l-1}_{kj} \right\} \quad \text{if } l > 0$

Here we used the notation $\ell(l)$ to denote the elements of layer l, and we changed w_{ij} with our list-of-matrices representation g, which thus has three indices.

This is sufficient for computing the optimal cost, which will be located at the end of the structure `c` (in the only entry of the last layer).

Then, you will want to actually get the best path. You need another structure similar to the one for the cost here, except that: (1) it should be made of integers (2) it should be initialized with some sentinel value, like `-1`. Then you use it to store the "argmin" corresponding to the $\min$ of the above recursive relation. You can call it `whence`, like in our previous codes.

At the end, you need to back-track the `whence` structure and build the path from the end to the front (which is sort of tricky, as usual, but not terrible), and you return it. If you have written the index conversion functions mentioned above, you can return the path as a list or array of integers, which maybe looks a little nicer; otherwise return a list of tuples, each with (layer,inter-index).

Verify that when you run the algorithm on the test graph of the notes you get the correct cost and path. Also verify that everything works in the extreme case of `test_g0`.

Advanced task - numpy optimizations

As for the Floyd-Warshall case, the algorithm has 3 nested loops, and only the outer one (on the layers) is the dynamic programming one. The inner two can be vectorized by leveraging numpy's broadcasting rules. In fact, they can be reduced to 3 lines of code, and not even too cryptic. Can you do that?

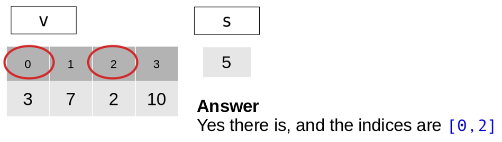
4. Subset Sum

We consider this problem:

- given:
    - a vector v (of length n) whose elements are strictly positive integers ($v_i \in \mathbb{N}$, $v_i > 0$)
    - a non-negative integer number s
- is there at least one sub-set of elements of v such that their sum is s? And if there is, what are the indices of the v elements that make up this sub-set?

Examples:

- Suppose `v=[3,7,2,10]` and `s=5`. This will be our main example in the following:



Answer
Yes there is, and the indices are `[0,2]`

- Suppose `v=[3,7,2,10]` and `s=17`. Then, the answer is "yes" and the indices are `[1,3]` because `v[1]+v[3]==17`.
- Suppose `v=[3,7,2,10]` and `s=10`. Then the answer is "yes" and there are two valid answers, either `[0,1]` or `[3]` are indices that identify a valid sub-set (and we just need one answer, we don't care which).
- Suppose `v=[3,7,2,10]` and `s=21`. Then, the answer is "no": no combination of the elements is a valid sub-set. (Note that you cannot use the same element more than once.)

This problem is much harder than it may seem, since there are 2^n possible subsets of v. However, if `s` is small, we can solve it efficiently with dynamic programming.

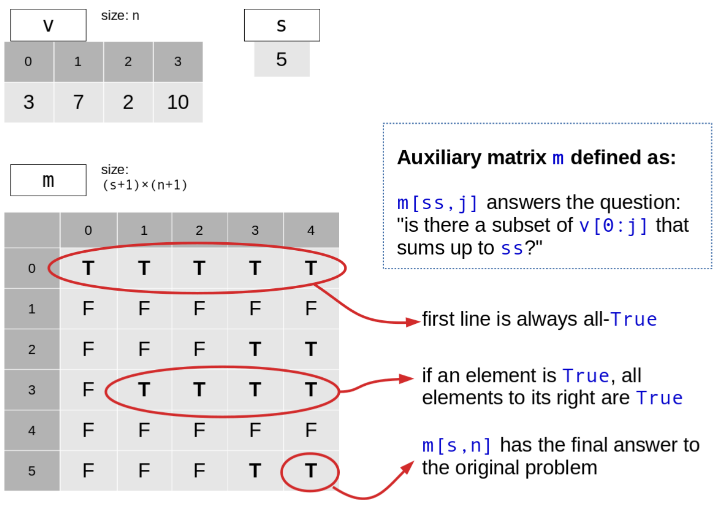
The dynamic programming scheme

The crucial idea is to have an auxiliary matrix, call it m, of size $(s+1) \times (n+1)$, whose elements are boolean (`True`/`False`). Then, for each value of $ss \in \{0, ..., s\}$ and of $j \in \{0, ..., n\}$ we want to have this:

