Singapore Informatics League 2026

Task Editorial (Online Contest)

July 2026

Contents

Level 1

Task 1: `Enclose Duck`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Seah E-Ket (JustKitkat)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtasks 1, 2 and 3

Additional constraints for Subtask 1: $n = 7$

Additional constraints for Subtask 2: $n = 10$

Additional constraints for Subtask 3: $n = 10$

You could determine the solution by just using some brainpower, considering which choke-points are the tightest, to block off the maximum number of potential moves.

The Drawn solutions below use # to denote new walls added.

Drawn solution for Subtask 1:

```
7
X.X.X.X
...#D#.
X...#.X
.......
X.....X
.......
X.X.X.X
```

Drawn solution for Subtask 2:

```
10
...XX.....
.....XXX..
..X.....X.
..X.....X.
.X.......X
X....X#..X
.....#DX..
.X....X..X
.X..XX....
..X.......
```

Drawn solution for Subtask 3:

```
10
.X..X.....
.X..XXX.X.
.XX#.....X
X..D#.....
.X...XXXXX
.X.......X
X......X#.
X.....X...
.X.X.X....
..X.X.....
```

Fun fact

Since you can always block off all of the first four moves of the duck, this gives you an upper bound (highest possible value) on the answer of 4, meaning for any testcase, you need at most 4 walls, making it easier to check.

Subtask	Answer
1	3
2	2
3	3

Task 2: `Tag`

Authored by: Brian Lee (penguin133)

Prepared by: Jayden Tiew (Shadow1)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $n = 2$

Consider what happens when two runners are positioned on the circle.

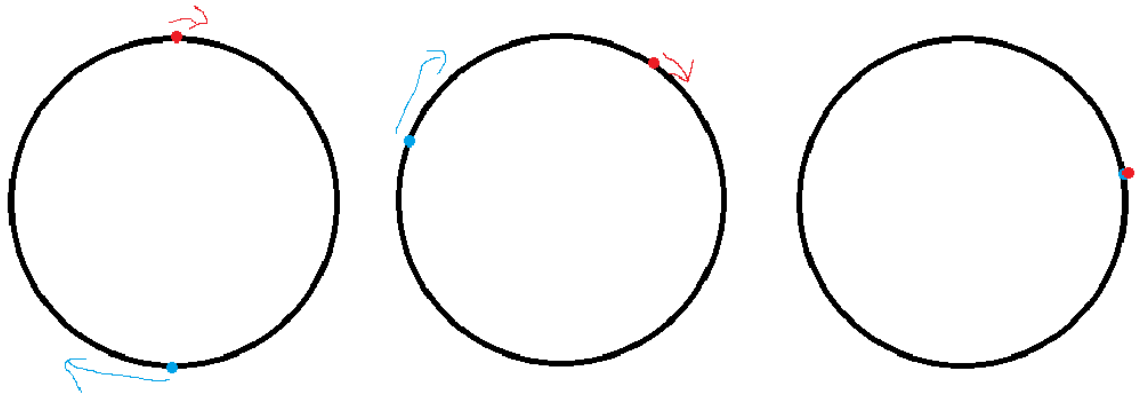


Figure 1: Blue is faster than red

In this case, you can see that the Blue runner's clockwise distance to the Red runner decreases over time, allowing Blue to catch Red. This leads to only one runner remaining.

Now, consider the case that Red is faster than Blue (using the same starting positions as the figure above). Now, the clockwise distance from Red to Blue decreases over time, allowing Red to catch Blue. Thus, this also leads to only one runner remaining.

Finally, the last unsettled case is when the runners have the same speed. You will notice that the distance from one runner to the next stays constant no matter how much time passes, resulting in both runners remaining forever.

In the input for subtask 1, the runners have different speeds, and hence the answer is 1. We will reuse some of the logic to solve the further subtasks.

Subtask 2

Additional constraints: All runners are running at the same speed. $v[i] = v[j]$ for all $i \neq j$

Using the same logic as Subtask 1, if all runners run at the same speed, then considering any pair of runners a and b, the circular distance between a and b remains constant and non-zero, thus making it impossible for a to catch b or the other way around.

Since we can apply this logic to all pairs of runners, it means no runner ever catches another runner, and hence the answer would be $n = 10$, as all n runners remain.

Subtask 3

We can reuse the idea of examining the circular distance between each pair of runners over time to make some critical observations:

Observation 1: The fastest runner(s) never get caught

For a runner to get caught, the clockwise distance from the would-be catcher to this runner must decrease to 0. However, if the runner is the fastest runner, then this value can never decrease to 0, as it would only keep increasing unless the runner overtakes the would-be catcher, which would tag out the would-be catcher, not the fastest runner. Hence, it's not possible for the fastest runner to get caught.

Observation 2: Any runner that is slower than any other runner will always get caught eventually

Remember what happened in subtask 1? We can conclude that if two runners remain, the slower one is always tagged out.

In fact, we can say that slower runners are always tagged out, because the fastest runner(s) always survive, the fastest runner(s) can always catch all slower runners.

Following this, we reach a state where only the fastest runner(s) remain, all with an equal speed.

Observation 3: If all runners have the same speed, none are caught We can reuse the logic from Subtask 2.

Hence, the only remaining runners are the ones tied for the fastest speed.

In the input:

```
10 20
7 15 3 19 11 1 17 9 13 5
20 5 20 1 15 20 8 12 5 20
```

The fastest speed is 20, and there are 4 runners sharing that speed, so the answer is 4.

Subtask	Answer
1	1
2	10
3	4

Task 3: `Evil`

Authored by: Brian Lee (penguin133)

Prepared by: Jayden Tiew (Shadow1) and Ernest Kiew (Shor the Duck)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $n = 5$

Since all plates are ice cream, you can only eat 1 plate of ice cream. To minimise the longest row of happy children, you should eat the middle plate, so that the 2 rows of happy children are as even as possible. Hence, he will eat the plate of ice cream in the middle, forming 2 rows of 2 happy children.

Hence, the answer is 2.

Subtask 2 and 3

Additional constraints for Subtask 2: $n = 12$

Additional constraints for Subtask 3: $n = 40$

Firstly, observe that you should eat all the pizza slices, as decreasing the number of happy children will always be good when trying to minimise the largest row of happy children.

After eating all the pizza slices, we are left with some consecutive plates of ice cream with arbitrary lengths, and we can only eat at most one plate of ice cream in each row. Utilising the idea from subtask 1, we know that we should eat the ice cream plate in the middle to split the happy children evenly. Hence, a consecutive row of length x will be split into 2 slices with the bigger length being $\lfloor \frac{x}{2} \rfloor$. After which, we take the maximal of such lengths over all rows and that will be our final answer.

Subtask	Answer
1	2
2	3
3	5

Task 4: `Six Seven`

Authored by: Brian Lee (penguin133)

Prepared by: Quinn Chua (sheep) and Brian Lee (penguin133)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $n = 6$

There are many ways to solve this, including through trial and error. One possible solution is to observe the explanation of the sample testcase, and notice that if we made one swap to form the string `66677`, there would be exactly 6 ways.

Hence, the answer is 5.

Subtask 2 and 3

Additional constraints for Subtask 2: $n = 36$

Additional constraints for Subtask 3: $n = 67$

We will make a few observations.

Observation 1: In any string of a fixed number of `6` and `7`, to maximise the number of ways, all `6`s should be placed in front of all `7`s.

Since we are counting the number of ways to pick a `6` and then pick a `7` to the right of it, to maximise the number of such ways, there should be as many `7`s to the right of each `6`. Hence we would place all the `6`s first and follow that with all the `7`s.

Observation 2: In a string of fixed length x, to maximise the number of ways, half should be `6`s and the other half should be `7`s.

Following our previous observation, suppose we want to place a `6`s followed by b `7`s, where $a + b = x$. Then, the number of ways would be $a \cdot b$, and we aim to maximise this value by selecting values of a and b.

Using some algebraic manipulation, $a \cdot b = a \cdot (x - a) = ax - a^2$, which is a quadratic expression in a and holds a maximal value of $\frac{x^2}{4}$ at the turning point $a = \frac{x}{2}$. In the event that x is odd, we can set $a = \frac{x-1}{2}$ since a must be an integer. In that case, $a \cdot b = \frac{x-1}{2} \cdot \frac{x+1}{2} = \frac{x^2-1}{4} = \lfloor \frac{x^2}{4} \rfloor$.

Given this, we are able to find the maximal number of ways for a string of length x, and it suffices to find the smallest x where the maximal number of ways is greater or equal to n. Hence, we need to find the smallest x such that $\lfloor \frac{x^2}{4} \rfloor \geq n$, and this reduces to $x = \lceil 2\sqrt{n} \rceil$.

Alternative solutions include binary search the answer (BSTA) or trial and error which do not require as much rigorous math.

Subtask	Answer
1	5
2	12
3	17

Task 5: `Circles`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Sun Beichen (TheRaptor)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtasks 1, 2 and 3

Additional constraints for Subtask 1: $n = 1$

Additional constraints for Subtask 2: $n = 3$

Additional constraints for Subtask 3: $n = 5$

Note that the following is rigorous proof: during the contest, you are not expected to rigorously prove it, and for easier problems like this, you are expected to intuit the correct solution and submit it instead of spending time rigorously proving it.

We can use a standard technique present in both Informatics Olympiad and Maths Olympiad to prove our solution here, generating a bound for the answer.

To contain a circle, it is obvious that you must contain the top-most, left-most, bottom-most and right-most points of that circle.

What does this mean for our answer rectangle? The rectangle's top border must be at least at $y_i + r_i$ to cover the top-most point.

The rectangle's left border must be no more than $x_i - r_i$ to cover the left-most point.

The rectangle's bottom border must be no more than $y_i - r_i$ to cover the bottom-most point.

The rectangle's right border must be at least $x_i + r_i$ to cover the right-most point.

This gives us four conditions for the rectangle's borders, one for each side of the rectangle.

In fact, we can then combine the conditions for each circle, to form one singular condition per border, equal to the strictest condition for that border.

