Problem 1. You are a freelancer who has to choose which jobs to take on each of n days. On each day $i \in \{1, 2, \ldots, n\}$ you have three options — take a *light* job, take a *heavy* job, or take no job. Your pay for the day i is ℓ_i if you take a light job, h_i if you take a heavy job, and nothing if you take no job. There is one additional constraint: for each day $i \in \{2, 3, \ldots, n\}$, you can take a heavy job on day i only if you took no job in day $i - 1$. (You can take a heavy job on day 1.)

Fix an input $(\ell = (\ell_1, \ldots, \ell_n), h = (h_1, \ldots, h_n))$.

For each $j \in \{0, 1, 2, \ldots, n\}$, a *schedule* for days $\{1, 2, \ldots, j\}$ selects, for each of those j days, one of the three job options (light, heavy, none). The schedule is *feasible* if, for every day $i \in \{2, 3, \ldots, j\}$, if the schedule selects the heavy job for day i, then it selects no job for day $i - 1$. The *earnings* of the schedule is the sum of its earnings for each day $i \in \{1, 2, \ldots, j\}$, where the earnings for day i is ℓ_i if the schedule selects a light job that day, h_i if the schedule selects a heavy job, and nothing if it selects no job. (You may assume ℓ_i and h_i are non-negative, but *not* that $\ell_i \leq h_i$.) Two schedules are *distinct* if they are not identical (i.e., for at least one day, their selections differ).

For $j \in \{0, 1, 2, \ldots, n\}$, define $N(j)$ to be the number of distinct feasible schedules for days $\{1, \ldots, j\}$. Define $M(j)$ to be the maximum, over all feasible schedule for days $\{1, \ldots, j\}$, of the total earnings of the schedule.

(a) Give a recurrence relation for $N(j)$ in terms of $N(1), \ldots, N(j-1)$.

(b) Explain precisely in at most four lines why your recurrence relation is correct.

(c) Give a recurrence relation for $M(j)$ in terms of $M(1), \ldots, M(j-1)$, ℓ, and h.

(d) Explain precisely in at most four lines why your recurrence relation is correct.

(e) What is the asymptotic worst-case running time of the resulting (iterative, or recursive and memoized) algorithm for computing $M(n)$, given (ℓ, h)?

(f) Explain precisely in at most three lines why the bound you give is correct.

(g) Give pseudo-code for the resulting iterative dynamic-programming algorithm, for computing $M(n)$, given (ℓ, h).

For readability, use mathematical expressions (such as a sums, maximums, etc.) that any competent programmer could implement in an obvious way, without giving the details of how to compute those expressions.

(continued)

Problem 1 answer

(a) Here is the recurrence relation for $N(j)$:

$$N(j) = \begin{cases} 1 & \text{if } j = 0 \\ 3 & \text{if } j = 1 \\ N(j-2) + 2N(j-1), & \text{if } j \geq 2. \end{cases}$$

(b) Here is why the above recurrence relation holds:
If j is 0, the schedule can only be no jobs (1 option), if it's 1, we have 3 choices for the first day (light, heavy, none), and anything two or above, we add the number of schedules where the given day was heavy (this means no job day before so we recurse two days back as the day before is already set) and the number if it wasn't heavy, in which we have 2 cases (light and none). .

(c) Here is the recurrence relation for $M(j)$:

$$M(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left(\ell_j, h_j, 0\right) & \text{if } j = 1 \\ \max\left(M(j-2) + h_j, M(j-1) + \ell_j, M(j-1)\right), & \text{if } j \geq 2. \end{cases}$$

(d) Here is why the above recurrence relation holds:
You make no money on 0 days, you max of light, heavy, none pay (0 pay on none) for the first day (as no restrictions), and after you take the max of the heavy case plus heavy pay plus 0 (no work day before, recurse on -2), light case plus light pay, or no pay just recurses 1 day back (will always be less than light case unless ℓ_j is negative, but we leave it there as we don't want to prove this).

(e) The asymptotic worst-case running time is

$$\Theta(n).$$

(f) Here is why the above bound holds:
J ranges from 0 to n, so there are n subproblems for the memoization. Each subproblem takes constant time, as the only operations are max and addition and subtraction. Thus the total time for n subproblems of constant time is $\Theta(n)$.

(g) Here is the resulting iterative dynamic-programming algorithm:

freeLancer$(\ell = (\ell_1, \ldots, \ell_n), h = (h_1, \ldots, h_n))$: — *iterative version*

1. for $j = 0, 1, 2, \ldots, n$
2. set $$M(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left(\ell_j, h_j, 0\right) & \text{if } j = 1 \\ \max\left(M(j-2) + h_j, M(j-1) + \ell_j, M(j-1)\right), & \text{if } j \geq 2. \end{cases}$$
3. return M[n].

