

Analysis: Spiraling Into Control

Test Sets 1 and 2

You may or may not have run across the old chestnut of a technical interview question that asks you to number all of the cells of a grid in a spiral pattern. In this problem, doing that is potentially helpful for the first two test sets. However, the last test set can be (and must be) solved without explicitly creating and numbering a spiral. (For a case with $N = 9999$, we would need to store and number almost 10^8 cells!) We'll worry about that later in the analysis.

To make a numbered spiral, we can create a grid and loop through it, starting from the upper left cell, and making a 90 degree turn to the right each time we encounter a grid boundary or an already-numbered cell. One clean way to do this is to use an array of "directions": $((0, 1), (1, 0), (0, -1), (-1, 0))$, where the starting direction $(0, 1)$ means stay in the same row and move one column right, and so on, with the other three referring to moves downward, leftward, and upward, respectively, in the grid. When our current direction is $(\Delta r, \Delta c)$ and we are at cell (r, c) in the grid, we check cell $(r + \Delta r, c + \Delta c)$ to see if it is outside the grid or has already been numbered. If so, we choose the next direction in the direction vector, looping back to the start if we go off the end, and then calculate $(r + \Delta r, c + \Delta c)$ using the new $(\Delta r, \Delta c)$. Then, whether or not we changed direction, we label the current cell, increment our label counter, and proceed to the cell $(r + \Delta r, c + \Delta c)$. We stop when we label the N^2 -th cell.

For Test Set 1, we can exhaustively enumerate all legal paths, using our spiral numbering to determine which moves are allowed. We can keep track of the shortcuts taken in each possible path, and see whether any path finishes in exactly K moves. Since there is a shortcut to take (or not take) in almost every cell, this solution is exponential.

For Test Set 2, we can refine this idea to avoid explicitly considering all possible paths. For each cell in the grid, we create an array that can hold up to one path for every possible number of moves "so far". Then we proceed along the spiral in consecutive numerical order. For every cell we visit, for every path in that cell's array, we try to extend it into all legal neighboring cells.

For example, in a 5×5 grid, we start at cell 1 with only a way to get there in 0 moves. We tell each of the legal neighboring cells (2 and 16) that we have a way to get there in 1 move, starting from cell 1. Then, in cell 2, we tell each of cells 3 and 17 that we have a way to get there in 2 moves, with the starting prefix 1, 2, and so on. The key difference from the Test Set 1 solution is that when a cell is offered a path and it already has a path with exactly that number of moves, it does not store the new path. This cuts down on the proliferation of possible paths.

And so we have a [dynamic programming](#) / [memoization](#) solution. Since there is at most one shortcut to take in each cell, we do constant work per cell; since we are populating a table that is $N^2 \times N$, this solution takes $O(N^3)$ time.

Test Set 3

To be able to handle grids with N up to 9999, we need to do three things efficiently:

- Given a value of K , determine whether a path with exactly K moves exists.
- Find such a path.
- Given the coordinates of a grid cell, find its number in the spiral.

We can begin by observing that the following values of K are IMPOSSIBLE:

1. $K < N - 1$. In this case, even if we move as directly as possible toward the center (taking only shortcuts), there are not enough moves to get there.
2. Odd K . We can see this via a "checkerboard argument". Imagine that the grid has a [checkerboard pattern](#), with the upper left cell being black. Then the diagonal running from the upper left cell to the lower right cell is entirely black, and so the central cell is black. Now, each move is in an orthogonal (i.e., "compass") direction, so each move either takes us from a black cell to a red cell or vice versa. Therefore, to end on the central cell, which is black, we must make an even number of moves.

As it turns out, these are the *only* impossible cases! To see why, it helps to think of the spiral as consisting of concentric square rings, with "ring" 0 being just the central cell, ring 1 being the eight cells around that, ring 2 being the sixteen cells around ring 1, and so on. For example, the rings in the $N = 7$ grid look like this:

```

3333333
3222223
3211123
3210123
3211123
3222223
3333333

```

In ring 1, there are three possible shortcuts: we can move into ring 0 from the top, right, or bottom, and this will save us 6, 4, or 2 moves, respectively, compared to just walking through all of ring 1 (and into ring 0) without taking a shortcut. But we can only take one of these shortcuts.