For example, if the top border is at least 2, and also at least 5, we only need to care about the

bigger condition, as it's the stricter condition, so the only useful condition is the top border is at least 5.

For example, if the left border is no more than 6, and also no more than 7, we only need to care about the smaller condition, as it's the stricter condition, so the only useful condition is the left border is no more than 6.

If we combine conditions like this, we can choose the number specified by the condition (so if $top \geq 6$, we can choose the top border to be 6), we have constructed the smallest rectangle that could fulfil the necessary condition of containing the outermost points of the circle. We know the answer has to be at least this large, so this is our lower bound (lowest possible answer). Hence, the area of this rectangle or larger is proven to be necessary to solve the problem.

Now, we notice that this rectangle is always sufficient. That is, this rectangle always gives us a valid answer.

Since this area of rectangle is both necessary to solve the problem, and also sufficient to solve the problem, we have proven that this is our final answer.

Further reading on Sufficient and Necessary conditions

Computing the following restrictions, we get the following:

Input 1

	Top	Left	Bottom	Right
Circle 1	2	-2	-2	2
Combined	2	-2	-2	2

$Width = 2 - (-2) = 4$
$Height = 2 - (-2) = 4$
$Area = 16$

Input 2

	Top	Left	Bottom	Right
Circle 1	3	1	1	3
Circle 2	0	-2	-4	2
Circle 3	6	-6	0	0
Combined	6	-6	-4	3

$Width = 3 - (-6) = 9$
$Height = 6 - (-4) = 10$
$Area = 90$

Input 3

	Top	Left	Bottom	Right
Circle 1	1	4	-1	6
Circle 2	5	-4	-1	2
Circle 3	3	-4	1	-2
Circle 4	1	-3	-5	3
Circle 5	-1	-3	-5	1
Combined	5	-4	-5	6

$Width = 6 - (-4) = 10$
$Height = 5 - (-5) = 10$
$Area = 100$

Subtask	**Answer**
1	16
2	90
3	100

Task 6: `Virus`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Ernest Kiew (Shor the Duck)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: The number of sneezes is at most 2**.** Firstly, if a person with 0 viruses sneezes, nothing changes.

Hence, the only sneeze that changes the virus counts initially is if person 1 sneezes, creating a situation where everyone has the same virus count of x.

The next sneeze will then create a situation where everyone except the sneezer has a virus count of $2x$, while the sneezer has a virus count of x.

Thus, you can find x by finding the minimum virus count of all people.

The values of x are 4, 7, and 67, which sum to 78.

Subtask 2

Additional constraints: x **is either** 2 **or** 3 for each testcase.

You should consider what setups are possible with each value of x.

When you start with a value of $x = 2$, everyone's virus counts are always even, regardless of who sneezes. (All sneezes can only add even numbers to the virus counts, which were initially even, resulting in only even counts)

When you start with a value of $x = 3$, everyone's virus counts are always divisible by 3, for the same reason as $x = 2$.

You can then inspect each testcase to determine which one it is (it can't be both, given by the constraint in the testcase, but also the proof is left to the reader :D)

The values of x are 2, 3, and 3, which sum to 8.

Subtask 3

It is given that at most 6 sneezes have occurred.

Let's define a non-trivial sneeze as a sneeze by a person with at least 1 virus count.

We can determine who had the last non-trivial sneeze, because that person that had a virus count added a to everyone else's virus count, so everyone else also must now have at least a virus count.

Therefore, only people with the current lowest virus count can be the last non-trivial sneezer. Since the ordering of people doesn't matter, if two people have the same virus count, it doesn't matter which person sneezed, as the sorted virus counts would be equal. It is then okay to pick one arbitrarily, then work backwards and reduce everyone else's virus count by the chosen person's virus count.

If we repeat this, we will eventually hit the case where everyone except one person has 0 virus count. The answer is then the last remaining person's virus count.

The values of x are 101 and 4, which sum to 105.

Subtask	Answer
1	78
2	8
3	105

Task 7: ABBABBA

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Siew Jing Hong (siewjh)

Editorial written by: Siew Jing Hong (siewjh)

Subtask 1

Additional constraints: s and t consist of only the character `'A'`

At each step, the optimal move is to extend s as much as possible. If doubling the length of s (by copying the entirety of s and pasting at the end) does not cause us to exceed the length of t, we will do that. Otherwise, we will copy and paste a substring of $tlen - slen$ `'A'` to turn s into t.

Since the initial value of $slen = 2$ and $tlen = 17$, we can turn s into t in 4 moves by having $slen$ go $2 \to 4 \to 8 \to 16 \to 17$.

Subtasks 2 and 3

Below is a rigorous explanation and proof of the solution. Do note that you are not expected to rigorously prove it in the contest. Skip to the TLDR if you do not want to read the whole explanation.

Notice that at every point s is a prefix of t. This is due to the fact that we can only add characters at the end of s. Hence if s is not a prefix of t there is no way for us to turn s into t.

This means that the only thing that matters about current string is its length as the individual letters are forced to be the prefix of t.

We define "reach" as the possible lengths we can obtain in 1 move. We can see that "reach" is a continuous range, i.e. if we can reach length l in 1 move, there does not exist a shorter length we cannot reach. Suppose we pasted the substring `'ABA'` to obtain length l. We can obtain $l - 1$ and $l - 2$ with the substrings `'AB'` and `'A'`.

The key idea of the solution is that a longer string can always reach at least as far as a shorter string. Suppose we are at length l_1 and one move can bring us to length l_3. If we started at

a longer length l_2 (but still short of l_3), since whatever string used to extend l_1 to l_3 is also a substring of l_2, we are guaranteed to be able to reach l_3 from l_2.

We formulate a greedy solution to take the longest possible extension at each move. We can prove it optimality with the stay ahead argument.

Consider any solution. We will show that our greedy solution is always able to obtain a length at least as good as the other solution after any number of moves.

This is trivially true for 0 moves as they start with the same string s.

Suppose this is true for k moves. The greedy string is at least as long as the other string. As a longer string can always reach at least as far as a shorter string, the greedy string will be at least as long as the other string after $k + 1$ moves. Hence we have shown by induction that the greedy solution is always at least as good as any solution and is hence optimal.

TLDR: The optimal solution is to take the longest possible extension at each step.

Applying our solution to the 2 tests.

Subtask 2: `'BAB'` $\rightarrow$ `'BABAB'` $\rightarrow$ `'BABABB'` $\rightarrow$ `'BABABBBA'` $\rightarrow$ `'BABABBBAABBB'` in 4 moves

Subtask 3: `'AB'` $\rightarrow$ `'ABB'` $\rightarrow$ `'ABBA'` $\rightarrow$ `'ABBAABBA'` $\rightarrow$ `'ABBAABBAAABB'` $\rightarrow$ `'ABBAABBAAABBBBAAABB'` in 5 moves

Subtask	Answer
1	4
2	4
3	5

Task 8: `Block Game`

Authored by: Siew Jing Hong (siewjh)

Prepared by: Sun Beichen (TheRaptor)

Editorial written by: Ziv Lim (misalignedDiv)

Since every block covers exactly k squares, if the game ends after m blocks then exactly $n-mk$ squares are empty. Alice therefore tries to minimise the final number of blocks, while Bob tries to maximise it.

For a compact manual minimax calculation, we represent a position by the lengths of its empty intervals. Placing a block in an interval of length L replaces it by intervals of lengths x and $L-k-x$, for some offset $0 \le x \le L-k$ chosen by the player placing the block, omitting intervals of length 0. A position is terminal when every interval has length less than k. Working backwards from terminal positions, Alice takes the minimum number of eventual blocks over her moves and Bob takes the maximum.

Subtask 1

Additional constraints: $n = 5$ and $k = 2$

After Alice places the first block, at least one of the remaining empty intervals has length at least 2, so Bob can place a second block. After two size-2 blocks have been placed, only one square remains and no further move is possible. Thus the game always ends with $5 - 2(2) = 1$ empty square.

Subtask 2

Additional constraints: $n = 6$ and $k = 2$

Alice places her block on squares 2 and 3, leaving empty intervals of lengths 1 and 3. Bob can place one block in the interval of length 3, after which all remaining intervals have length at most 1. This leaves $6 - 2(2) = 2$ empty squares.

Bob can always place at least one block after Alice's first move, so Alice cannot leave more than 2 squares empty. Hence the answer is 2.

Subtask 3

Additional constraints: $n = 20$ and $k = 4$

We apply the minimax calculation described above. By reflection, it is sufficient to consider Alice's first block starting at squares 1 through 9, since mirroring the board about its center maps a first block starting at square s to one starting at square $18 - s$, so every start beyond square 9 duplicates an earlier one. The table below records the number of blocks on the board when both players continue optimally.

Alice's first block	Empty interval lengths	Final number of blocks
1-4	0, 16	4
2-5	1, 15	4
3-6	2, 14	4
4-7	3, 13	4
5-8	4, 12	5
6-9	5, 11	4
7-10	6, 10	4
8-11	7, 9	4
9-12	8, 8	4

Alice chooses any row giving 4 final blocks, for example squares 1 to 4. The minimax table also shows that Bob can force at least 4 blocks for every possible first move. Therefore exactly 4 blocks are placed under optimal play, leaving $20 - 4(4) = 4$ empty squares.

Subtask	Answer
1	1
2	2
3	4

Level 2

Task 9: `Pringle Sort 2`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Lim Chang Jun (lcjly)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 2

Additional constraints: $a[1] = n$

The first chip you see is the last chip you can eat.

Therefore, one hand must be holding this chip until the very end.

You only have one free hand, and must have a chip in your hand to eat it.

Therefore, you can only eat all chips if the chip in your other hand is exactly the number you need.

The answer is therefore 1 if and only if the chip numbers are:

```
n 1 2 3 4 5 ...  n-1
```

Subtask 1 and 3

Additional constraints for Subtask 1: $n = 5$ Note that your left hand and right hand are indistinguishable.

That is, which hand you hold a chip in doesn't matter. Therefore, when encountering a new chip, you either eat it, or keep it.