Problem 2. Consider the following problem:

input: A universe U (a set) and sequence $X = (x[1], x[2], \ldots, x[n])$, where each $x[i]$ is in U.

output: The number of *valid* subsets S of $\{1, 2, \ldots, n\}$, where S is valid if the corresponding subsequence of X has no adjacent equal elements. For example, if $U = \{a, b\}$ and $X = a, b, b, a$, then 11 of the 16 possible subsets are valid:

subset	subsequence	valid?	subset	subsequence	valid?
$\{1,2,3,4\}$	a,b,b,a	no	$\{1,2,3\}$	$a,b,b,-$	no
$\{2,3,4\}$	$-,b,b,a$	no	$\{2,3\}$	$-,b,b,-$	no
$\{1,3,4\}$	$a,-,b,a$	yes	$\{1,3\}$	$a,-,b,-$	yes
$\{1,2,4\}$	$a,b,-,a$	yes	$\{3\}$	$-,-,b,-$	yes
$\{3,4\}$	$-,-,b,a$	yes	$\{1,2\}$	$a,b,-,-$	yes
$\{2,4\}$	$-,b,-,a$	yes	$\{2\}$	$-,b,-,-$	yes
$\{1,4\}$	$a,-,-,a$	no	$\{1\}$	$a,-,-,-$	yes
$\{4\}$	$-,-,-,a$	yes	$\{\}$	$-,-,-,-$	yes

(If all elements of X are the same, there are $n+1$ valid subsets. If all are distinct, there are 2^n.)

A dynamic-programming algorithm (with one small bug that you must find). Fix an input $X = x[1], \ldots, x[n]$. Assume for notational convenience that U contains some element that is not in X. (This is without loss of generality; otherwise, just add an artificial element to U.) Let $\star$ denote this element.

The subproblems to be solved. For each $i \in \{0, 1, \ldots, n\}$ and $u \in U$, define $V(i, u)$ to be the collection of valid subsets of $\{1, 2, \ldots, i\}$ such that the corresponding subsequence of X ends with an element *other than* u. The desired output, for input X, is $|V(n, \star)|$.

The recurrence relation?

Lemma 1. *For all $u \in U$ and each $i \in \{0, 1, \ldots, n\}$,*

$$|V(i,u)| = \begin{cases} 1 & \text{if } i = 0, \\ |V(i-1,u)| & \text{if } i > 0 \text{ and } x[i] = u, \\ |V(i-1,u)| + |V(i-1,\star)| & \text{if } i > 0 \text{ and } x[i] \neq u. \end{cases}$$

Proof sketch? Consider any $u \in U$ and $i \in \{0, 1, \ldots, n\}$. For $i = 0$, we have $V(0, u) = \{\emptyset\}$, so $|V(0, u)| = 1$. So consider the remaining case, $i \geq 1$.

In the subcase that $x[i] = u$, index i is not in any subset in $V(i, u)$, so $V(i, u)$ is just $V(i-1, u)$, and the recurrence holds.

In the remaining subcase, $x[i] \neq u$. Partition the subsets S in $V(i, u)$ into two types: those that don't contain element i, and those that do. Those that don't are just the subsets in $V(i-1, u)$. Those that do are of the form $S \cup \{i\}$ where S is a subset in $V(i-1, \star)$. Thus, the total number of subsets in $V(i, u)$ is $|V(i-1, u)| + |V(i-1, \star)|$, which equals the right-hand side of the recurrence (in the subcase $x[i] \neq u$). ?

(continued)

The algorithm. This gives the following iterative algorithm:

def countV$(U, X = (x[1], x[2], \ldots, x[n]))$: *(assume U contains some element $\star \notin X$)*

1. for $i = 0, 1, 2, \ldots, n$:
2. for $u \in U$:
3. $$v[i,u] = \begin{cases} 1 & \text{if } i = 0, \\ v[i-1,u] & \text{if } i > 0 \text{ and } x[i] = u, \\ v[i-1,u] + v[i-1,\star] & \text{if } i > 0 \text{ and } x[i] \neq u. \end{cases}$$
4. return $v[n, \star]$

Run time. By inspection, the running time is dominated by the time for the additions. There are $O(n|U|)$ additions — at most one for each subproblem. The integers that arise can be large, but no larger than 2^n, so can be represented in $O(n)$ bits, so each addition can be done in $O(n)$ time. Hence, the algorithm runs in time $O(n^2|U|)$.