In ring 2 (and beyond), for simplicity, let's only consider taking shortcuts that are in the same row or column as the central cell. There are four such shortcuts, and they will save us 14, 12, 10, or 8 moves, respectively. Notice that if we take the 8-move shortcut from ring 2, for example, we cannot use any of the shortcuts in ring 1. However, if we take the 14-move shortcut from ring 2, we will still have access to all three shortcuts in ring 1.

Similarly, in ring 3, the shortcuts save us 22, 20, 18, or 16 moves, and if we take the first of these, we still have access to all four shortcuts in ring 2.

We can turn these observations into a constructive solution for any even K that has an answer:

- If we want to save between 2 and 22 moves, we take the specific shortcut with that savings.
- If we want to save between 24 and 36 moves, we can take the 22-move shortcut and then take the specific shortcut (from ring 1 or 2) that gets us the remaining savings.
- If we want to save 38, 40, or 42 moves, we can take the 22-move and 14-move shortcuts and then the 2, 4, or 6-move shortcut from ring 1.
- We already established that we cannot save 44 moves or more (since then K would be less than $N - 1$).

Generalizing this strategy: we can find the number of moves we need to save (which is $N^2 - 1 - K$; call it s), and then proceed from the outermost ring to the innermost. At each ring r ,

1. If s is greater than or equal to the number of moves saved by the largest shortcut (i.e., $8r - 2$ moves), we take that shortcut, subtract its savings from s , and proceed to the $r - 1$ -th ring.
2. If s is equal to the number of moves saved by some other shortcut in ring r (i.e., $8r - 4$, $8r - 6$, or $8r - 8$), we take that shortcut and stop. (We must be careful not to do this when

$8r - 8 = 0$, since that move — from cell $\mathbf{N}^2 - 1$ to cell $\mathbf{N}^2$ — saves us nothing and is not a shortcut!

3. Otherwise, we reject all shortcuts in ring r , and proceed to ring $r - 1$.

All that remains is to find the room numbers for the shortcuts that we take. Instead of trying to generate the entire spiral, we can notice that, say, the upper left corners of the rings (starting in the central cell and going outward) have values $\mathbf{N}^2, \mathbf{N}^2 - 8, \mathbf{N}^2 - 8 - 16, \mathbf{N}^2 - 8 - 16 - 24$, etc. We can even turn this into a formula: the upper left cell of ring r has number $\mathbf{N}^2 - 8 \sum_{i=0}^r i = \mathbf{N}^2 - 4(r)(r + 1)$.

Once we have the number of the upper left cell of a ring, it is not too hard to find the values of the cells we are using for shortcuts in that ring (i.e., the ones in the central row or column). If the upper left cell's number is x , the shortcuts have numbers $x + r, x + 3r, x + 5r, x + 7r$.

Because there are $\frac{\mathbf{N}+1}{2}$ rings in total, and we do $O(1)$ work in each of them, this solution is $O(\mathbf{N})$ and easily passes Test Set 3. There are values of $\mathbf{K}$ that require a shortcut to be used in every ring (except the central cell), so we can't do better than this in general.

It could have been worse...

We thought about presenting this problem in a different way, without mentioning spirals: starting from the upper left cell of an $\mathbf{N} \times \mathbf{N}$ grid, give a path that reaches the central cell in exactly $\mathbf{K}$ moves, or say it is impossible. One solution is to come up with the spiral path and strategy from this problem! But there were many other time-wasting roads to go down there, and the problem was already challenging for the first problem of a Round 2.

Analysis: Pixelated Circle

Let C and C_w be the set of pixels colored by `draw_circle_filled(R)` and `draw_circle_filled_wrong(R)`, the number of pixels that have different colors in these pictures would be the cardinality (size) of the symmetric difference of C and C_w , that is: $|C \Delta C_w| = |(C \setminus C_w) \cup (C_w \setminus C)|$.