You can only keep two chips at most in your hands.

As you must eat chips in order, you must only eat a chip if it's the next lowest number you

haven't eaten yet.

As there is no reason to keep a chip that can be eaten, we will eat it if it can be eaten.

This process actually solves the full problem.

1. If any of the chips in our hand is eatable, we eat them.
2. Otherwise, we pick up the next chip. If there are no free hands to do this, we fail.
3. Repeat from step 1 until all chips are eaten, or we fail.

Subtask	Answer
1	10101
2	0001110001
3	0100101000

Task 10: `Duck Race`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 2

Additional constraints: $1 \leq n \leq 15\,000$ and $l[i] = r[i] = 1$ for each test case

If the section lengths are fixed at 1, then the time taken for both ducks to complete the race is fixed.

We simply output 0 or 1 depending on which duck wins the race.

Subtask 1 and 3

Additional constraints for Subtask 1: $1 \leq n \leq 2$ for each test case

Additional constraints for Subtask 3: $1 \leq n \leq 15\,000$ for each test case

We now consider the benefit to a duck when we change a race section's length.

When we increase the race section length by 1, duck A's time advantage over duck B increases by $b[i] - a[i]$. (If negative, then duck A loses time advantage)

As a result, if we want to force duck A to win, we must maximise the section lengths of sections where $b[i] - a[i]$ is positive, and minimise the section lengths of sections where $b[i] - a[i]$ is negative.

We can then simulate the race to determine if it is possible for duck A to win. We can do a similar process for duck B, using $a[i] - b[i]$ instead.

This fully solves the problem.

Subtask	Answer
1	10022000
2	11110011
3	2102122220

Task 11: `Heights`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Ziv Lim (misalignedDiv)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $tc = 1$ and $n = 5$

This subtask is meant for manual trial and error to find the best order in which to grow the five people's heights by hand. The experimentation can in fact provide valuable insights to the full solution.

Subtask 2

Additional constraints: $tc = 4$, and $h_i = h_{i-1}$ or $h_i = h_{i-1} + 1$ for all $2 \leq i \leq n$

The initial heights are non-decreasing and contain every integer from $h[1]$ to $h[n]$. Therefore, no valid operation can create a height greater than $h[n]$, so no person can be raised above $h[n]$.

We can raise every person to $h[n]$. Process the heights in ascending order. Before raising the people at height x to $x + 1$, at least one person still has height $x + 1$; use that person as the required anchor. Repeating this for each height up to $h[n]$ raises everyone to $h[n]$.

This leads to the result

$$\sum_{i=1}^{n} (h[n] - h[i])$$

Time Complexity: $\mathcal{O}(n)$

Subtask 3

Additional constraints: No additional constraints

The key observation is that a height that is absent initially can never be created. If no person currently has height x, a person at height $x-1$ cannot be raised to x, because a valid day requires another person at height x to exist before the increase.

A further observation is that for height x to grow into height y, all of $x+1,\ x+2,\ x+3,\ \ldots,\ y$ must exist. This mirrors our result from Subtask 2.

Sort the heights and split them into maximal segments of consecutive values. (See Figure 2 for illustration) For a segment ending at height M, every person in the segment can be raised to M by processing the levels from bottom to top as shown in Subtask 2. This results in the segment contributing:

$$\sum_{\substack{i \\ h[i] \text{ is in the segment}}} (M - h[i])$$

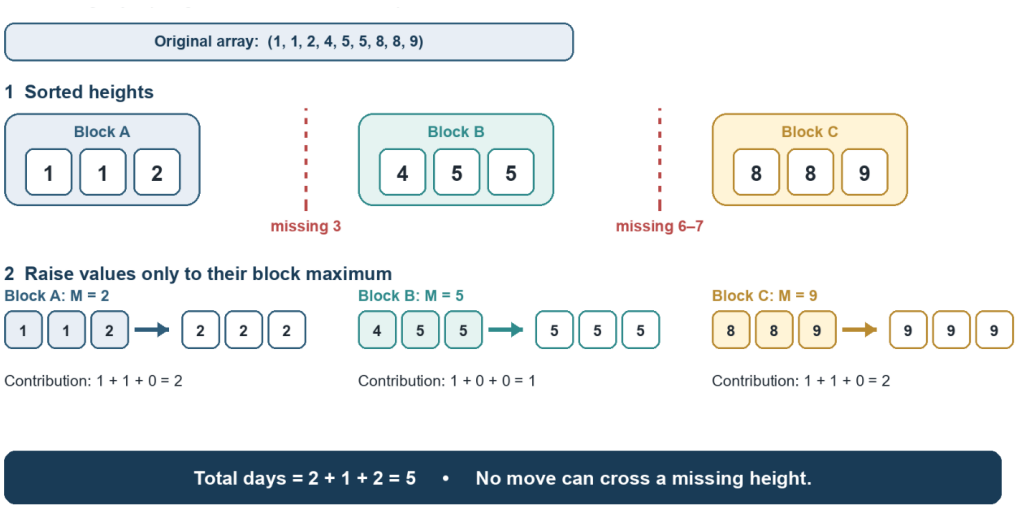


Figure 2: An example of how the array $[1, 1, 2, 4, 5, 5, 8, 8, 9]$ is split

Summing over all segments gives the total number of days.

Time Complexity: $\mathcal{O}(n \log n)$

Subtask	Answer
1	4
2	10400913236
3	10988046285

Task 12: `Mahjong 2`

Authored by: Siew Jing Hong (siewjh)

Prepared by: Siew Jing Hong (siewjh)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $a[i] = 1$ for all $1 \leq i \leq t$

A turn replaces exactly one tile, and the hand always has nm tiles. A winning hand is one in which the count of each tile type is a multiple of m.

Initially, every tile has a different tile type, so $t = nm$. Since each tile type occurs exactly once, a winning hand can retain at most one original tile of each selected type, then draw $m - 1$ more tiles of each selected type. Exactly n original tiles are retained, and the other $nm - n$ tiles are replaced. Hence, the answer for one test case is $n(m - 1)$ turns.

Time complexity: $\mathcal{O}(t)$

Subtask 2

Additional constraints: $m \leq 3$

For each tile type i, $\left\lfloor \frac{a[i]}{m} \right\rfloor$ complete sets are already formed, so they require no turns. We only need to consider the remainders $a[i] \bmod m$.

For $m = 3$, every non-zero remainder is either 1 or 2. Pair a remainder-1 type with a remainder-2 type. Changing the one tile of a remainder-1 type into the type with remainder 2 completes a set in one turn. This is the cheapest way to complete a set.

After making as many pairs as possible, only remainder-1 or remainder-2 types remain. The number of remaining types must be divisible by 3, since the total number of remaining tiles is divisible by 3.

- Three remainder-1 types can be converted into one complete set in two turns.

- Three remainder-2 types can be converted into two complete sets in two turns.

For $m = 2$, all non-zero remainders are 1. Pairing two 1s completes a set in one turn. For $m = 1$, the hand is already winning.

Time complexity: $\mathcal{O}(t)$

Subtask 3

Additional constraints: No additional constraints

For general m, the same observation from Subtask 2 still applies: only the remainders matter. For each tile type, let

$$r[i] = a[i] \bmod m$$

The $\left\lfloor \frac{a[i]}{m} \right\rfloor$ complete sets of type i are already formed and require no turns. The number of additional sets needed is

$$k = n - \sum_{i=1}^{t} \left\lfloor \frac{a[i]}{m} \right\rfloor$$

If we complete a set using remainder $r[i]$, we keep those $r[i]$ tiles and replace $m - r[i]$ other tiles with tiles of the same type. By an exchange argument, it is optimal to choose the largest remainders. Choosing a smaller remainder would require replacing more tiles to complete the same set, so it is strictly worse. Therefore, choose the k largest remainders.

After sorting the remainders in descending order, the answer is

$$\sum_{i=1}^{k} (m - r[i])$$

Time complexity: $\mathcal{O}(t \log t)$

Subtask	Answer
1	1196186
2	393447
3	493385910935471247

Task 13: `Tokens`

Authored by: Seah E-Ket (JustKitkat)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $1 \leq n \leq 10$ and $0 \leq a[i] \leq 100$

This subtask is meant for manual trial and error to find the best way to schedule Ket's token collections and returns over a small number of days. The experimentation can in fact provide valuable insights to the full solution.

Subtask 2

Additional constraints: $1 \leq n \leq 15\,000$ and $0 \leq a[i] \leq 1000$

Let $dp[x]$ be the minimum cost after the days processed so far when Ket holds exactly x tokens. Initially, $dp[0] = 0$ and all other states are unreachable.

Suppose Ket has y tokens before the next day and x tokens afterwards. Because he may collect or return one token for free, this transition costs

$$\max(0,\ |x - y| - 1).$$

To compute all transitions efficiently, first set

$$next[x] = \min\left(dp[x-1],\ dp[x],\ dp[x+1]\right), \quad 0 \leq x \leq 1000$$

We then perform a left-to-right pass: $next[x] = \min(next[x],\ next[x-1] + 1)$.
Similarly, a right-to-left pass: $next[x] = \min(next[x],\ next[x+1] + 1)$.

After the one free collection or return, every additional token collected or returned costs 1. Therefore, the left-to-right and right-to-left passes propagate the minimum cost to each state,

increasing it by 1 for every additional step. After both passes, $next[x]$ equals the minimum over y of $dp[y] + \max(0,\ |x - y| - 1)$.

Finally, set to ∞ every state that violates the current day's condition. If $b[i] = 0$, set the states with $x < a[i]$ to ∞; otherwise, set the states with $x > a[i]$ to ∞.

After processing all days, the answer is the minimum value in dp.

Time complexity: $\mathcal{O}(n \max(a[i]))$

Subtask 3

Additional constraints: $1 \leq n \leq 15\,000$

For the full constraints, we cannot store the minimum cost for every possible number of tokens. Instead, maintain an interval $[L, R]$ containing all token counts that can currently be reached with the minimum total cost. Initially, $L = R = 0$.