Your task: fix the bug! Given the example input where $U = \{a, b, \star\}$ and $X = (a, b, b, a)$, the algorithm sets the array v as shown to the right, then outputs 16. This is wrong! As discussed at the start, the correct output for this example is 11.

i	$x[i]$	$v[i,a]$	$v[i,b]$	$v[i,\star]$
0		1	1	1
1	a	1	2	2
2	b	3	2	4
3	b	7	2	8
4	a	7	10	16

(a) What is the smallest i for which, on the example input, $v[i, \star]$ as computed is not $|V(i, \star)|$?

(b) What is the set $V(i, \star)$ for that i?

(c) Give a corrected recurrence relation, making as few changes as possible.

(d) Simulate the corrected algorithm on the example. Show the resulting array v.

(e) Give pseudo-code for the correct algorithm, making as few changes as possible.

Problem 2 answer

(a) The smallest i for which $v[i, \star]$ does not equal $|V(i, \star)|$ is

$$i = 3$$

(b) The set $V(i, \star)$ for that i is

$$V(i, \star) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}\}$$

(continued)

(c) The correct recurrence relation is

$$|V(i,u)| = \begin{cases} 1 & \text{if } i = 0, \\ |V(i-1,u)| & \text{if } i > 0 \text{ and } x[i] = u, \\ |V(i-1,u)| + |V(i-1,x[i])| & \text{if } i > 0 \text{ and } x[i] \neq u. \end{cases}$$

(d) Here is the array v as computed by the correct algorithm:

i	$x[i]$	$v[i,a]$	$v[i,b]$	$v[i,\star]$
0		1	1	1
1	a	1	2	2
2	b	3	2	4
3	b	5	2	6
4	a	5	7	11

(e) Here is pseudo-code for the correct algorithm:

def countV$(U, X = (x[1], x[2], \ldots, x[n]))$: *(assume U contains some element $\star \notin X$)*

1. for $i = 0, 1, 2, \ldots, n$:
2. for $u \in U$:
3. $$v[i,u] = \begin{cases} 1 & \text{if } i = 0, \\ v[i-1,u] & \text{if } i > 0 \text{ and } x[i] = u, \\ v[i-1,u] + v[i-1,x[i]] & \text{if } i > 0 \text{ and } x[i] \neq u. \end{cases}$$
4. return $v[n,\star]$

Problem 3. Design a dynamic-programming algorithm for the following problem:

input: A sequence $(a_1, a_2, \ldots, a_n)$ of n non-negative numbers, defining the following "Game of Piles".

The game starts with a row of n piles $1, 2, \ldots, n$ of coins, where pile i contains coins of total value a_i. Alice and Bob take turns removing either the first or last of the remaining piles, and placing it in their personal stash, until there no piles remain. (If Alice moves pile 1 into her stash, then Bob can move either pile 2 or pile n into his, and so on.) Alice's final score is the total value of her stash, minus the total value of Bob's. Bob's final score is the total value of his personal stash, minus the total value of Alice's. (For example, suppose $a = (10, 2, 8)$. Alice can take the first pile to guarantee a score of at least $10 + 2 - 8 = 4$. But if she takes the last pile she could only guarantee a score of at least $8 + 2 - 10 = 0$.)

output: The maximum score that the first player can guarantee for his or herself, assuming the opponent plays optimally to maximize his score. Each player's score is the negation of the opponents score, so by playing to maximize his or her own score, each player is also playing to minimize the opponent's score.

(a) Define the subproblems your algorithm solves. Which is the final answer?

(b) State an appropriate recurrence relation, including boundary cases.

(c) Explain clearly in at most five lines why the recurrence relation is correct.

(d) Give the big-Θ run-time of the resulting algorithm (iterative, or recursive and memoized).

(e) Explain clearly in at most three lines why this bound holds.

(f) Give pseudo-code for an iterative implementation of the resulting algorithm.

(g) Submit your algorithm to the Hacker-Rank challenge, your submission should pass all tests. What is the username of your Hacker-Rank submission?

Clarification of what it means to "play optimally". Consider the game with $a = (2, 2, 5, 3)$, with Alice as first player.

Option 1: Suppose Alice starts by taking the first pile (value 2).

She'll score 2 for that, and the game that remains will be 2, 5, 3, with Bob as first player. Considering just that game 2, 5, 3 — whatever Bob does, Alice will take the 5, so Bob will end up with the 2 and the 3. So Alice will score $5 - 2 - 3 = 0$ for this sub-game. So, if Alice starts the game $2, 2, 5, 3$ by taking the 2, her total score for the game will be $2 + 0 = 2$.

