exercises

Find the optimal skipped-choice via dynamic programming

Setup: the problem

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. In the directory called `"exercises"` you'll find a file called `"skipchoice.py"` that contains a partial implementation.

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

The code

Large parts of this code are already implemented in the function `skipchoice`, but the implementation is incomplete. Then there is a stub of a function `checksolution` that you will have to write. After that, there are a few test cases, followed by some definitions of test functions (that you can ignore), and at the very end of the file those test functions are called. The calls are commented out, except for `test0` (which passes). If you uncomment the call to `test1` it will produce an error, but solving task 1 will then fix it. The rest of the functions are associated with the respective tasks in the same way. Keep in mind however that a passing test is not a guarantee that the code is actually correct, and correspondingly that if some test X is failing it might be that the problem lies in the solution of a previous task Y.

Your tasks are described below. In the code, you can find them by looking for the string `"TASK"` in the comments. Do the task in their numeric order, not the order in which they appear in the code!

1. The second recursive relation is implemented separately in the code, between the base case `i=0` and the `for` loop for the general case for `i>=2`. This is ugly and furthermore it doesn’t work when `n==1`. Your task is to comment out this part of the code and treat this case within the `for` loop. **Hint:** the solution is in the explanation of the recursive relation: you’ll need to fix the issue with `i-2` which creates problems when `i==1`, but you’ll also need to pay attention to the range of the loop.

2. The path reconstruction code (the `while` loop at the end of the code) is incomplete. The `whence` vector is read correctly, so that the index `i` jumps to the right places (note how our choice of using `1` and `2` for the values of `whence` allowed some simplifications in the code here: this will be relevant for task 4!). However, the `inds` list (used to represent what we called $\mathcal{S}$ above) is not being computed. You need to do that, by filling it with those indices `i` that are being “picked” in the optimal solution. The result must be sorted in ascending order.

3. (This task is independent of 1 and 2.) The main code is complete with tasks 1 and 2. After the function `skipchoice` there is a definition of a function called `checksolution` which takes 3 arguments: `w`, `s` and `inds`. It currently does nothing, your task is to actually write it. This function must ensure the consistency of the solution `(s, inds)` with the input `w`. It must perform two checks:

   - The sum of the elements of `w` at the indices `inds` must sum 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 must give an error if these conditions are not satisfied. To that end, you can use simple assertions.

   For the full score, 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.

4. (**Suggestion:** backup your code before attempting this task.) So far, throughout the text and the code, we have assumed a required minimum gap of `1` within the solution `inds`. We now 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.

   Your task is to 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.

   This task has two tests associated with it, one for `skipchoice` and one for `checksolution`.

Solving the same problem with `minimize`

In the directory called `"exercises"` you'll find the file `"optim.py"`. We will use it to try to solve the same problem as before, but going through a continuous optimization procedure instead. So we are again given a vector w of length n. This time however, we will optimize over a real vector x of length n.

We want to minimize the following function over x:

$$f(x) = \sum_{i=0}^{n-1} x_i^2\,(x_i - 1)^2 - \sum_{i=0}^{n-1} w_i\, x_i + 5\sum_{i=0}^{n-2} (x_i\, x_{i+1})^2$$

Notice that the first term drives the entries x_i to go either towards 0 or 1. The second term drives x_i to be 1 for those i where w_i is large (notice the negative sign!). The last term picks pairs of consecutive terms and favors situations in which at least one of them is close to 0 (notice the different range here).

Therefore, this is basically a continuous version of the problem in the first part of the exam if we interpret x_i as being an indicator that the index i should belong to the set $\mathcal{S}$, i.e.: $x_i = 1$ should correspond to $i \in \mathcal{S}$ and $x_i = 0$ should correspond to $i \notin \mathcal{S}$.

Below here are your tasks for this exercise. If you complete them all you should see that the script performs a test and finds the optimal solution, and prints `"OK"`. In all the tasks, for the full score, don't use any for loops, only numpy operations.

1. Write the function `f`. Notice that in the code `f` has two inputs, `x` and `w`.
2. Write a function `skipchoicemini` that takes a vector w as input and calls `minimize` over `f`. Don't bother with argument checks, and you can use standard options for `minimize`. You should however choose a reasonable starting value `x0`, keeping in mind that we are looking for something where the entries are somewhere around 0 or 1. Then get the optimal value of `x` in a local variable called `x`.
3. In the same function, compute a binarized version of `x`, transforming it into a boolean mask `y`, as follows. Each entry y_i should be `True` if x_i is closer to 1 than it is to 0, and `False` otherwise. Then use y to compute $s = \sum_{i=0}^{n-1} w_i y_i$. Then compute a list (or better an array) called `inds` as the list of indices where `y` is `True`. Make the function return s and `inds`.

Theory questions

Answer on paper to the questions below.

1. Consider the cost function from the "minimize" exercise, that we rewrite here for convenience:

$$f(x) = \sum_{i=0}^{n-1} x_i^2\,(x_i - 1)^2 - \sum_{i=0}^{n-1} w_i\, x_i + 5\sum_{i=0}^{n-2} (x_i\, x_{i+1})^2$$

   - Given a generic point x, compute the gradient of f at x, i.e. for each index $i \in \{0, 1, \ldots, n-1\}$ express its component in the form

$$(\nabla f(x))_i = \ldots$$

     Notice that, while in the mathematical convention vector components' indexes start from 1, here we are assuming indexes to range from 0 to $n-1$ to match Python's 0-indexing convention.
   - Starting from a generic configuration x, express the configuration x' obtained after a step of gradient descent (with some step-size α) in terms of the x components.

2. A square island is represented as a grid of $n \times n$ points (i, j), with $0 \leq i < n$ and $0 \leq j < n$. At time zero, a person is standing on a point (i_0, j_0) inside the grid. Then time proceeds in steps, and at each step the person moves in any direction (left, right, up, down) uniformly at random. If they step outside the matrix,

they die.

- Say that we are interested in knowing the probability that they are alive after t steps. What method can you use to approximately measure its value via a simulation?
- Write down (on paper) a function (using Python or pseudo-code) that implements this method. The function should take $n, i_0,\ j_0$ and t as inputs, but can also take some extra parameters if needed.