At the start of each day, Ket may collect or return one token for free. Therefore, the interval expands to $[\max(0, L - 1), R + 1]$

Suppose $b[i] = 0$, so Ket must have at least $a[i]$ tokens.

- If $a[i] \leq R$, some token counts in the interval already satisfy the requirement, so set $L = \max(L, a[i])$.
- Otherwise, even R tokens are insufficient. Starting from R is cheapest, so Ket pays $a[i] - R$ to reach $a[i]$. Add this cost to the answer and set $L = R = a[i]$.

For $b[i] = 1$, Ket must have at most $a[i]$ tokens:

- If $a[i] \geq L$, set $R = \min(R, a[i])$.
- Otherwise, even L tokens are too many. Ket pays $L - a[i]$ to return the extra tokens, after which $L = R = a[i]$.

This works because when the required range intersects $[L, R]$, Ket can satisfy it without any additional cost. Otherwise, the closest endpoint of $[L, R]$ is always the cheapest starting point. (See Figure 3 for illustration)

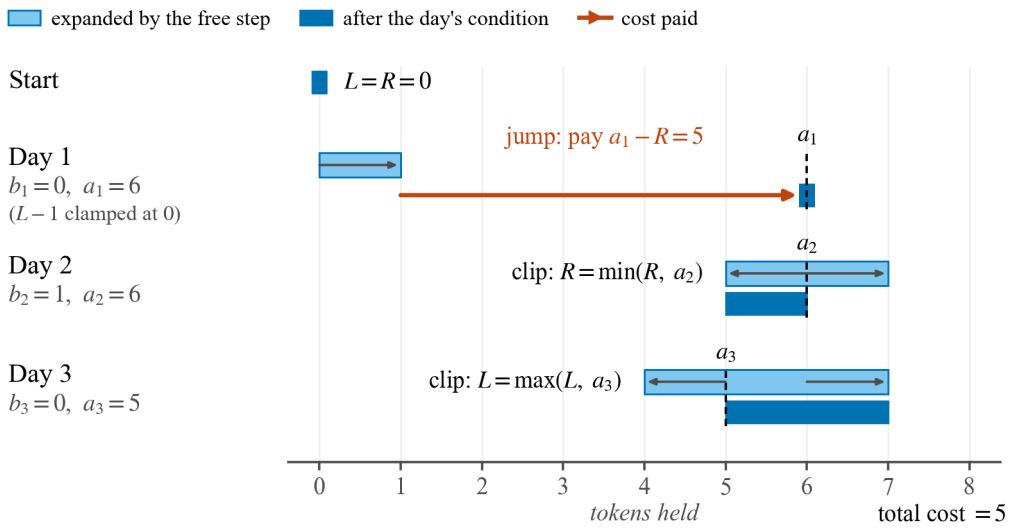


Figure 3: Each day the range first expands to $[\max(0,\ L-1),\ R+1]$ (light), then the condition either clips the range at $a[i]$ for free (dark) or forces a jump to $a[i]$, paying the distance (red).

Time complexity: $\mathcal{O}(n)$.

Subtask	Answer
1	876
2	15864598
3	1598325984

Task 14: `Eat Sleep Repeat`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Xin Anxiaoyang (Xiaoyang)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $a[i] = 1$ for all i

There are n units of food in total, so at most $\left\lfloor \frac{n}{x} \right\rfloor$ digestions are possible. This bound is attained by eating one meal every morning. Whenever the accumulated food reaches x, a subsequent night digests exactly x units. Since $\left\lfloor \frac{n}{x} \right\rfloor \leq n$, the n nights provide enough digestion opportunities.

Therefore, the answer for each test case is $\left\lfloor \frac{n}{x} \right\rfloor$

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $x = 1$

Let e be the number of mornings on which Shor eats. This leaves $2n - e$ opportunities to digest.

Choosing the e largest meals is always optimal. This follows from an exchange argument: replacing a chosen meal with a larger one gives Shor more food to digest, while the cost is unchanged because either choice uses one morning.

It is also optimal to eat the selected meals in descending order, so that larger meals are available earlier and can be digested during more of the remaining sleep opportunities.

After sorting the meals in decreasing order, let P_e be the sum of the first e values. Since $x = 1$, no schedule can gain more than either P_e energy from the food or $2n - e$ energy from the available sleeps. Both bounds can be attained by eating the selected meals first, in decreasing order, and sleeping at every other opportunity. The best value for this e is

$$\max_{0 \le e \le n} (\min(2n - e,\ P_e)).$$

Time complexity: $\mathcal{O}(n \log n)$

Subtask 3

The same argument applies for general x. For a fixed e, the selected meals provide at most $\left\lfloor \frac{P_e}{x} \right\rfloor$ digestions, while eating e meals leaves $2n - e$ sleep opportunities.

Eat the e largest meals first, in descending order. Let S_i be their prefix sum after i meals. If $\left\lfloor \frac{S_i}{x} \right\rfloor < i$, then $S_i < ix$. Since the meals are in descending order, meal i and every later meal have value less than x. Therefore,

$$\left\lfloor \frac{P_e}{x} \right\rfloor \le \left\lfloor \frac{S_i}{x} \right\rfloor + (e - i).$$

Hence, after the first e nights, Shor completes exactly

$$\min\left(e, \left\lfloor \frac{P_e}{x} \right\rfloor\right)$$

digestions.

The remaining $2(n - e)$ sleep opportunities increase this to

$$\min\left(2n - e, \left\lfloor \frac{P_e}{x} \right\rfloor\right).$$

Therefore, the answer is

$$\max_{0 \le e \le n} \left(\min\left(2n - e, \left\lfloor \frac{P_e}{x} \right\rfloor\right)\right).$$

Time complexity: $\mathcal{O}(n \log n)$.

Subtask	Answer
1	301571
2	2694607
3	1929925

Task 15: `Bamboo 3`

Authored by: Brian Lee (penguin133)

Prepared by: Jayden Tiew (Shadow1)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $tc = 2$ and $n = 2$

This subtask is meant for manual trial and error to find the minimum total height achievable when each test case has only two bamboos. The experimentation can in fact provide valuable insights to the full solution.

Subtask 2

Additional constraints: $tc = 5$ and $a[1] = a[2] = \cdots = a[n]$

Let every initial height be h. There are three cases.

If $2h < k$, no pair can be used, so the answer is nh. If $h \geq k$, keep one bamboo at height h and use it to reduce every other bamboo to 0. A zero-height bamboo can then be paired with the remaining bamboo whenever its height is at least k, reducing it to $k - 1$. Thus, the answer is $k - 1$.

It remains to consider $h < k \leq 2h$. Keep one bamboo at height h as a helper. Using this helper, every other bamboo can be cut while its height is at least $k - h$. Hence, each of the other $n - 1$ bamboos can be reduced to $k - h - 1$, giving a total height of

$$h + (n - 1)(k - h - 1).$$

To see why the helper should remain uncut, suppose its height is reduced from h to $h - 1$. Another bamboo must then have height at least

$$k - (h - 1) = k - h + 1$$

before it can be cut. Consequently, it can be reduced only to $k - h$, rather than $k - h - 1$.

Cutting the helper saves one unit, but causes every bamboo cut afterwards to finish one unit taller. At least one such bamboo is needed to cut the helper, so cutting it cannot decrease the final total. Hence, it is optimal to keep one bamboo at height h as the helper.

Time Complexity: $\mathcal{O}(n)$

Subtask 3

Additional constraints: $tc = 10$

Let $M = \max(a[i])$, and keep one bamboo of height M as the helper.

The argument from Subtask 2 generalises directly. If the helper is reduced from M to $M - 1$, every other bamboo reduced afterwards can only be cut to $k - M$ instead of $k - M - 1$. The helper becomes one unit shorter, but at least one other bamboo must finish one unit taller. Thus, cutting the helper early cannot improve the answer, and there is an optimal sequence that keeps it unchanged while reducing all other bamboos.

The reason for choosing a bamboo of height M is that a bamboo can be cut only when the sum of its height and its partner's height is at least k. With a helper of height H, every other bamboo can be reduced to at most $k - H - 1$. This bound decreases as H increases, so a taller helper is never worse. Therefore, choosing a bamboo of maximum height M is optimal.

If $2M < k$, even the two tallest bamboos cannot form a valid pair, since any other pair has a smaller height sum, so if even the two tallest cannot reach k, no pair can. Therefore, no operation is possible, and the answer is the initial sum. Otherwise, if $M \geq k$, the helper can reduce every other bamboo to 0. A zero-height bamboo can then be used to reduce the helper to $k - 1$. Hence, the answer is $k - 1$.

Finally, suppose $M < k \leq 2M$. For every other bamboo with initial height $a[i]$, pairing it with the helper allows it to be reduced to

$$\min(a[i], k - M - 1),$$

The minimum handles a subtlety absent from Subtask 2: if $a[i] \leq k - M - 1$, the bamboo cannot be paired with the helper because their heights would sum to less than k. It is therefore never cut and remains at height $a[i]$.

Therefore, the answer is

$$M + \sum_{i \neq j} \min(a[i], k - M - 1),$$

where j is the index of one occurrence of the maximum.

Time Complexity: $\mathcal{O}(n)$

Subtask	Answer
1	13
2	12000199980027
3	12003499980328

Task 16: Power Sum

Authored by: Quinn Chua (sheep) and Ernest Kiew (Shor the Duck)

Prepared by: Xin Anxiaoyang (Xiaoyang)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $n \leq 300$ and $x \leq 10^6$

We enumerate every ordered triple of pairwise distinct indices (i, j, k) and test whether $a[i] + a[j]^2 + a[k]^3 = x$.

Time Complexity: $\mathcal{O}(n^3)$

Subtask 2

Additional constraints: $n \leq 10\,000$

We fix the value $c = a[k]$ used in the cubic term. The remaining two terms must then satisfy

$$a[i] + a[j]^2 = x - c^3.$$

Since all array values are positive, $a[i] + a[j]^2 > 0$, so $x - c^3$ must also be positive. As $x \leq 10^9$, there are at most 999 possible integer values of c satisfying $c^3 < x$. Thus, the problem reduces to a 2-sum problem with target $x - c^3$.