Test Set 1

For test set 1, **R** is small enough to build the sets of colored pixels from `draw_circle_filled(R)` and `draw_circle_filled_wrong(R)` with hash set of tuples. By implementing the pseudocode in the problem statement, the time complexity would be $O(\mathbf{R}^2)$ to get all colored pixels and $O(\mathbf{R}^2)$ to compute the symmetric difference of the two sets.

Test Set 2

The key observation for optimizing the solution is that for any **R**, every pixel colored by `draw_circle_filled_wrong(R)` is also colored by `draw_circle_filled(R)`, that is $C_w \subseteq C$. Therefore, we can simplify the size of symmetric difference to:

$$\begin{aligned} |C \Delta C_w| &= |(C \setminus C_w) \cup (C_w \setminus C)| \\ &= |(C \setminus C_w) \cup \emptyset| \\ &= |C| - |C \cap C_w| \\ &= |C| - |C_w| \end{aligned}$$

which means we can count the number of pixels colored by `draw_circle_filled(R)` and `draw_circle_filled_wrong(R)` separately, and the answer would be the difference between these two numbers. The proof of this observation is given at the [end of this analysis](#).

Count pixels colored by `draw_circle_filled(R)`

To get the number of pixels colored by `draw_circle_filled(R)`, we need to iterate through all possible values of x and find y_{min} and y_{max} for each x which satisfy $\text{round}(\sqrt{x^2 + y^2}) \leq \mathbf{R}$ for all $y_{min} \leq y \leq y_{max}$. A solution for them is $y_{max} = \text{floor}(\sqrt{\mathbf{R} + 0.5^2 - x^2})$ and $y_{min} = -y_{max}$. Therefore, we can get the number of colored pixels with a for-loop for the following equation:

$$|C| = \sum_{x=-\mathbf{R}}^{\mathbf{R}} \text{floor}(\sqrt{(\mathbf{R} + 0.5)^2 - x^2}) \times 2 + 1$$

Time complexity: $O(\mathbf{R})$

Count pixels colored by `draw_circle_filled_wrong(R)`

`draw_circle_filled_wrong(R)` is composed of `draw_circle_perimeter( $r$ )` calls with r from 0 to **R**. Notice that pixels colored by `draw_circle_perimeter( $r_1$ )` and `draw_circle_perimeter( $r_2$ )` never overlap if $r_1 \neq r_2$. Based on this observation, we can count the number of pixels colored by each `draw_circle_perimeter( $r$ )` separately and

sum them up to get the total number of colored pixels. The proof of this observation is given at the [end of this analysis](#).

Looking into function `draw_circle_perimeter(r)`, we can break the colored pixels into 4 quadrants and count them separately. Since the colored pattern is symmetric to both x-axis and y-axis, we just need to count the pixels in Quadrant 1 (Q1), and the total count excluding the origin pixel would be that number times 4, and plus 1 to include the origin pixel.

For $r \geq 1$, the colored pixels in Q1 are symmetric to the line $x = y$, and there are exactly x_t colored pixels between y-axis and (x_t, y_t) , the closest point to line $x = y$ (above or on the line, $x_t \geq y_t$). Since $x = y$ is at 45° to the x-axis, the integer x_t would be either $\text{ceil}(r/\cos(45^\circ))$ or $\text{floor}(r/\cos(45^\circ))$. We can compute the corresponding $y_t = \text{round}(\sqrt{r^2 - x_t^2})$ and choose the closer one above or on the line $x = y$. Afterwards, the number of colored pixels in Q1 including x-axis would be $2 \times x_t + 1$, and minus 1 if (x_t, y_t) lies on the line $x = y$ since it is not mirrored in this case.

Time complexity: $O(1)$ for counting pixels colored by `draw_circle_perimeter(r)`, and $O(\mathbf{R})$ for all colored pixels.