> `m[ss,j]` **answers the question *"is there a subset of*** `v[0:j]` ***that sums up to*** `ss` ***?"***

Note that (see also the example in the figure below):

1. `m[s,n]` is exactly the answer to our original problem because `v[0:n]` is just equal to `v`.
2. `m[0,j]` is always `True` for all `j`, because in order to obtain a sub-sum of `0` we can just take an empty subset.
3. `m[ss,j]` can be easily computed if we already know all the values of `m[zz,k]` with `zz<=ss` and `k<j`, as we will detail below.


v
size: n
0
1
2
3
3
7
2
10
s
5
m
size:
(s+1)×(n+1)
first line is always all-True
if an element is True, all elements to its right are True
m[s,n] has the final answer to the original problem

	0	1	2	3	4
0	T	T	T	T	T
1	F	F	F	F	F
2	F	F	F	T	T
3	F	T	T	T	T
4	F	F	F	F	F
5	F	F	F	T	T

Auxiliary matrix `m` defined as:

`m[ss,j]` answers the question:
"is there a subset of `v[0:j]` that sums up to `ss`?"

The above three facts mean that we can simply compute the matrix `m` one row at a time, one column at a time. At the end we read out the answer from the value at the bottom-right corner.

This means that at the start of our program we will need to:

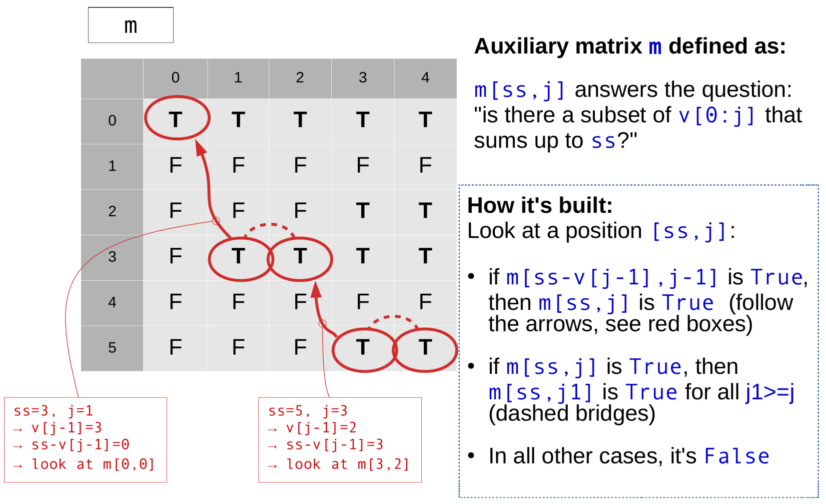
- allocate the matrix `m`, with type `bool`, initialized to `False`
- overwrite its first line with `True` (point 2 above)

Then the forward pass will need to be structured in this way: there must be a double `for` loop, an outer one over the matrix rows (`ss` in `0...s`) and then one over the columns (`j` in `0...n`).

Inside these, given `ss` and `j`, we need to compute the quantity `rest=ss-v[j-1]` and check whether `rest` is non-negative and `m[rest,j-1]` is `True`. If it is:

- It means that there is a subset of `v[0:j-1]` that sums up to `rest=ss-v[j-1]`. Therefore, we can put that subset together with `v[j-1]` and obtain a subset of `v[0:j]` that sums up to `ss`.
- So in that case, we can clearly set `m[ss,j]=True`.
- But notice that, and **this is very important**, also all the other elements with `j1>=j` are set to `True`, because any subset of `v[0:j]` is also a subset of `v[0:j1]` when `j1>=j`. Thus we need to do that and then skip the remainder of the `j` loop and proceed to the next row.

If, on the other hand, `rest` is negative or `m[rest,j-1]` is `False`, we do nothing and move on.



