Please enter your name and uID below.

Name: Devin White

uID: u0775759

Collaborators, if any, and how you collaborated:

Submission notes

- Due at 11:59 pm (midnight) on Thursday, Sept 22.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for this problem set, you are allowed to collaborate in detail with your peers, as long as you cite them. However, you must write up your own solution, alone, from memory. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Wall backtracking)

   (a) **Base Cases**

   - MinWall(n+1,0) should return 0. There are no exercises to be done because n+1 is out of bounds, and it is already at the ground. So it should return 0, as there is no wall to scale;
   - MinWall(n+1,10) should return ∞ or rather, our problem's equivalent of infinity which is 1000There are no exercises to be done and they are not at ground level, making getting to the ground impossible.

   (b) **Recursive cases for MinWall(i,h);**

   - If choosing to climb down, then you move on to the next "wall"(by incrementing the current index) and the current height needs to be subtracted by the height of D[i+1]. Which you will then recurse as $MinWall(i+1, n-D[i+1])$ Esentially, now you need to figure out the shortest path from i+1 with a starting height of n-(height of i)
   - If choosing to climb up, then you move on to the next "wall"(by incrementing the current index)and the current height needs to be added to the height of D[i+1]. Which you will then recurse as $MinWall(i+1, n+D[i+1])$ Esentially, now you need to figure out the shortest path from i+1 with a starting height of n+(height of i)
   - if $h < D[i]$ and you are close to the ground, then the only recursive call you should need to make is the one to go Up, as there is no possible path to the ground since you can't go spelunking underneath. it should return $MinWall(i+1, n+D[i+1])$
   - in the general case, we want to check BOTH the UP and DOWN paths in their, and take the Maximum value found down each of those paths(with each of those paths deciding between 2 paths),we then compare the two maximum values found down each branching path, then take the minimum out of those 2 and that gives us our final answer. it looks something like:
     - $up = MinWall(index+1, n+D[i+1])$
     - $down = MinWall(index+1, n-D[i+1])$
     - $returnMin(up, down)$

   (c) MinWall(i,h)=
   $$\begin{cases} 0 & i > n \text{ and } h = 0 \\ 100000 & i > n \text{ and } h > 0 \\ Minwall(i+1, n+D[i+1]) & h < D[i+1] \\ min \begin{cases} max(Minwall(i+1, n-D[i+1]) & \\ max(Minwall(i+1, n+D[i+1]) & otherwise \end{cases} & \end{cases}$$

   Eseentially, this algorithm is saying concisely what is said above. If $index > n$(length of array) and you're at the ground, it returns 0. You already completed. If $index > n$ and you are above ground, that route is impossible(and maybe the whole things). if h(the height) is less than the value of height in the Distances array, then we have to go up with the given recurrence. Otherwise, we take the minimum value of the maximum value of the 2 branching paths/choices, and that is our optimal path!

   (d) The **Main** call to MinWall should be $MinWall(0, 0)$.With the 0-th index of the array indicating to do all instructions starting with the first(every instruction inside the array), and the 0 specifying that we are starting at the ground level. This gives us the shortest wall height needed from ground to ground

2. (Wall dynamic programming)

   (a) There needs to be enough columns to account for all the possible sub-problems, while there only needs to be enough rows to account for the size of the original array, one representing each "value" in side it, you need a TON more columns to account for all the mix-and-match possibilities that will be inside the table. The total size of the structure comes out to something like [n+1][H] (Where n is the total size of the "exercise" array. and H is the total sum of all the distances in the array)This is to account for all possible routes one can take at each "intersection" alongside making room for the results of the 2 possible base cases(in this particular case, we reached our destination or we cannot reach our destination, so 0 or infinity). Another possibility if you feel like being less optimal is to hard-code the Array size to be [n+1][$j > 1000$] or so, since the original programming problem specifies that there will never be a maximum height greater than 1000 with any set of inputs.

   (b) In theory, The storage array should be filled initially your known base cases. in this case, with 0 in some corner, and the infinity filling out the rest. Whether you fill out i with large values of i or small values of i doesn't really matter in the grand scheme of things, it just depends on the problem. In this case filling up i starting from 0 makes the most sense as it's the easiest to wrap my head around converting the recursive algorithm created in problem 1.