Proof: $C_w \subseteq C$

For every positive $\mathbf{R}$, r and x such that $0 \leq r \leq \mathbf{R}$ and $-r \leq x \leq r$, we want to prove that the following inequality always satisfies:

$$\begin{aligned} & \text{round}(\sqrt{x^2 + \text{round}(\sqrt{r^2 - x^2})^2}) \leq r \\ \iff & \sqrt{x^2 + \text{round}(\sqrt{r^2 - x^2})^2} - 0.5 \leq r \\ \iff & \sqrt{x^2 + \text{round}(\sqrt{r^2 - x^2})^2} \leq r + 0.5 \\ \iff & x^2 + \text{round}(\sqrt{r^2 - x^2})^2 \leq r^2 + r + 0.25 \\ \iff & x^2 + (\sqrt{r^2 - x^2} + 0.5)^2 \leq r^2 + r + 0.25 \\ \iff & r^2 + \sqrt{r^2 - x^2} + 0.25 \leq r^2 + r + 0.25 \end{aligned}$$

which always holds since $\sqrt{r^2 - x^2} \leq \sqrt{r^2} = r$ when $|x| \leq r$ and $r \geq 0$.

Secondly, $y = \text{round}(\sqrt{r^2 - x^2}) \leq \text{round}(\sqrt{r^2}) \leq \mathbf{R}$. Therefore $-\mathbf{R} \leq y \leq \mathbf{R}$ always holds.

The proof above shows that pixels colored by `draw_circle_filled_wrong(r)` with $0 \leq r \leq \mathbf{R}$ also satisfy the coloring condition in `draw_circle_filled(\mathbf{R})`, which implies $C_w \subseteq C$.

Proof: `draw_circle_perimeter(r1)` and `draw_circle_perimeter(r2)` never overlap

For the first coloring statement in `draw_circle_perimeter(r)`, we want to prove that given a fixed x , the inequality $y_1 = \text{round}(\sqrt{r_1^2 - x^2}) \neq y_2 = \text{round}(\sqrt{r_2^2 - x^2})$ for any pair of integers r_1 and r_2 such that $r_1 > r_2 \geq 0$ and $|x| \leq r_2$ is always true:

$$\begin{aligned}
& |\sqrt{r_1^2 - x^2} - \sqrt{r_2^2 - x^2}| \\
= & \sqrt{r_1^2 - x^2} - \sqrt{r_2^2 - x^2} & r_1 > r_2 \\
\geq & \sqrt{r_1^2 - x^2} - \sqrt{(r_1 - 1)^2 - x^2} & r_1 \text{ and } r_2 \text{ are integers} \\
\geq & (\sqrt{r_1^2} - \sqrt{x^2}) - (\sqrt{(r_1 - 1)^2} - \sqrt{x^2}) \text{ for any } |x| \leq r_1 - 1 \\
= & r_1 - x - (r_1 - 1) + x \\
= & 1
\end{aligned}$$

Based on the proof above, we can further get:

$$\begin{aligned}
& \mathbf{round}(\sqrt{r_1^2 - x^2}) - \mathbf{round}(\sqrt{r_2^2 - x^2}) \\
\geq & \mathbf{round}(\sqrt{r_1^2 - x^2}) - \mathbf{round}(\sqrt{(r_1 - 1)^2 - x^2}) \\
\geq & \mathbf{round}(\sqrt{r_1^2 - x^2}) - \mathbf{round}(\sqrt{r_1^2 - x^2} - 1) \\
\geq & 1
\end{aligned}$$

Therefore $y_1 \neq y_2$ always holds for all $r_1 > r_2 \geq 0$ given a fixed x such that $|x| \leq r_2$. For the cases that $r_2 < |x| \leq r_1$, the pixels will never satisfy the coloring condition in `draw_circle_perimeter( $r_2$ )`.

We can further extend the proof above for the 2nd, 3rd, and 4th coloring statement in `draw_circle_perimeter( $r$ )` and prove that pixels colored by `draw_circle_perimeter( $r_1$ )` and `draw_circle_perimeter( $r_2$ )` never overlap.