	0	1	2	3	4
0	T	T	T	T	T
1	F	F	F	F	F
2	F	F	F	T	T
3	F	T	T	T	T
4	F	F	F	F	F
5	F	F	F	T	T

```
ss=3, j=1
→ v[j-1]=3
→ ss-v[j-1]=0
→ look at m[0,0]
```

```
ss=5, j=3
→ v[j-1]=2
→ ss-v[j-1]=3
→ look at m[3,2]
```

Auxiliary matrix `m` defined as:

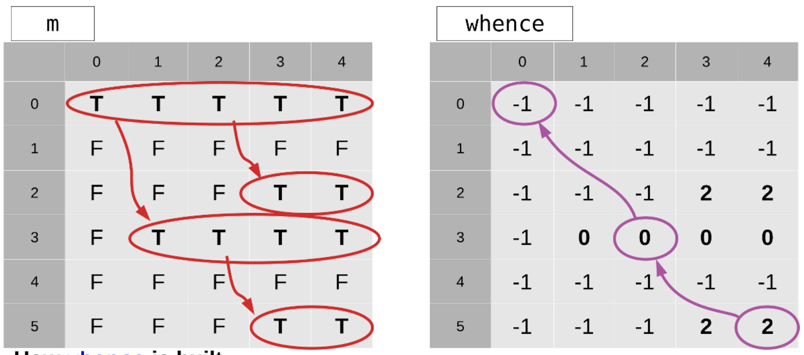
`m[ss,j]` answers the question: "is there a subset of `v[0:j]` that sums up to `ss`?"

How it's built:
Look at a position `[ss,j]`:

- if `m[ss-v[j-1],j-1]` is `True`, then `m[ss,j]` is `True` (follow the arrows, see red boxes)
- if `m[ss,j]` is `True`, then `m[ss,j1]` is `True` for all j1>=j (dashed bridges)
- In all other cases, it's `False`

With the matrix `m`, we only get to know if a valid subset exists, not what it is. As usual in dynamic programming, you should start out by implementing this and checking that it works by looking at the bottom-right element of the matrix, and comparing it with a few examples. When this works, move on.

After the forward pass, we need to implement the backward pass too. We need another auxiliary structure (called `whence` as usual) to keep track of the choices we made while building `m`, and then we use it to reconstruct the subset at the end. There are hints on how to do this in the following, and the figure below has some details that should help you.



m

	0	1	2	3	4
0	**T**	**T**	**T**	**T**	**T**
1	F	F	F	F	F
2	F	F	F	**T**	**T**
3	F	**T**	**T**	**T**	**T**
4	F	F	F	F	F
5	F	F	F	**T**	**T**

whence

	0	1	2	3	4
0	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1
2	-1	-1	-1	**2**	**2**
3	-1	**0**	**0**	**0**	**0**
4	-1	-1	-1	-1	-1
5	-1	-1	-1	**2**	**2**

How `whence` is built:
In each row except the first:
In the position `j` where `m` has the first `T`, `whence` stores `j-1`. In all the other positions with `j1>j`, it stores `j-1` too. In all other positions it's `-1`.
Using `whence`, starting from the bottom-right corner (if it is `True`), you can back-track and get the indices (in this case, `[0,2]`, see the purple annotations)

[Side note: this overall implementation could easily be improved to save a lot of space in memory, but we will ignore that. Feel free to think about it and do it as a challenge at the end.]

Some additional details and hints:

1. You should have an argument check that ensures that all elements of the input vector are strictly positive. You should try to do it without using `for` loops, only numpy methods.

2. After the forward pass, you should make the function return `None` in case there is no valid subset.

3. You should initialize the auxiliary structure `whence` to a matrix filled with `-1`. Try to do it with a single line of code.

4. The auxiliary matrix `whence` must be updated as appropriate during the forward pass. Look at the figure above: you only change the default (`-1`) in the positions where you set `m` to `True` (which are parts of rows from some `j` to the end) and you always fill in `j-1` for the whole partial row.

5. Before you implement the backward pass, you should probably have a look at the matrix `whence` and compare it with the example in the figure. If it's correct, write the backward pass. This you'll need to figure out by yourself.

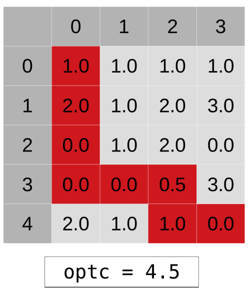
6. If you succeed, it's good practice to check your solution: right before returning, add a line that verifies that the indices `inds` are valid: compute the sum of `v` on those indices explicitly and check that the result is `s`. Make sure that an error occurs otherwise. Try to do it in one line of code exploiting numpy, and use an assertion (since this is debug-type code that should never fail if things are correct).