   (c) It doesn't matter the order in which you fill h, it depends entirely on the problem and which order you decide to fill i. So either is ok. In this case it makes sense to fill h starting with the smaller values(starting from h=0) because it more closely represents the original recursive formula and is easier to wrap my mind around. That said, you should ALWAYS fill out the base cases first, as they are your means of filling the table

   (d) The outer loop should iterate over i, while the inner loop should iterate over h. The reason the outer loop should iterate over i is that you want the results to be stored in column major(in whichever order makes the most sense for the given problem) order so that the possibilities of heights for a given index are mapped closely to the applicable areas, this allows results to be grabbed for calculation within the current cells which allows us to find answers to subsequent sub problems eventually culminating in the answer to find out final solution.

   (e) Depends on the order i and h are iterated through(n to 0 or 0 to n). But the final answer should be something along the lines of in [n,0], but this depends entirely on the order in which the table is filled up. In this case, [n,0] is the area where the final answer should be.

   (f) As this algorithm is running a nested loop in order to populate its tables, it should run in approximately $O(n^2)$, or rather, in this case $O(N * H)$, as the size of the columns may be drastically larger than the size of the rows. aside from that, the only other costs to this algorithm are constants, with arithmetic and pulling values from the table. This is a MASSIVE increase in speed compared to the obscenely slow Recursive method, which runs in something like $O(2^n)$

3. (Card game)

$$\text{Count(C[0..i],Ch,HIC,LoC,S)}= \begin{cases} 0 & i = n \text{ or } n = 0 \\ Score - C[i] & \text{If } Hi \text{ and } C[i] < 0 \\ Score - C[i] & \text{else If } LO \text{ and } C[i] > 0 \\ Score + C[i] & \text{else If } LO \text{ and } C[i] < 0 \\ Score + C[i] & otherwise \\ Count(C[i+1], LO, HIC, LOC, S) & \text{If } HiC = 10 \\ Count(C[i+1], HI, HIC, LOC, S) & \text{else If } LOC = 10 \\ max \begin{cases} Count(C[i+1], Hi, HIC, LOC, S) \\ Count(C[i+1], Lo, HIC, LOC, S) \end{cases} & Otherwise \end{cases}$$

- The base case(s):
  - If $C[i]Size <= 0$ then it returns 0.
  - If $i = n$(the maximum index of $n + 1$) it also returns 0
    These are the base cases because n+1 and no cards are not valid, so the maximum possible score for these is 0;
- The If statements to calculate score:
  - If HI and $C[i] < 0$ reduce the score by C[i]
  - If LO and $C[i] < 0$ increase the score by C[i]
  - If HI and $C[i] > 0$ increase the score by C[i]
  - If LO and $C[i] > 0$ reduce the score by C[I]
    These can be if statements or some equal alternative, the reason this is laid out like this instead of a much more clever way is because 1) laziness, and secondly, because we don't need to be clever. This algorithm is greedily checking(greed is good. Right?) every possible path, so it doesn't matter if we can see numbers on the cards, some branch is going to select every single possibility anyway, so we simply need to increase or decrease the score accordingly.
- The recursive statements:
  - If HiCount = 10 take the low path. This is self explanatory. we can't take the same path more than 10 times so we have a condition to switch down our endlessly large check every path algorithm.
  - If LoCount = 10 take the low path. see above...but lower
    Now the question you may ask is: How are you counting these? While recursion and global variables usually do not mix, since we're feeling extremely spicy and NOT clever today, we can simply pass the 2 counters on each recursive call, and increment them once inside any of the score altering if statements, resetting them in the recursion if statement once either reaches 10. The increments and decrements and resetting were not included because it makes the recursion formula EXTREMELY messy.
  - $$max \begin{cases} Count(C[i+1], Hi, HIC, LOC, S) \\ Count(C[i+1], Lo, HIC, LOC, S) \end{cases} \quad Otherwise$$

    This is simply saying to take both paths, and to return the value that is larger. This is the key to make the algorithm work, as this is going to be the branch primarily happening when HiCount or LoCount are below 10, and should return the highest possible value at the top level(if it works like I think it does)

The recursive call to find the maximum possible score would be Max(count(C[0....i],lO,0,0),count(C[0..i],HI,0,0,0)) This Max in the original call is to account for the fact that the original call doesn't have the recursive "branch" so we have to create a sort of recursive branch ourselves. In this Case, HI and LO would represent Boolean True and false. It doesn't really matter which represents which. Note equation would be count(C[i..n],Bool,Count1,Count2,Score)