Sorting an array by Simulated Annealing

You are provided with a directory called `"exercises"` which contains 3 files:

1. `"SimAnn.py"` is the generic program for Simulated Annealing that we have used for the lectures and the following exercises. You don't even really need to look at this if you recall how it works.
2. `"SortProbl.py"` contains a class that defines a problem to be optimized via Simulated Annealing. This is the main file that you will need to modify. See below.
3. `"sortproblrun.py"` is a test script. You can run it in Spyder and it will create a problem and then try to optimize it.

The class definition in `"SortProbl.py"` creates a random 1-d array of some given length `n`, and it defines a cost function by just counting the number of times in which two adjacent elements are in the wrong order.

For example, in this array there are three mis-orderings (the ones marked, since 0.5 > 0.4 etc.):

```
v = [0.1, 0.5, 0.4, 0.3, 0.6, 0.2, 0.8, 1.0]
           └──┘ └──┘      └──┘
```

The cost here is `3`. The minimum cost of `0` obviously corresponds to a sorted array. Therefore, one can try to sort an array by Simulated Annealing (which is a terrible idea of course, there are much better ways to sort an array as you know).

The class already has most methods defined. In particular, it has the constructor, and the implementation of the cost function. If you try to run the script `"sortproblrun.py"`, however, it won't do anything useful: the costs will be fixed and the acceptance rates will all be `1.0`. You need to complete/fix the implementation.

Your tasks are as described below (see also the note at the end). In the code, you will find the pieces of code that you need to change by looking for comments that start with "TASK".

1. In the constructor, there is a `seed` argument that is set to `None` by default, and it isn't used. You should make sure that, if the user passes it and it isn't `None` (as in `"sortproblrun.py"`), it will be used to seed numpy's random number generator. To test it, run the `"SortProbl.py"` file and check that if you construct a `SortProbl` object its contents are determined by the seed that you pass to the constructor.
2. The moves consist of producing 2 indices, `i` and `j`, each between `0` and `n-1`, and attempting to swap them, see the `propose_move` method. The `accept_move` method is only half-written: complete it. (This task is crucial for the rest!)
3. If you performed task 2, your code should now at least be doing something: you should see some changes in the acceptance rates. It's still not working properly though. There is a subtle mistake in the `propose_move` method, such that, at times, the proposed move will always be accepted no matter what. Identify the issue and fix it. After this, the code should be working correctly (i.e. the `simann` algorithm works as intended).
4. Change the values of the parameters to the `simann` function in the test script `"sortproblrun.py"`, such that it finds a solution (in a reasonable time, a few seconds at most!) with the given problem size (i.e. don't touch the `n` parameter passed to the `SortProbl` constructor). The parameters should make sense in the context of the simulated annealing algorithm: if you test different seeds in the constructor of `SortProbl`, it should then work most of the times (say, 80%), and at least get to at most `cost=1` basically every time.
5. The `compute_delta_cost` method uses a very inefficient method, and the efficient one is commented out. Comment out the inefficient part (select it all and press "Ctrl+1" in Spyder) and uncomment the other one (also "Ctrl+1"). Now the test script will fail again with an error. This method should compute the cost difference

associated with a proposed move, in order to decide whether to accept the move. However, it misses one part of the computation, plus there is a bug:

- There is a `## TASK 5a: implement this` comment that shows you where you should add a missing computation.
- There is a `## TASK 5b: fix this` comment that shows you which part of the code contains a bug that you should fix.

If you do both tasks, the test code will run again. At this point (and *only* at this point) you could then set the option `debug_delta_cost=False` in the test script.

6. The function that computes the cost can be improved using numpy's constructs (indexing, functions, broadcasting rules…). Write the cost computation without the `for` loop, in one single and efficient line of code.

   Important tip: While you experiment, keep both the old and new codes, use different variable names and use `assert` to make sure that the two computations give the same result. Only when you're sure it works, remove or comment-out the inefficient version and the assertion.

Note: you should try to do the tasks in order. However: tasks 1 and 6 stand on their own; the others can in principle be done independently, but it's harder. Task 2 is absolutely required, task 4 depends on 3, task 5 could be done before 3 and 4.