5. Optimal path through a matrix

We consider this problem:

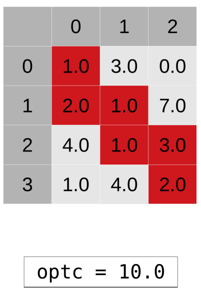
- given a matrix w (of size $n \times m$) whose elements are non-negative ($w_{ij} \in \mathbb{R}$, $w_{ij} \geq 0$)
- find the path of minimal cost that connects the top-left corner with the bottom-right corner. The path can only move down or to the right, not diagonally.

See the examples in the figure below:



	0	1	2	3
0	**1.0**	1.0	1.0	1.0
1	**2.0**	1.0	2.0	3.0
2	**0.0**	1.0	2.0	0.0
3	**0.0**	**0.0**	**0.5**	3.0
4	2.0	1.0	**1.0**	**0.0**

`optc = 4.5`



	0	1	2
0	**1.0**	3.0	0.0
1	**2.0**	**1.0**	7.0
2	4.0	**1.0**	**3.0**
3	1.0	4.0	**2.0**

`optc = 10.0`

The dynamic programming scheme

This problem can be solved efficiently with dynamic programming. The crucial idea is to have an auxiliary matrix, call it c, of size $n \times m$, whose element i, j represents the solution of the problem for the sub-matrix `w[0:(i+1), 0:(j+1)]`.

We have the following recursive relations:

1. $c_{i0} = \sum_{k=0}^{i} w_{k0}$ for $0 \leq i < n$
2. $c_{0j} = \sum_{k=0}^{j} w_{0k}$ for $0 \leq j < m$
3. $c_{ij} = w_{ij} + \min\left(c_{(i-1),j}, c_{i,(j-1)}\right)$ if $i, j > 0$

The first two cases are the base cases of the recursion.

As usual in dynamic programming, we also need another auxiliary matrix `whence` too keep track of the choices made at each step. This matrix will have size $n \times m$ and be made of integers. We will represent with `-1` the "left" choice, and with `1` the "up" choice. The top-left corner has no predecessor and thus will contain a sentinel value.

You may notice that this problem is very similar to, but simpler than, the seam carving problem. You should thus have no particular problems in writing the code from start to end, following the usual pattern (forward pass first, test that, add the `whence` matrix, implement the backward pass).

Some additional details and hints:

1. Check the argument: make sure that the matrix is not empty (has more than zero rows/columns) and that there are no negative entries. Try to do it without for loops.

2. Implement the two base cases for the c matrix, i.e. fill the first row and column, with numpy operations, without for loops.

3. The output path should be a list or array, containing just `-1` ("horizontal movement") or `1` ("vertical movement"). Its length must be $(n + m - 2)$, can you see why? You can thus either pre-allocate it and fill it in starting from the back, or you can start with an empty list, append values to is as you backtrack and reverse it at the end.

4. Add checks to your code. If you used the method of growing an empty list, check that its length is correct. Check that the output path contains exactly $n - 1$ ones, and $m - 1$ minus ones. You should also to another check: start from the $0, 0$ corner and accumulate the costs as you follow the path through the w matrix until the last entry, and check that the total equals what you had in the bottom-right corner of the `c` matrix.

5. In case of equal costs, your code will either favor the "left" case over the "top" case, or vice versa. Which one did you implement? How would you switch between the two possibilities?

6. Find the optimal skipped-choice subset

We consider this problem:

- given a vector w (of size n) with real positive entries ($w_i \in \mathbb{R}, w_i > 0$)
- we want to find a subset of the indices $\mathcal{S} \subseteq \{0, \ldots, n-1\}$ such that the sum of the elements at those indices $s = \sum_{i \in \mathcal{S}} w_i$ is maximum…
- …with the constraint that there are no consecutive indices within $\mathcal{S}$. In other words, if we pick an element, than we cannot pick either the previous one or the following one.