Analysis: Saving the Jelly

Test Set 1

With $N \leq 10$, it is possible to use dynamic programming with bitmasking, with one bit for each candy and each student remaining. This effectively lets us try all possible orderings of students as well as all possible ways to break ties. The total complexity is $O(N^2 \times 2^{2 \times N})$. This runs in time because we only consider states where the same number of candies and students have been matched.

Test Set 2

Firstly, let's construct a bipartite graph with the N children and the N candies as vertices (not including Mr Jolly's blueberry jelly). Add an edge from child a to candy b if child a is equally close or closer to b than the blueberry jelly.

It's clear that a necessary (but perhaps not sufficient) condition for Mr Jolly to get his blueberry jelly is that the graph has a [perfect matching](#).

It turns out, this is also a sufficient condition. Let's try to show this by turning a perfect matching into an order that Mr. Jolly can call the children's names.

Firstly, if there is a child who is matched to the candy that is closest to them, then we can call that child's name and remove them and the candy from the graph.

Otherwise, we are in the situation where every child is matched to a (ungrabbed) candy that is *not* the closest one to them. Then, we will find a cycle in the graph using the following procedure. Pick an arbitrary child a to start at.

1. Find the candy b that is closest to child a . Go to b . This edge is guaranteed not to be part of the current matching.
2. Find the child a that is currently matched to candy b . Go to a . This edge is in the current matching.

Eventually, this process will create a cycle of even length (not necessarily including the child we started with). Because we alternated between edges in the matching and edges not in the matching, we can swap the matched edges/unmatched edges to create a new perfect matching. Note that in doing so, all the children in the cycle are now matched to the closest candy to them, so at least one child's name can now be called.

We have shown that a perfect matching is a necessary condition, and that an order can be constructed from a perfect matching that satisfies Mr. Jolly's requirements, thus proving that a perfect matching exists if and only if Mr. Jolly's requirements can be satisfied.

A perfect matching can be found in $O(N^2 \sqrt{N})$ using [Hopcroft-Karp](#), and constructing a solution can be done in $O(N^2)$ if implemented with some care.

Analysis: I, O Bot

There is no value in carrying balls across the origin without depositing them into the warehouse, therefore, collecting the balls with positive coordinates X_i and those with negative coordinates are two similar but independent tasks. Hence, in what follows, we assume that $X_i > 0$ for all i . Moreover, let us assume that the balls are sorted in ascending order by X_i .

A solution to the problem consists of a number of passes or round-trips from the origin and back with one or two balls collected in each pass. The time required to collect a single ball i in a pass is $2X_i$. The time required to collect two balls i and j is $2 \times \max(X_i, X_j)$ if the balls are of different shapes and $2 \times \max(X_i, X_j) + C$ otherwise. We say that two balls i and j are *matched* (and write (i, j)) if they are collected in the same pass. Since the order of passes is not affecting the overall time for collecting all balls, we can equivalently think of the problem as one of finding an optimal matching of balls.

The following observation will be useful throughout the analysis.

Observation 1: Suppose we want to collect the first i balls ($i \geq 2$) and $S_i \neq S_{i-1}$. In an optimal matching, the i -th ball is matched with the $(i-1)$ -th ball.

Proof: Consider any matching of balls, where i -th ball is *not* matched with $(i-1)$ -th ball, and assume that i -th ball is a 0.

1. If none of the two balls is matched, we can match the balls and save $2X_{i-1}$ seconds.
2. If there is a matching $(i-1, j)$, $j < i-1$, and i -th ball is not matched, then we can match $(i-1)$ -th ball with i -th ball instead and save at least $2 \times (X_{i-1} - X_j)$ seconds ($2 \times (X_{i-1} - X_j) + C$, if j -th ball is 1-shaped).
3. Similarly, if there is a matching (i, j) , $j < i-1$, and $(i-1)$ -th ball is not matched, we can match i -th ball with $(i-1)$ -th ball instead and, again, save at least $2 \times (X_{i-1} - X_j)$ seconds.
4. Lastly, if there are matchings (i, j) and $(i-1, k)$, $j < i-1$ and $k < i-1$, then we can rearrange the matchings as $(i, i-1)$ and (j, k) saving at least $2 \times (X_{i-1} - \max(X_j, X_k))$ seconds.