Option 2: Suppose Alice starts by taking the last pile (value 3).

She'll score 3 for that, and the game that remains will be 2, 2, 5, with Bob as first player. Considering just that game $2, 2, 5$ — Bob should take the 5, then Alice will take a 2, and Bob will take the other 2. So Alice will score $2 - 5 - 2 = -5$ for this sub-game. So, if Alice starts the game $2, 2, 5, 3$ by taking the 3, her total score for the game will be $3 - 5 = -2$.

The optimal first move for Alice is to take the first pile (value 2). This maximizes her score, assuming both players play the remaining game optimally. This kind of reasoning defines "optimal play" for the general case. Assuming we know what optimal play is for any game with, say, $n - 1$ piles, we can inductively define what optimal play is for any game with n piles. Namely, the first player chooses the first move to maximize his or her resulting total score, assuming that after the first move, both players optimally play whatever game (on $n - 1$ piles) remains.

Warning: you might be tempted to try a greedy algorithm that chooses the first move just by looking at just a few piles near each end. This won't work. You'll need to bring to bear the full power of dynamic programming!

Problem 3 answer

(a) Here is a definition of all subproblems that the algorithm solves, given $(a_1, a_2, \ldots, a_n)$.
For i $\in 0, 1, 2, ..., n$, j $\in 0, 1, 2, ..., n$, Define S(i, j) = The maximum score the first player can guarantee for themselves from pile i to pile j $(a_i, a_i + 1..., a_j)$. Goal: S(1, n)
The final answer is $S(1, n)$.

(b) Here is the recurrence relation:

$$S(i,j) = \begin{cases} 0 & \text{if } j < i, \\ \text{a}_i & \text{if } j = i, \\ \max\left(a_j - S(i, j-1), a_i - S(i+1, j)\right) & \text{if } j > i \end{cases}$$

(c) Here is why the recurrence relation is correct:
If i > j, return 0 (start after end); if j = i, one pile left, return its value (has to be chosen by whoever playing the turn); else i < j: 2 cases: where the first player picks the last pile, in which you must subtract the second player playing optimally for the rest of the piles (total score for player 1). Do the same thing if player chose the first pile, and take the maximum score.

(d) The big-Θ running time of the resulting algorithm is

$$\Theta(n^2).$$

(e) Here is why the bound above holds:
i ranges from 0 to n, j ranges from 0 to n, so there are n^2 subproblems, and each of those problems take constant time for addition and subtraction and maximum. Thus the total time for n^2 subproblems of constant time it $\Theta(n^2)$.

(f) Here is pseudo-code for the iterative implementation of the resulting algorithm.

def maxPilesScorePlayerOne$(a_1, a_2, \ldots, a_n)$:
1. for $i = n, n-1, n-2, \ldots, 1$:
2. for $j = i, i+1, \ldots, n$:
3. $$S(i,j) = \begin{cases} 0 & \text{if } j < i, \\ \text{a}_i & \text{if } j = i, \\ \max\left(a_j - S(i, j-1), a_i - S(i+1, j)\right) & \text{if } j > i \end{cases}$$
4. return $S(1, n)$

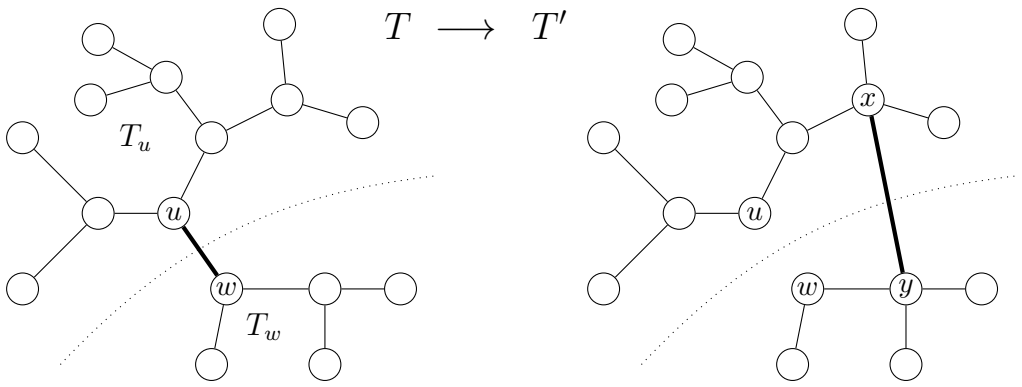
(g) The username for my Hacker-Rank submission is sisolta75 .