For example:

- if the input is `w = [2, 3, 4]` then our best choice is to pick `2` and `4`, and thus $\mathcal{S} = \{0, 2\}$ and $s = 6$

- if the input is `w = [2, 9, 4]` then our best choice is to pick only `9`, and thus $\mathcal{S} = \{1\}$ and $s = 9$
- if the input is `w = [14, 3, 27, 4, 5, 15, 1]` then our best choice is to pick `[14, 27, 15]` and thus $\mathcal{S} = \{0, 2, 5\}$ and $s = 56$
- if the input is `w = [14, 3, 27, 4, 5, 15, 11]` then our best choice is to pick `[14, 27, 5, 11]` and thus $\mathcal{S} = \{0, 2, 4, 6\}$ and $s = 57$

Additional (obvious) corner cases:

- if the input list is empty, `w = []`, then $\mathcal{S} = \emptyset$ and $s = 0$
- if the input list has only one element, `w = [w0]`, then $\mathcal{S} = \{0\}$ and $s = w_0$

The dynamic programming scheme

This problem can be solved efficiently with dynamic programming. The crucial idea is to have an auxiliary vector, call it c, of size n, whose element i represents the optimal value for the sub-vector `w[0:(i+1)]`. Clearly, the last element of c is the optimal value s for the original problem.

We have the following recursive relations:

1. $c_0 = w_0$
2. $c_1 = \max\left(c_0, w_1\right)$
3. $c_i = \max\left(c_{i-1}, c_{i-2} + w_i\right) \qquad$ if $i \geq 2$

The **first** case is the base case of the recursion. It just means that, when you have only one element, you pick it.

The **third** case is the general recursive formula: we can either *skip* the current element (left entry in the $\max$) or *pick* the current element and discard the previous one (right entry in the $\max$). Observe that in the "skip" case, we are allowed to use the best result including the previous element `i-1`. In the "pick" case, we add `w[i]` to our value, but then we can only use the best result up to element `i-2` since we cannot use `i-1`.

The **second** case is like the third, but it is written separately because for `i=1` we don't have c_{i-2}. In fact, it's the same formula as in the third case, but with 0 instead of c_{i-2}.

As usual in dynamic programming, we also need another auxiliary vector `whence` too keep track of the choices made at each step. This matrix will have size n and be made of integers. We will represent with `1` the "skip" choice, and with `2` the "pick" choice. As usual, the base case doesn't matter and we can set it to a sentinel value.

You should implement the forward pass based on the recursive relation first. It's a single for loop, and at the end the last element of `c` gives you the optimal cost. There is a small twist though, because of the special case for `i==1`. You could implement it separately, just like you do the base case `i==0`. But be careful, you also want your code to work in the case `n==1`. The best way to write all of this is to include the `i==1` case in the same loop that you use for all non-base-case values, but inside the loop check for the `i==1` case and use the suggestion that was given in the above explanation about what to do with the problematic c_{i-2} quantity.

After the forward pass, you'll need the backward one. You don't know in advance the size of the solution (what we called $\mathcal{S}$ above), so use a list and grow it in steps. What you want to do is: start from the last index, looking at the corresponding value of `whence`. If it is `2` ("pick") we add the index into the list, otherwise we don't. In any case, the value inside `whence` also tells us how to update the index, i.e. how much we should go back. We stop when we have reached the end of the list.

Some additional details and hints and extra tasks:

1. The returned list should be sorted in ascending order. Try to do it without explicitly using a sorting function.
2. You should implement a check. Write a function called `checksolution` which takes 3 arguments: `w`, `s` and `inds`. The purpose of this function is to ensure the consistency of the solution `(s, inds)` with the input `w`. There are two things to check:

  - The sum of the elements of `w` at the indices `inds` must be equal (within floating point precision) to `s`
  - The list `inds` must satisfy our constraint, i.e. it must be a *sorted* list of integers in which there are no consecutive indices (there is a gap of at least 1 between each two consecutive elements).

  This function should thus give an error if these conditions are not satisfied. To that end, you can use simple assertions.

  Try, if you can, to perform these checks without `for` loops, using numpy functions instead. One line of code for each is enough. **Hint:** have a look at the `np.diff` function.

3. (**Suggestion:** backup your code before attempting this task.) So far we have assumed a required minimum gap of `1` within the solution `inds`. Suppose we want to generalize the code to arbitrary (non-negative) gaps. For example:

   - if the input is `w = [14, 3, 27, 4, 5, 15, 1]` and the gap is `2`, our best choice is to pick `[27, 15]` and thus $\mathcal{S} = \{2, 5\}$ and $s = 42$
   - with the same `w` as above but a gap of `0`, we can just pick all elements.

   Add to both functions, `skipchoice` and `checksolution`, an optional argument called `gap`, with a default value of `1`. Then generalize the code such that you can deal with arbitrary gaps. There are surprisingly few modifications required, when you think about it.