CSCI B505 Spring 20: Programming assignment 3

Due online: March 15, 11:59pm EST

Submit your work via Canvas, using "File upload". All work is strictly individual. If you have difficulties, please ask AI's for help during their office hours.

What to submit

You should submit a single Python or Java file. The file should contain the following:

- Methods best_share_sort **(4 points)** and best_share_dp **(13 points)** as described below. You must implement both methods.
- Runtime and memory complexity for each method **(1 point for each method)**. You should specify them in the doc comments of the methods.
- An exhaustive set of tests **(1 point)**.

Problem

Alice and Bob have n items, where n is even, and they would like to equally share the items between them: both Alice and Bob will receive $n/2$ items each. For i-th item we know value a_i, which represents how happy Alice is to get this item. Similarly, b_i represents how happy Bob is to get the item.

Your task is to maximize the total happiness. You should find a set of items which should be assigned to Alice. Formally, for a set of items I, you should find A^* such that:

$$A^* = \underset{A \subseteq I:\ |A| = n/2}{\arg\max} \left(\sum_{i \in A} a_i + \sum_{i \in I \setminus A} b_i \right)$$

Items can be returned in an arbitrary order. If there are multiple solutions, return any of them.

You should create two methods, best_share_sort and best_share_dp, each solving the problem as described below. They should have the same signature; for best_share_sort, the signature is (for Python)

```
def best_share_sort(a: List[int], b: List[int]) -> List[int]
```

or (for Java)

```
int[] best_share_sort(int[] a, int[b])
```

Items are numbered from 0 to $n-1$. It's guaranteed that $0 \leq n \leq 10^3$ and $0 \leq a_i, b_i \leq 10^5$ for all i.

See Table 1 for examples.

Input	Output
$[2, 1]$, $[1, 2]$	$[0]$ (items are 0-based)
$[10, 20, 30, 40]$, $[8, 18, 25, 35]$	$[2, 3]$
$[10, 10, 10, 10]$, $[7, 9, 11, 13]$	$[0, 1]$

Table 1: Examples

Sorting Solution (4+1 points)

Let's rewrite our expression:

$$\sum_{i \in A} a_i + \sum_{i \in I \setminus A} b_i = \sum_{i \in A} (a_i - b_i) + \sum_{i \in I} b_i$$

The second term is a constant, and therefore we should only optimize the first term. The first term is maximized when we select items with largest $a_i - b_i$. Therefore, we should select $^n/_2$ items with largest $a_i - b_i$.

You should implement method best_share_sort which uses this idea.

In the method's doc comment, please report the running time and memory complexity of the algorithm (1 point).

Dynamic programming solution (13+1 points)

Dynamic programming solution is similar to the one of the dice problem from the midterm. You should implement **one** of the following two algorithms.

1. Let $dp[i][j]$ denote the best total happiness which can be obtained after we have processed first i items $(0, \dots, i-1)$, among which Alice got j items (therefore, Bob got $i - j$ items). We process items one-by-one: we first compute $dp[1][j]$ for all j, then $dp[2][j]$ for all j, etc. When computing $dp[i][j]$, we have a choice what to do with the last item. *A technical detail*: since items are numbered from 0, the last (i-th) item has number $i - 1$; i.e., when computing $dp[i][j]$, we use $a[i-1]$ and $b[i-1]$ instead of $a[i]$ and $b[i]$ as one may expect. There are two options:

   - We can assign item number $(i - 1)$ to Alice, getting $a[i - 1]$ happiness. Alice has to select $j - 1$ items from the first $i - 1$ items, and the maximum happiness from doing this is computed in $dp[i-1][j-1]$.
   - We can assign item number $(i - 1)$ to Bob, getting $b[i - 1]$ happiness. Alice has to select j items from the first $i - 1$ items, and the maximum happiness from doing this is computed in $dp[i-1][j]$.

   We select the best of two options.

2. The idea is very similar, but now $dp[i][j]$ represents the maximum total happiness after items $i, \dots, n-1$ are processed, among which Alice got j items. Computation goes from right to left, i.e. to compute $dp[i][\cdot]$ you compute all $dp[i + 1][\cdot]$ first. A good thing about this approach is that when computing $dp[i][\cdot]$, you use $a[i]$ and $b[i]$, i,e, indices are natural. While it may seem as a small change, it actually makes implementing the algorithm much more comfortable.

It's up to you which approach to select. In method best_share_dp, you should implement a **bottom-up** dynamic programming solution based on one of the described recurrences. In the method's doc comment, please report the running time and memory complexity of the algorithm (1 point).

Tests (1 point)

Please test your solution with an exhaustive set of tests.