We use a standard $\mathcal{O}(n)$ two-sum method with a dictionary or unordered set, while ensuring that the three indices are pairwise distinct.

Time Complexity: $\mathcal{O}(n\sqrt[3]{x})$

Subtask 3

Additional constraints: No additional constraints

Let $q = a[j]$ be the value used in the squared term. Then the value used in the linear term must be

$$a[i] = x - c^3 - q^2.$$

We can also discard any squared candidate with $q^2 \geq x$, because the other two terms are positive. Build two lists:

- sq, containing the array values q with $q^2 < x$;
- cu, containing the array values c with $c^3 < x$.

For every pair (c, q) from these lists, compute $z = x - c^3 - q^2$. If z is a positive array value and c, q, and z are pairwise distinct, count the corresponding ordered triplet. Since all array values are distinct, each value identifies a unique index. Check whether z is present using a dictionary.

Let $Q = |sq|$ and $C = |cu|$. Because the array values are distinct and positive, $Q \leq \min(n, 31622)$ and $C \leq \min(n, 999)$. Hence, at most 3.16×10^7 pairs (c, q) need to be checked per test case, which is feasible for the algorithm.

As $Q = x^{\frac{1}{2}}$ and $C = x^{\frac{1}{3}}$, $QC = x^{\frac{5}{6}}$, meaning the algorithm runs in $\mathcal{O}\left(n + x^{\frac{5}{6}}\right)$

Time Complexity: $\mathcal{O}\left(n + x^{\frac{5}{6}}\right)$

Subtask	Answer
1	311
2	13141
3	230593

Level 3

Task 17: `Keys`

Authored by: Seah E-Ket (JustKitkat)

Prepared by: Seah E-Ket (JustKitkat)

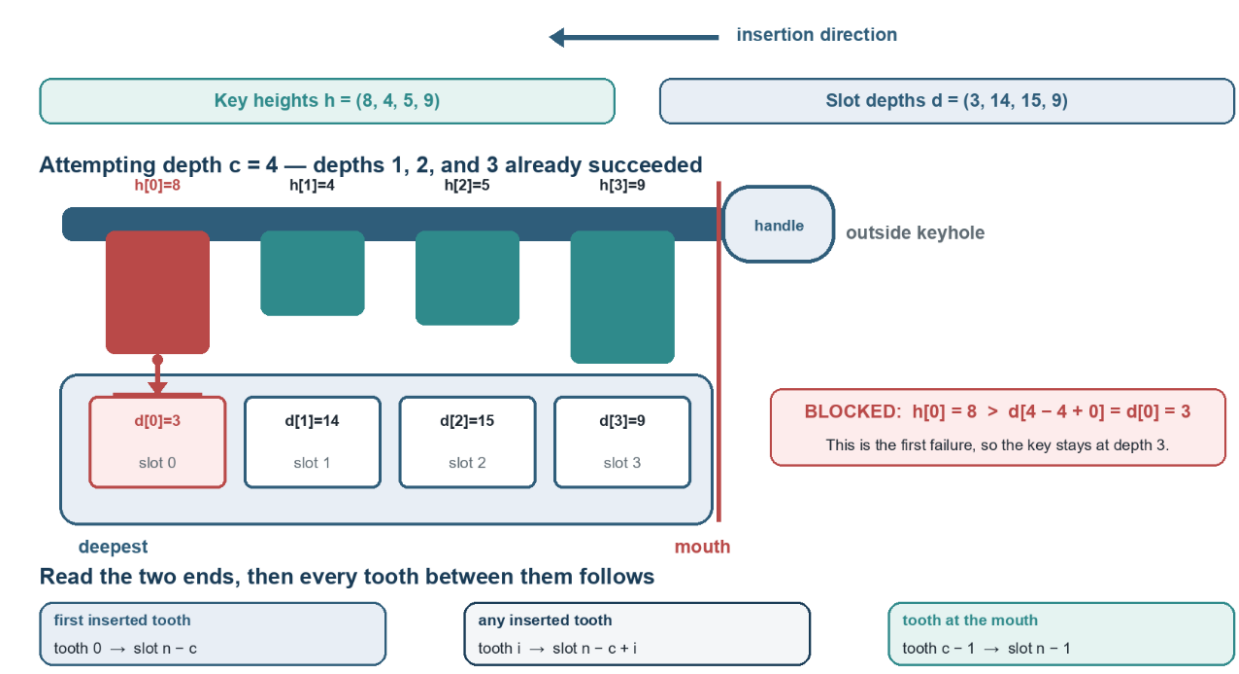
Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $tc = 2$ and $1 \leq n \leq 5$

This subtask is meant for manual trial and error to find the best way to determine how far a small key can be inserted into the keyhole. The experimentation can in fact provide valuable insights to the full solution.

A figure is attached here to understand the mechanics of the key insertion better



Subtask 2

Additional constraints: $tc = 10$ and $n = 100$

Simulate the insertion one unit at a time. To attempt depth c, compare h_i with d_{n-c+i} for every $1 \leq i \leq c$. If every comparison succeeds, continue to depth $c+1$; otherwise, the insertion stops at depth $c-1$.

Time Complexity: $\mathcal{O}(n^2)$

Subtask 3

Additional constraints: $tc = 10$ and $n = 10^5$

Consider a tooth i and a slot j, where $i \leq j$. Tooth i reaches slot j when the key attempts depth $n-(j-i)$. If $h[i] > d[j]$, this pair blocks the insertion, so the maximum completed depth is

$$n-(j-i)-1$$

Therefore, the first obstruction is given by the invalid pair with maximum $j-i$.

To find this pair efficiently, we only need to consider the strict suffix minima of d. Suppose $j < k$ and $d_j \geq d_k$. Any tooth that fits through slot k must also fit through slot j. Equivalently, any tooth blocked by slot j would already have been blocked by slot k. Since slot k is encountered earlier during insertion and yields the larger gap $k-i > j-i$, slot j can be discarded.

Scan d from right to left, retaining a slot j only when $d[j]$ is smaller than every depth retained so far. After reversing the retained slots, both their depths and indices are strictly increasing.

For each tooth i, binary-search the rightmost retained slot j whose depth satisfies $d[j] < h[i]$. If $j \geq i$, update the maximum gap with $j-i$. If no invalid pair exists, the answer is n. (See Figure 4 for an illustration.)

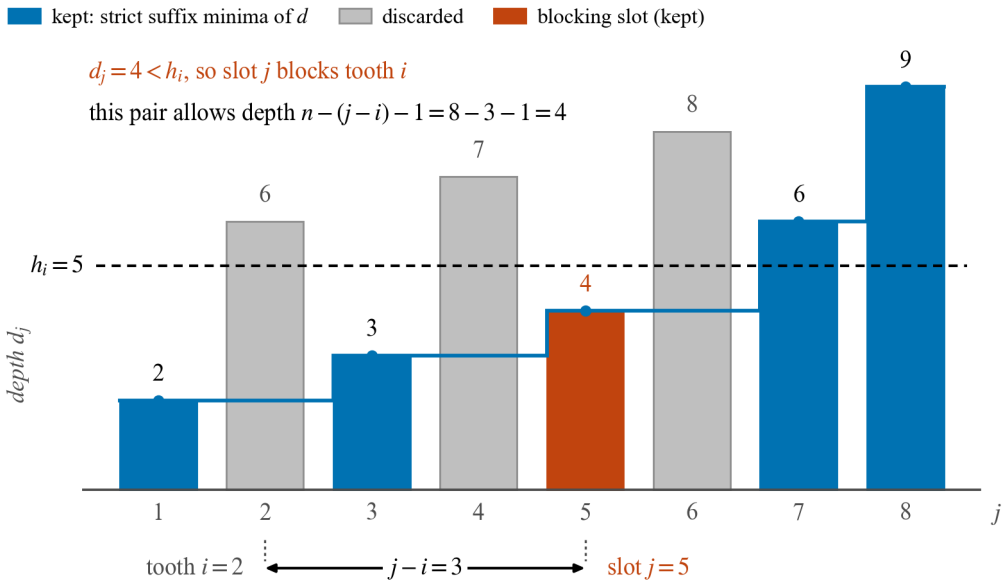


Figure 4: The strict suffix minima of d (blue) form a staircase of strictly increasing depths. A tooth of height $h[i] = 5$ is blocked by the rightmost retained slot with $d[j] < h[i]$ (red), giving the invalid pair with maximum $j - i$.

Time Complexity: $\mathcal{O}(n \log n)$

Subtask	Answer
1	6
2	526
3	527499

Task 18: `Reels`

Authored by: Brian Lee (penguin133)

Prepared by: Sun Beichen (TheRaptor)

Editorial written by: Ziv Lim (misalignedDiv)

Note

Let f_w be the minimum time needed to watch exactly w reels. Since all reel lengths are positive, f_w is non-decreasing in w. Therefore, a query with time x is answered by finding the largest w such that $f_w \leq x$.

To watch w reels, we may stop immediately after the last watched reel. Therefore, it is enough to consider a prefix containing the watched reels and at most k skipped reels.

Subtask 1

Additional constraints: $k = 0$

No reel can be skipped, so watching w reels means watching the first w reels. Hence

$$f_w = t[1] + t[2] + \cdots + t[w].$$

We compute these prefix sums once. For each query, binary search for the largest prefix sum not exceeding x.

Across one test case, the time complexity is $\mathcal{O}(n+q \log n)$ and the memory complexity is $\mathcal{O}(n)$.

Subtask 2

Additional constraints: $1 \leq n, q \leq 5\,000$

We can compute all values of f using dynamic programming. Let $dp[i][j]$ be the minimum time spent after processing the first i reels using j skips ($0 \leq j \leq k$), where the skips may occur

anywhere. The base case is $dp[0][0] = 0$, with $dp[0][j] = \infty$ for $j > 0$.

$$dp[i+1][j] = \min\left(dp[i][j-1],\ dp[i][j] + t[i+1]\right)$$

Since $dp[w+j][j]$ is the minimum time to process $w+j$ reels while skipping j of them, it corresponds to watching exactly w reels. Therefore,

$$f_w = \min_{0 \le j \le k} dp[w+j][j].$$

These transitions consider every valid choice of watched and skipped reels. Conversely, every state uses at most k skips and therefore describes a valid viewing sequence.