Problem 4. Here is a greedy Minimum Spanning Tree algorithm:

cycleBreaker($G = (V, E)$): — *G is a connected, undirected, edge-weighted graph*
1. initialize $E' = E$
2. while the edges in E' contain a cycle:
3.1. choose any cycle C in E', and let (u, w) be an edge of maximum weight in C
3.2. delete (u, w) from E'
4. return the spanning tree formed by the edges in E'

Complete the incomplete proof of correctness for this algorithm that is given in the template. You may wish to review the proof of correctness of Prim's MST algorithm. We will use the following observation:

Observation 7. *Let (u, w) be any edge in any spanning tree T of an undirected graph $G = (V, E)$. Removing (u, w) splits T into two connected subtrees T_u and T_w (see below). Let (x, y) be any edge in E with $x \in T_u$ and $y \in T_w$. Then removing (u, w) from T and adding (x, y) gives a spanning tree $T' = \{(x, y)\} \cup T \setminus \{(u, w)\}$ of G.*



Problem 4 answer

Lemma 2. *In any execution on a graph as described in the problem statement,* cycleBreaker *maintains the following invariant:*

the edge set E' contains some minimum spanning tree of G.

Proof (long form).

1. Consider any execution of the algorithm on some connected graph $G = (V, E)$.
2. At the start of the first iteration, $E' = E$.
3. Since G is connected, it has some MST, so the invariant holds initially.
4. Next we show that each iteration maintains the invariant.

5.1. Consider any iteration such that the invariant holds at the start of the iteration.
5.2. Let E' denote the set E' at the start of the iteration.
5.3. Let C and $(u, w) \in C$ be the cycle and edge chosen in the iteration.
5.4. So the weight of (u, w) is as large as the weight of any other edge in C.
5.5. And $E' \setminus \{(u, w)\}$ is the set of edges remaining after the iteration.
5.6. The invariant holds at the start of the iteration, so E' contains some MST, say T, of G.
5.7.1. Case 1. Suppose T doesn't contain (u, w).
5.7.2. Then T is also a subtree of $E' \setminus \{(u, w)\}$, so the invariant holds at the end of the iteration.
5.8.1. Case 2. Otherwise T contains (u, w).
5.8.2. Let $P = C \setminus \{(u, w)\}$ be the path from u to w formed by removing (u, w) from the cycle C.
5.8.3. By Step 5.4 the weight of each edge in P is at most the weight of (u, w).
5.8.4. Let (x, y) be the edge in P that is not in T.
5.8.5. This edge must exist, as (u, w) was the edge in T that completed the path between vertices u and w, so (x, y) can't also be in T, as this would create the cycle C, making the spanning tree T invalid.
5.8.6. Additionally, once edge (u, w) is removed, the path between u and w in T is broken, as T was a spanning tree, and for a spanning tree to exist, a path between u and w must exist, so we can create this path by adding (x, y) to T, which creates path P between u and w.
5.8.7. So, by Observation 7, adding (x, y) to T and removing (u, w) gives the spanning tree

$$T' = \{(x, y)\} \cup T \setminus \{(u, w)\}.$$

5.8.8. Edge (x,y) and edge (u,w) are both edges in C, and (u,w) is the maximum weight edge in C as chosen by the algorithm.
5.8.9. So $\mathsf{wt}(x, y) \leq \mathsf{wt}(u, w)$. This implies that $\mathsf{wt}(T') = \mathsf{wt}(T) + \mathsf{wt}(x, y) - \mathsf{wt}(u, w) \leq \mathsf{wt}(T)$.
5.8.10. So T' is also a minimum-weight spanning tree of G.
5.8.11. Edge (x, y) is in P, which is in $E' \setminus \{(u, w)\}$, so T' is in $E' \setminus \{(u, w)\}$.
5.8.12. So, at the end of the iteration, the new edge set $E' \setminus \{(u, w)\}$ contains the MST T'.
5.8.13. So the invariant holds at the end of the iteration.
5.9. By Cases 1 and 2 above, the invariant holds at the end of the iteration.

6. As observed previously, the invariant holds at the start of the first iteration.
7. By Block 5, each iteration maintains the invariant. So it holds throughout. □

(continued)

Next we use the invariant to prove correctness.

Theorem 1. *Given any graph as described in the problem statement,* cycleBreaker *returns a minimum spanning tree.*

Proof (long form).

1. Consider any execution of the algorithm on any input G.
2. By Lemma 2, the invariant holds after the last iteration of the algorithm.
3. So the final edge set E' contains some minimum spanning tree T of G.
4. Since E' contains no cycles, it must contain only the edges of T
   (as any edge (u, w) not in T would form a cycle with the u-to-w path in T).
5. That is, the edges of E' form a minimum spanning tree. □