Test Set 1

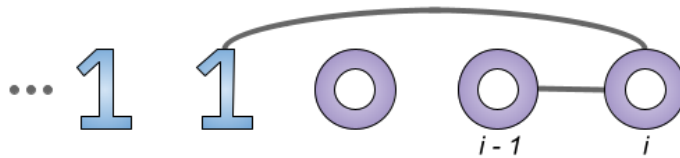
Observation 1 helps us match the balls if the last two balls have different shapes. But what if they have the same shape, say a 0?

Observation 2: Suppose we want to collect the first i balls ($i \geq 2$) and $S_i = S_{i-1} = 0$. There is an optimal matching of balls such that one of the following conditions holds:

1. The last two 0-shaped balls i and $i-1$ are matched.
2. There is a matching (i, j) with $S_j = 1$ and, for all $k \in [j+1, i]$, $S_k = 0$. In other words, i -th ball is matched with the nearest 1-shaped ball on its left.
3. There are no 1-shaped balls and i -th ball remains unmatched.

Proof: The full proof is a lengthy case analysis, which we omit here. The idea is that matching i -th ball with the rightmost ball of a particular shape is generally at least as good as matching with another ball of that shape. For example, suppose that i -th ball is matched with a 1-shaped ball l such that there is another 1-shaped ball j with $l < j < i$. If the ball j is unmatched, we can match the ball i with j instead and save $2 \times (X_j - X_l)$ seconds. Otherwise, if the ball j is matched with some other ball k , we can swap the roles of balls l and j and create the matchings (i, j) and (k, l) obtaining the same overall time (if $k > j$) or better.

This means that we can try matching the last 0-shaped ball with the 0-shaped ball before or the rightmost 1-shaped ball (if any), and at least one of these moves will be optimal.



The two observations lead to a dynamic programming solution. Let $dp[i][j]$ be the optimum time to collect the first i 0-shaped balls and the first j 1-shaped balls. The base case is $dp[0][0] = 0$. For $i + j > 0$, suppose again that the rightmost of these $i + j$ balls is 0-shaped and it has the coordinate x . The case when the rightmost ball is 1-shaped is symmetric. To eliminate some other corner cases, $dp[1][0] = 2x$, $dp[i][0] = \min(dp[i-1][0], dp[i-2][0] + C) + 2x$ for $i \geq 2$, and $dp[1][j] = dp[0][j-1] + 2x$ for $j \geq 1$. For the general case with $i \geq 2$ and $j \geq 1$, if the penultimate ball is 1-shaped, then $dp[i][j] = dp[i-1][j-1] + 2x$ (Observation 1). Otherwise, we can choose to match the last 0-shaped ball with the previous 0-shaped ball or the rightmost 1-shaped ball (Observation 2), namely, $dp[i][j] = \min(dp[i-2][j] + C, dp[i-1][j-1]) + 2x$.

The final answer is $dp[N_0][N_1]$, where N_0 and N_1 denote the total number of 0-shaped and 1-shaped balls, respectively. The time complexity of this algorithm is $O(N^2)$.