Time complexity: $\mathcal{O}(n^2 + q \log n)$ under this subtask's constraints

Subtask 3

Additional constraints: No additional constraints

By an exchange argument, it is optimal to skip longer reels rather than shorter ones.

To watch w reels, consider the first $w+k$ reels and skip the k longest reels in this prefix. Then f_w is the sum of these $w+k$ reel lengths minus the sum of the k largest lengths among them.

Process the reels from left to right while maintaining the prefix sum and using a min-priority queue containing the k largest lengths seen so far. When a new length exceeds the smallest value in the queue, replace that value. At prefix length $L \ge k+1$, the prefix sum minus the queue sum gives f_{L-k}.

For $w > n-k$, the required prefix of $w+k$ reels is longer than the array. We handle this by appending k reels of length ∞.

Time complexity: $\mathcal{O}(n \log n + q \log n)$

Subtask	Answer
1	874875513418
2	25111505
3	917780454100

Task 19: `Dictator`

Authored by: Siew Jing Hong (siewjh)

Prepared by: Siew Jing Hong (siewjh)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $1 \leq n \leq 5000$

We can simulate the process directly. For each round, scan the current array and keep its first element. Keep every later element that is larger than the most recently kept element, and append every removed element to the array for the next round. Count the round even when no element is removed, as required by the statement.

Time Complexity: $\mathcal{O}(n^2)$

Subtask 2

Additional constraints: There exists exactly one value of i such that $a[i] < a[i+1]$

The array consists of two strictly decreasing runs. Let the unique ascent be between positions k and $k+1$. If a decreasing subsequence uses $a[j]$ as its last element from the first run, it can include every element $a[1], \ldots, a[j]$, followed by every element in the second run that is smaller than $a[j]$. It may also use only the second run.

Trying all choices of j gives the longest decreasing subsequence. Since both runs are decreasing, a pointer into the second run only moves forward as j increases, so all candidates can be evaluated in $\mathcal{O}(n)$ time. As shown in the full solution, this longest decreasing subsequence length is exactly the number of Stalin-sort rounds.

Time Complexity: $\mathcal{O}(n)$

Subtask 3

Additional constraints: No additional constraints

In each round, the kept elements form a strictly increasing subsequence. Therefore, from any strictly decreasing subsequence, at most one element can be kept per round.

Hence, if the longest decreasing subsequence has length L, at least L rounds are needed.

Conversely, suppose an element is only kept in round r. In every previous round, it must have been blocked by a larger element appearing before it. Tracing these blocking elements backwards gives a strictly decreasing subsequence of length r, since removed elements are re-appended preserving their relative order (Subtask 1), so each traced blocker also precedes the element in the original array.

Therefore, no element can survive beyond round L, so at most L rounds are needed.

Thus, the number of rounds is exactly the length of the longest decreasing subsequence. This length can be computed using the standard longest decreasing subsequence algorithm, which can be searched online.

Time Complexity: $\mathcal{O}(n \log n)$

Subtask	Answer
1	38289
2	10044590
3	9766326

Task 20: `House Visit`

Authored by: Siew Jing Hong (siewjh)

Prepared by: Brian Lee (penguin133)

Editorial written by: Ziv Lim (misalignedDiv)

Note

Include times 0 and y with the inspection times. These times partition the day into intervals whose endpoints are times when Jayden is at home. For an interval of length d, define its excursion budget as $\lfloor \frac{d}{2} \rfloor$, since a round trip reaching distance q from home takes $2q$ seconds.

Let B_k be the k-th largest excursion budget, with $B_k = 0$ if there are fewer than k intervals. Errands at home are always completed and are counted separately.

Subtask 1

Additional constraints: $x = 1, 1 \leq n, m \leq 10^6$

All errands lie at or to the right of Jayden's home. During an excursion, Jayden can travel to the farthest errand within his budget and complete every closer errand along the way.

Let B_1 be the largest excursion budget. Consider any other budget B_j. Since $B_j \leq B_1$, every errand that can be completed using B_j can also be completed using B_1. Therefore, the smaller budgets cannot help Jayden complete any additional errands, and only an interval with the largest budget matters. (See Figure 5 for illustration)

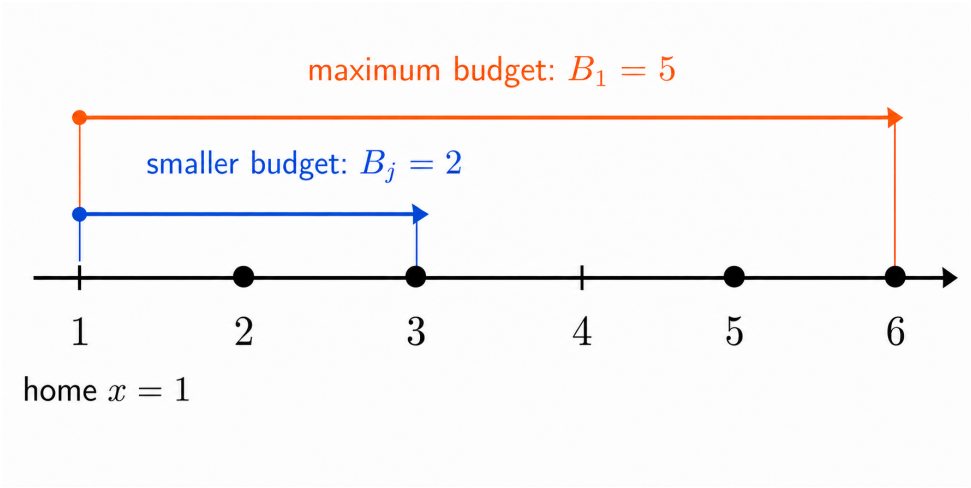


Figure 5: Every errand reachable within a smaller budget $B_j = 2$ is also reachable within the largest budget $B_1 = 5$, so only the largest budget matters.

Jayden can reach every position p_i satisfying

$$p_i - x \leq B_1.$$

Compute the gaps between consecutive required times, find the largest excursion budget, and use it to count the reachable errands.

Time Complexity: $\mathcal{O}(n + m)$

Subtask 2

Additional constraints: $1 \leq n, m \leq 5000$

We label errands to the left of Jayden's home as L and errands to the right as R.

First, Jayden may visit the two sides during separate intervals. Only the two largest budgets, B_1 and B_2, are needed. This is because all errands on one side can be completed by travelling to the farthest reachable errand on that side. Hence, at most one interval is needed for each side.

This means that there will only be two cases when visiting the two sides in separate intervals:

1. Using B_1 for L and B_2 for R
2. Using B_2 for L and B_1 for R

This essentially reuses the code from Subtask 1, run four times.

Alternatively, Jayden may visit both sides during the same interval. Suppose his leftmost and rightmost turning points are p_l and p_r, where $p_l < x < p_r$.

The journey from home to one turning point, then to the other, and finally back home has length $2(p_r - p_l)$. Therefore, this journey fits within the largest interval exactly when $p_r - p_l \leq B_1$.

Brute-force every possible left turning point. For each one, scan the errands to find the farthest possible right turning point and count all errands between them. Take the maximum over the same-interval case and the two separate-interval cases.

Time Complexity: $\mathcal{O}(n^2 + m)$

Subtask 3

Additional constraints: $1 \leq n, m \leq 10^6$

The $\mathcal{O}(n^2)$ factor in Subtask 2 comes from repeatedly searching for the right turning point in the same-interval case. Sort the errand positions in ascending order. Consider a left turning point $p_l < x$, and let $p_r > x$ be the farthest right turning point satisfying $p_r - p_l \leq B_1$.

As the left turning point moves farther left, p_l decreases. Consequently, less of the budget remains for travelling to the right, so p_r can only move left or remain unchanged.

This allows us to use the two-pointer technique. Process the possible left turning points from nearest to farthest from home. Whenever $p_r - p_l > B_1$, move the right pointer r to the left until the journey becomes feasible again. For each feasible pair of indices (l, r), all errands from index l to index r are completed, giving $r - l + 1$ errands.

Each pointer moves at most n times, so the same-interval case takes $\mathcal{O}(n)$ time after sorting. The two separate-interval cases are computed as in Subtask 2.

Time complexity: $\mathcal{O}(n \log n + m)$

Subtask	Answer
1	628929
2	9523
3	690389

Task 21: `Spinning Table`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $1 \leq n \leq 10$ for each test case

This subtask is meant for manual trial and error to find the best sequence of clockwise and anticlockwise rotations for each small test case. The experimentation can in fact provide valuable insights to the full solution.

Subtask 2

Additional constraints: $1 \leq n \leq 5000$ for each test case

We represent each orientation using its signed displacement from the starting position. Negative displacements represent anticlockwise rotations, while positive displacements represent clockwise rotations.

For each i, let $d[i] = (x[i] - i + n) \bmod n$.

Person i can obtain the required dish either by rotating $d[i]$ units clockwise or $d[i] - n$ units anticlockwise. Hence, we increment both $\text{point}[d[i]]$ and $\text{point}[d[i] - n]$.

Let $L \leq 0$ and $R \geq 0$ be the furthest positions reached in the anticlockwise and clockwise directions, respectively. The table visits every position in $[L, R]$. Using prefix sums, we can count the people covered by $[L, R]$ in $\mathcal{O}(1)$ time. The range is valid if this count is at least n.

If the table visits L before R, the time taken is $R - 2L$. Otherwise, use $2R - L$. (See Figure 6.)

Therefore, the minimum time for this range is

$$\min(R - 2L,\ 2R - L)$$

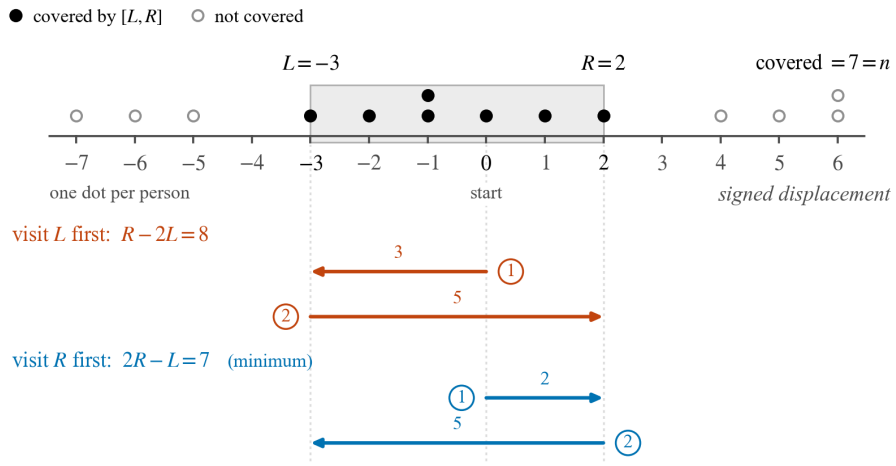


Figure 6: Signed displacements as points on a number line, with the range $[L, R]$ covering the chosen points. Starting from 0, the table either visits L first (time $R - 2L$) or visits R first (time $2R - L$).

We try every possible pair (L, R), where $L \in [-(n-1), 0]$ and $R \in [0, n-1]$ and take the minimum value over all valid ranges.

Time complexity: $\mathcal{O}(n^2)$

Subtask 3

Additional constraints: $1 \leq n \leq 10^6$ for each test case

For a fixed L, only the smallest valid R matters, since increasing R can only increase the time taken.

As L moves left, the range gains more anticlockwise positions. Hence, the smallest valid R can only move left or remain unchanged. This allows us to use the two-pointer technique.

Initially, set $L = 0$ and $R = n - 1$. For each value of L:

1. Continue decreasing R while the range $[L, R-1]$ still covers all n people, ensuring $R \geq 0$
2. Update the answer with $\min(R - 2L,\ 2R - L)$

The number of people covered by each range can be checked in $\mathcal{O}(1)$ using prefix sums. Both L and R move at most n times.

Time complexity: $\mathcal{O}(n)$

Subtask	Answer
1	402
2	274895
3	51999891

Task 22: `A Certain Scientific Committee`

Authored by: Siew Jing Hong (siewjh)

Prepared by: Siew Jing Hong (siewjh)

Editorial written by: Ziv Lim (misalignedDiv)

Note

For every problem $i > 1$, define

$$A_i = a[i] - a[1] \qquad \text{and} \qquad B_i = b[i] - b[1].$$

The difference between the final rating of problem i and that of problem 1 is

$$D_i(x) = xA_i + (1 - x)B_i = B_i + x(A_i - B_i).$$

Problem i is ranked above problem 1 exactly when $D_i(x) > 0$. Therefore, the rank of problem 1 is one plus the number of problems above it.

When we write $x = 0^+$, we mean any sufficiently small positive legal value of x. Similarly, when processing an intersection, we consider the open intervals immediately before and after it, since the intersection value itself produces a tie and cannot be chosen.

Furthermore for implementation, fractions / pairs are used instead of floats to prevent any floating-point errors. Using cross-multiplication to compare fractions will be used a lot later too.

Subtask 1

Additional constraints: $a[i] > a[1]$ for all $i > 1$.

The value $x = 0$ may be illegal because some problems may have $b[i] = b[1]$. We therefore begin at $x = 0^+$.

Suppose problem i is ranked above problem 1, so $b_i > b_1$. Due to the constraint, problem i remains above problem 1 at $x = 1$.

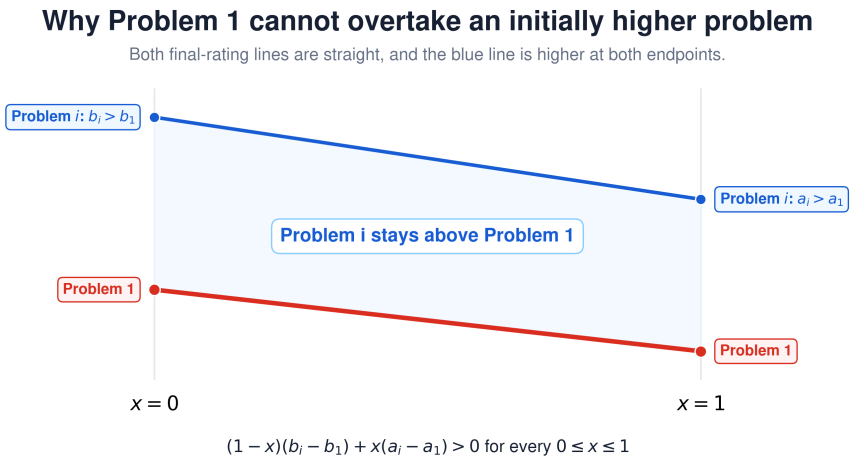


Figure 7: Both final-rating lines are straight, and the blue line is higher at both endpoints, so problem i stays above problem 1 for every $0 \le x \le 1$.

Problem i is above problem 1 for all sufficiently small positive x exactly when $b[i] > b[1]$, or when $b[i] = b[1]$ and $a[i] > a[1]$. Use this to calculate problem 1's rank at $x = 0^+$.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $n \le 5000$

Problem 1's rank changes only when it overtakes or is overtaken by another problem.

For each problem $i > 1$, the final ratings of problems i and 1 are equal when

$$B_i + x(A_i - B_i) = 0.$$

Solving this equation gives

$$x = \frac{B_i}{B_i - A_i}.$$

Collect all intersection values strictly inside $(0, 1)$, sort them, and group equal intersection values. These values divide the range into $\mathcal{O}(n)$ open intervals. Within each interval, no problem intersects problem 1, so its rank remains constant.

Choose one representative value of x from each interval and compare problem 1's final rating with the final ratings of all n problems. The smallest rank obtained is the answer.

The statement forbids ties between every pair of problems. Fortunately, this is easy to handle: each pair of problems can tie at most once, so only finitely many values inside an interval are forbidden. Therefore, every open interval contains a legal value of x. Moreover, an intersection between two problems other than problem 1 does not change problem 1's rank.

There are $\mathcal{O}(n)$ intervals, and calculating the rank in each interval takes $\mathcal{O}(n)$ time.

Time complexity: $\mathcal{O}(n^2)$.

Subtask 3

Additional constraints: No additional constraints

Instead of recalculating the rank in every interval, sweep through the intersections involving problem 1 while maintaining count_{curr}, the number of problems currently ranked above it.

First, compute count_{curr} at $x = 0^+$ using the rule from Subtask 1.

For every problem $i > 1$:

- If $B_i > 0$ and $A_i < 0$, problem i intersects problem 1 while moving from above it to below it, so count_{curr} decreases by 1.
- If $B_i < 0$ and $A_i > 0$, problem i intersects from below problem 1 to above it, so count_{curr} increases by 1.
- Otherwise, problem i has no intersection strictly inside $(0, 1)$ and can be ignored after calculating the initial count.

Sort these fractions using cross-multiplication to compare them. If several problems intersect problem 1 at the same value of x, process all their changes together. The intersection value itself is not legal, so applying only some of the changes would not represent any valid interval.

Process the distinct intersection values in increasing order. After applying all changes at one intersection value, count_{curr} gives the number of problems above problem 1 in the interval immediately to its right.

The result would be

$$1 + \min_{0 \le k \le m} \text{count}_k,$$

where count_0 is the initial count at $x = 0^+$, and count_k is count_{curr} after processing the k-th distinct intersection value.

Time complexity: $\mathcal{O}(n \log n)$

Subtask	Answer
1	3037845
2	38235
3	7448531

Task 23: `Bookshelf`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Quinn Chua (sheep)

Editorial written by: Ziv Lim (misalignedDiv)

Subtask 1

Additional constraints: $1 \leq n \leq 300$, w_i and h_i are strictly decreasing

Every earlier book is both wider and taller than every later book, so every consecutive pair of books is incompatible.

We can make the bookshelf valid either by inserting $n - 1$ new books, one between each consecutive pair, or by removing all but one book.

The answer is therefore $n - 1$ for each test case.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $1 \leq n \leq 300$

Suppose books i and j are kept, while every book between them is removed. If they are compatible, they may be placed next to each other immediately. Otherwise, one inserted book is necessary and sufficient. For example, we may insert a book with dimensions (w_i, h_j).

Let $dp[i][j]$ be the minimum number of operations needed to make the books from i to j valid, under the condition that books i and j are both kept.

If every book strictly between i and j is removed, the cost is

$$j - i - 1 + c(i, j)$$

where $c(i, j)$ is 0 if books i and j are compatible, and 1 otherwise.

Alternatively, we may keep an intermediate book k. The problem then splits into the two independent intervals $[i, k]$ and $[k, j]$, giving the transition

$$dp[i][j] = \min\left(j - i - 1 + c(i, j),\ dp[i][k] + dp[k][j]\right)$$

for all $i < k < j$, where the base case is $dp[i][i] = 0$.

The answer is then

$$\min_{1 \le i \le j \le n} \left((i - 1) + (n - j) + dp[i][j]\right)$$

Time complexity: $\mathcal{O}(n^3)$

Subtask 3

Additional constraints: $1 \le n \le 10^6$

Observe that one operation can remove at most one incompatibility.

An insertion affects only one existing adjacent pair, so it can remove at most one incompatibility.

Now consider removing a book. This removes at most two adjacent pairs and creates a new pair between its two neighbours. If both removed pairs were incompatible, then the left neighbour is both wider and taller than the removed book, which is in turn both wider and taller than the right neighbour. Hence, the newly adjacent pair is also incompatible. Therefore, removing a book also eliminates at most one incompatibility overall.

Thus, if the original bookshelf contains k incompatible adjacent pairs, at least k operations are required.