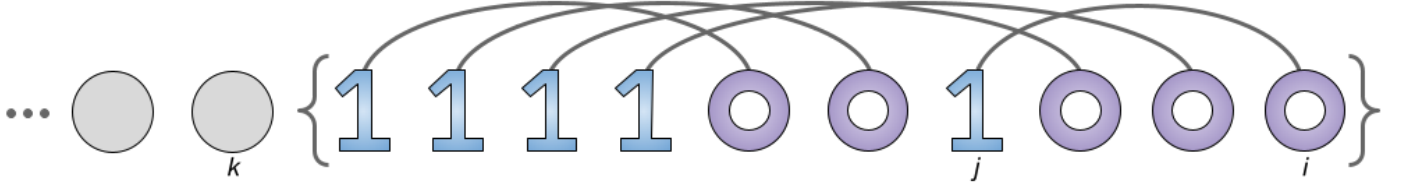
Test Set 2

Using dynamic programming from a different angle, we can solve the problem in linear time, apart from the initial sorting. Let $dp[i]$ be the optimum time to collect the first i balls. As the base cases, $dp[0] = 0$ and $dp[1] = 2X_1$. To calculate $dp[i]$ for $i \geq 2$, suppose once more that the i -th ball is 0-shaped. If the $(i-1)$ -th ball is 1-shaped, we can match the last two balls and $dp[i] = dp[i-2] + 2X_i$ (Observation 1). Otherwise, using Observation 2, we have the options to match the last two 0-shaped balls and collect all balls in $dp[i-2] + C + 2X_i$ seconds, or to match i -th ball with the rightmost 1-shaped ball j . The dynamic programming recurrence is not obvious in the latter case, though, as we do not know the optimum matching for the first $i-1$ balls except for ball j . What happens to the 0-shaped balls in-between j and i ? We are missing another key observation here.

Observation 3: If there is an optimal matching of the first i balls such that the 0-shaped ball i is matched with the rightmost 1-shaped ball j and $i - 1 \neq j$, then the 0-shaped ball $i - 1$ is *not* matched with another 0-shaped ball.

Proof: Assume on the contrary that we have two pairs of matched balls (i, j) and $(i - 1, k)$, $k < i - 1$, such that ball k is 0-shaped. These two matched pairs contribute $2\mathbf{X}_i + 2\mathbf{X}_{i-1} + \mathbf{C}$ seconds to the overall matching cost. But then we can rearrange the matchings as $(i, i - 1)$ and (j, k) costing us only $2\mathbf{X}_i + \mathbf{C} + 2 \times \max(\mathbf{X}_j, \mathbf{X}_k)$ seconds, which is $2(\mathbf{X}_{i-1} - \max(\mathbf{X}_j, \mathbf{X}_k))$ seconds less. This contradicts the optimality assumption of the given matching.

It follows from Observation 3 that the 0-shaped ball $i - 1$ must be matched with another 1-shaped ball, specifically the rightmost unmatched 1-shaped ball. And we can extend this argument and repeatedly match 0-shaped balls with 1-shaped balls sweeping leftward for as long as there is another 0-shaped ball to the right of a matched 1-shaped ball. This process is illustrated in the drawing below.



Let k be the rightmost unmatched ball after the above 0-1 matching process. There are no shape changes in the set of balls $\{k + 1, k + 2, \dots, i\}$ and the cost of collecting those balls is twice the sum $X_{0\text{-shaped}}(k + 1, i)$ of x -coordinates of 0-shaped balls in $\{k + 1, k + 2, \dots, i\}$. Therefore, the cost of collecting all i balls in this way is $dp[k] + 2 \times X_{0\text{-shaped}}(k + 1, i)$.

$X_{0\text{-shaped}}(k + 1, i)$ can be calculated in $O(1)$ time using prefix sums. But how do we get the index k efficiently without actually carrying out the matching process? Note that k is the largest index such that $k < i$ and the set $\{k + 1, k + 2, \dots, i\}$ contains equal number of 0-shaped and 1-shaped balls. Consider the balance b_i of 0/1 balls at each index i , namely, $b_i = z_i - o_i$, where z_i and o_i is the number of 0-shaped and 1-shaped balls in the set $\{1, 2, \dots, i\}$. The set $\{k + 1, k + 2, \dots, i\}$ has equal number of 0-shaped and 1-shaped balls if and only if $b_k = b_i$. The index k can be looked up in $O(1)$ time if we maintain a hash-table of indices, when each balance was last registered. If the current balance b_i is seen for the first time, it means that there are not enough 1-shaped balls to match all 0-shaped balls with, and we can choose $k = 0$.

We are performing a constant number of operations at each index in this approach, so the overall time complexity is dominated by the sorting, thus $O(N \log N)$.