This bound can always be achieved using only insertions. For every incompatible pair of consecutive books i and $i + 1$, insert a book with dimensions $(w[i], h[i + 1])$ between them. The inserted book is at least as wide as book i, while book $i + 1$ is at least as tall as the inserted book. Hence, both new adjacent pairs are compatible.

Therefore, the answer is simply the number of indices i such that the consecutive pair of books i and $i + 1$ is incompatible, that is, $w[i] > w[i + 1]$ and $h[i] > h[i + 1]$.

Time complexity: $\mathcal{O}(n)$

Subtask	Answer
1	1618
2	559
3	374642

Task 24: `Pawns`

Authored by: Quinn Chua (sheep)

Prepared by: Brian Lee (penguin133)

Editorial written by: Ziv Lim (misalignedDiv)

Note

We determine the winner from the total number of moves each player can make. Brian wins if he can make strictly more moves than Ryan. If they can make the same number of moves, Ryan wins because Brian moves first and is therefore the first player unable to move.

Subtask 1

Additional constraints: For all $1 \leq i \leq m-1$, if $c[i] = \texttt{R}$, then $c[i+1] = \texttt{R}$

The pawns appear in the following order.

`BBB...BBBRRRR..RRR`

Let p and q be the numbers of blue and red pawns, respectively. Let l be the position of the rightmost blue pawn and r the position of the leftmost red pawn. Define

$$g = r - l - 1,$$

the number of empty squares between them.

It is optimal for Brian to move the rightmost blue pawn. Moving any other blue pawn does not reduce the space available to Ryan, whereas moving the rightmost blue pawn claims one of the shared empty squares before Ryan can. By the same argument, Ryan should always move the leftmost red pawn.
Thus, the g empty squares are claimed alternately. Since Brian moves first, he claims $\left\lceil \frac{g}{2} \right\rceil$ of them and Ryan claims $\left\lfloor \frac{g}{2} \right\rfloor$. Therefore, the two front pawns eventually meet at

$$X = l + \left\lceil \frac{g}{2} \right\rceil \quad \text{and} \quad Y = r - \left\lfloor \frac{g}{2} \right\rfloor.$$

These two pawns now block the opposing colour from moving any further. Since pawns of the same colour do not block one another, every blue pawn can eventually move to X, while every red pawn moves to Y.

Time complexity: $\mathcal{O}(m)$

Subtask 2

Additional constraints: For all $1 \leq i \leq m-1$, if $c[i] = \texttt{R}$, then $c[i+1] = \texttt{B}$

This subtask is handled by the general solution in Subtask 3, so its explanation is omitted.

Subtask 3

Additional constraints: No additional constraints

The pawn sequence now consists of several blocks of the form from Subtask 1:

```
(R...)(BBB...RRR...)(BBB...RRR...)(BBB...RRR...)(...B)
```

Red pawns at the extreme left are never blocked by blue pawns, so they can walk to square 1, contributing $\sum(d[i]-1)$ moves to Ryan. Similarly, blue pawns at the extreme right can walk to square n, contributing $\sum(n-d[i])$ moves to Brian.

Between two blocks, the red pawns move left while the blue pawns move right. Thus, different blocks never interact and can be considered independently.

As in Subtask 1, while a gap remains, it is optimal to move one of the two front pawns. Moving any other pawn does not claim additional space and can always be done later.

For a block with an even-sized gap, both players claim the same number of squares. For a block with an odd-sized gap, the player who moves in that block first claims one additional square. Because resolving an odd-sized gap takes an odd number of moves, the other player moves first in the next odd-gap block.

Suppose a block contains p blue pawns and q red pawns, and let

$$t = p + q$$

If Brian claims the extra square, he gains p moves and denies Ryan q moves, for a total advantage of t. The roles reverse if Ryan claims it. Thus, process all odd-gap blocks in decreasing order of t; Brian and Ryan receive the extra square alternately, starting with Brian.

After determining who receives the extra square in each block, calculate both players' total moves for the blocks using the method from Subtask 1, accounting for the red pawns at the extreme left and blue pawns at the extreme right too.

Time complexity: $\mathcal{O}(m \log(m))$

Subtask	Answer
1	RBRBBRBRRR
2	RRBRBBRRBR
3	BRBBRBRBBR

Task 25: `Turbines`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Jayden Tiew (Shadow1)

Editorial written by: Ziv Lim (misalignedDiv)

For a chosen set of turbine nodes S, a turbine at node v receives strength

$$k - |\{u \in S : u \text{ is an ancestor of } v\}|.$$

Hence the total electricity is $|S|k$ minus the number of selected ancestor-descendant pairs.

Subtask 1

Additional constraints: The tree is a line; $u_i = i$, $v_i = i + 1$ for all $1 \le i \le n - 1$

Every pair of selected nodes is an ancestor-descendant pair. Therefore, placing exactly q turbines produces

$$qk - \frac{q(q-1)}{2},$$

where $q = \min(m, k)$, so that no turbine contributes negative energy.

Subtask 2

Additional constraints: $n \le 2000$

The Subtask 2 solution is more complicated than the Subtask 3 solution, and its observations are not needed for Subtask 3.

For each node u, define $dp[u][q]$ as the maximum energy obtainable from the subtree of u when exactly q nodes in that subtree are selected. Process the tree from the leaves upwards. For each node u, initialise $dp[u][0] = 0$, and let $sz[u] = 0$.

Process the children of u one by one during the DFS. Suppose that child v has already been processed. Merge $dp[v]$ into $dp[u]$ using the following knapsack transition:

$$new[a+b] = \max(new[a+b],\ dp[u][a] + dp[v][b]) \quad \text{for } 0 \le a \le sz[u] \text{ and } 0 \le b \le sz[v],$$

where a is the number of selected turbines among the children processed so far, and b is the number selected in the subtree of v.

After this merge, replace $dp[u]$ with new and increase $sz[u]$ by $sz[v]$.

Once all children have been merged, we may also choose to place a turbine at u itself, where the transition is

$$dp[u][q+1] = \max(dp[u][q+1],\ dp[u][q] + k - q),$$

looping q from $sz[u]$ down to 0. Do not forget to increase $sz[u]$ by 1 after this.

The answer would be

$$\max_{1 \le q \le m} (dp[1][q])$$

because taking fewer than m turbines can be better if taking another would lead to negative electricity.

Although the time complexity looks like $\mathcal{O}(nm^2)$, the time complexity is actually $\mathcal{O}(n^2)$. The DP array of a subtree has length proportional to its size. Across all merges, each pair of nodes is considered at most once, when distributing them between different child subtrees of their lowest common ancestor. There are only $\mathcal{O}(n^2)$ such pairs in total. This is famously known as distribution DP.

Time Complexity: $\mathcal{O}(n^2)$

Subtask 3

Additional constraints: No additional constraints

Observation 1: There exists an optimal selection in which, whenever a node is selected, every node in its subtree is also selected.

Suppose a selected node u has an unselected proper descendant v, and replace u with v (see Figure 8 for an illustration). Consider how this affects the number of selected ancestor-descendant pairs:

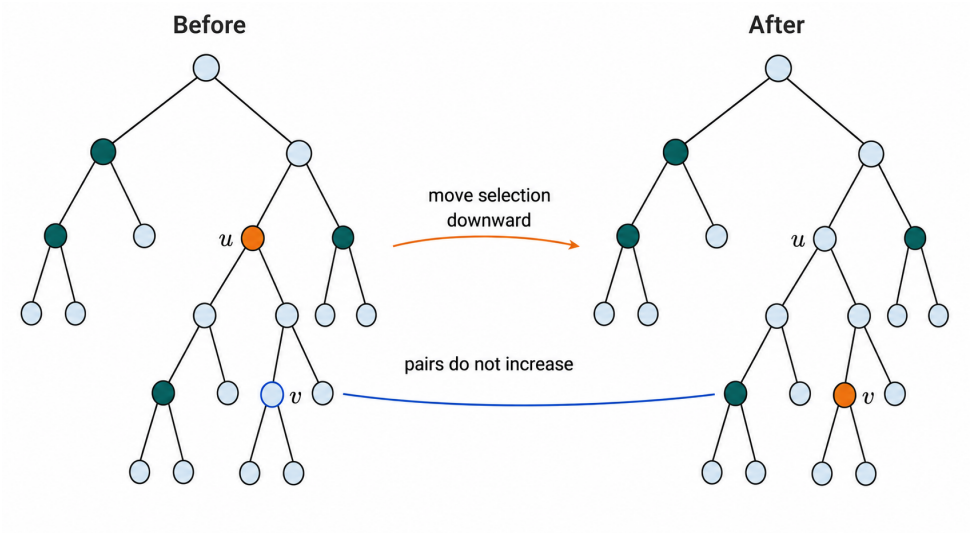


Figure 8: Moving the selection downward from u to its descendant v: the number of selected ancestor-descendant pairs does not increase.

1. Every selected ancestor of u is also an ancestor of v, so these pairs are preserved.
2. If a selected node lies between u and v, the pair involving u is replaced by a pair involving v.
3. Every selected descendant of v was also a selected descendant of u, so these pairs are also preserved.
4. A selected node that lies inside the subtree of u and is neither an ancestor nor a descendant of v forms an ancestor-descendant pair with u, but not with v. Such a pair disappears.

Therefore, moving the selected node from u down to v cannot increase the number of selected ancestor-descendant pairs. The number of selected nodes remains unchanged, so the total electricity does not decrease.

Observation 2: Building on Observation 1, process the nodes from the leaves upwards and select them greedily in increasing order of subtree size.

Let $sz[u]$ denote the number of nodes in the subtree of u. From Observation 1, if u is selected, all $sz[u] - 1$ proper descendants of u are also selected. This means that there are $sz[u] - 1$ additional ancestor-descendant pairs if u is selected, so it contributes $k - (sz[u] - 1)$ electricity.

It follows that we should greedily select nodes in this order to minimise the deduction. This order also satisfies Observation 1, because every proper descendant v of u has $sz[v] < sz[u]$. Only select a node if it contributes a positive amount of electricity.

Time Complexity: $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$ depending on implementation

Subtask	Answer
1	2699844301695733
2	16687004255851
3	2252951788520096