Singapore Informatics League 2025

Task Editorial

December 2025

Contents

Level 1

Task 1: `Broken Key`

Authored by: Brian Lee (penguin133)

Prepared by: Brian Lee (penguin133)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $n = 67$ and $k = 8$

Observe that if n does not contain the digit k, then the answer is 1. This is the solution for subtask 1, where 67 can be directly formed without the digit 8.

Subtask 3

Additional constraints: $n = 20490$ and $k = 0$

In subtask 3, 20490 can be expressed as $19379 + 1111$.

Subtask 2

Additional constraints: $n = 199$ and $k = 1$

In subtask 2, 199 cannot be formed with 2 non-negative integers that both do not contain $k = 1$, because at least one of the integers will be in the range $100 \ldots 199$. Hence, the minimum answer for this subtask is 3, expressed as $99 + 98 + 2$.

Subtask	Answer
1	1
2	3
3	2

Task 2: `Cats and Parking`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Chua Wee Chong (sheep) and Ernest Kiew (Shor the Duck)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $n = 15$, all parking spots are empty.

In subtask 1, all spots are empty. As a result, the number of consecutive empty spots on all days is equal to the length of the parking lot which is $n = 15$.

Subtasks 2 and 3

Additional constraints for Subtask 2: $n = 15$

Additional constraints for Subtask 3: $n = 26$

We note that any range of consecutive empty spots in the original array could be a possible answer.

Consider a range of spots to be almost-consecutive if linking the first and last spots together would make the spots consecutive. Almost-consecutive ranges of empty spots could also be possible answers.

For example, if there are 2 empty spots at the front of the parking lot, and 4 empty spots at the back, we can wait for 4 days for all these empty spots to "move" towards the start, making them consecutive.

Hence, the answer is the maximum length over all consecutive and almost-consecutive ranges of empty spots.

In subtask 2, the answer is 3, with a consecutive range of empty spots.

In subtask 3, the answer is 5, with an almost-consecutive range of empty spots.

Subtask	Answer
1	15
2	3
3	5

Task 3: `Distribution`

Authored by: Chua Wee Chong (sheep)

Prepared by: Brian Lee (penguin133) and Chua Wee Chong (sheep)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $n = 2$ and $t = 100$

For subtask 1, since there are only $n = 2$ people, we should try to split the 100 apples as evenly as possible. Since both cannot each have 50 apples, the next best distribution is for one person to have 49 apples and the other to have 51.

Hence, the answer for subtask 1 is 49.

Subtask 2

Additional constraints: $n = 10$ and $t = 55$

Consider trying to give the person with the least apples 1 apple. Then, to minimise the total number of apples given, the next person should have 2, third person should have 3, and so on until the last person who gets $n = 10$ apples. The total number of apples distributed turns out to be exactly equal to $m = 55$.

It follows that the answer is 1, since it is impossible for any person to receive more apples.

Subtask 3

Additional constraints: $n = 20$ and $t = 300$

Suppose the person with the smallest number of apples has x apples. For a valid distribution with a minimum sum of apples, we would need the other people to have $x + 1$, $x + 2$, $\cdots$, $x + n - 1$ apples respectively.

Effectively, we want the largest x such that $x + (x+1) + (x+2) + \cdots + (x+n-1) \leq s$.

Trying a few values of x in subtask 3, we can obtain the answer 5.

Subtask	**Answer**
1	49
2	1
3	5

Task 4: `Duck Fight`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $n = 12$ and $a[1] = a[2] = \cdots = a[n]$

All cards have equal power, which means that the total power of any person is equal to the power of a card multiplied by the number of cards taken.

In order to have a **strictly greater** power than the other person, you must take **strictly more** cards.

You can take 7 cards at minimum, leaving 5 cards for oolimry. Hence, 7 is our final answer.

Subtask 2 and 3

Additional constraints for Subtask 2: $n = 5$

Additional constraints for Subtask 3: $n = 12$

To find the optimal answer, you should play the role of Shor.

To maximise your total power, you should take the highest cards first. As this also prevents oolimry from taking those higher power cards, this is the optimal strategy.[1]

As such, if Shor decides to choose x cards, he does no worse by taking the x cards with highest power.

By trying various values of x, you can obtain the answer. You can try all values of x if you have more time, or you can try some middle value of x, and adjust upwards or downwards until you find the answer (similar to a binary search).

[1] Mathematical proof by Exchange Argument omitted for simplicity.

Subtask	Answer
1	7
2	2
3	4

Task 5: `Bench`

Authored by: Brian Lee (penguin133)

Prepared by: Brian Lee (penguin133)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $n = 4278$ and $m = 2$ There are only $m = 2$ people, so they should be seated on both ends of the row of seats. Since the first person is at seat 1 and the other person is at seat n, $d = n - 1$.

The answer is $4278 - 1 = 4277$.

Subtasks 2 and 3

Additional constraints for Subtask 2: $n = 8$ and $m = 3$

Additional constraints for Subtask 3: $n = 9996$ and $m = 35$

Suppose we want to construct a configuration with an answer of d. One such contruction is for person 1 to sit at seat 1, then the next person to sit at seat $d + 1$, then $2d + 1, \ldots, (m - 1)d + 1$, where each person sits d spaces away from the previous person.

This means that we would require at least $(m-1)d+1$ seats to achieve an answer of d; in other words. $(m - 1)d + 1 \leq n$ must hold for a value of d to be achievable.

Rearranging terms, we get $d \leq \dfrac{n-1}{m-1}$, and since d has to be an integer, the maximum value of d is $\left\lfloor \dfrac{n-1}{m-1} \right\rfloor$.

For subtask 2, we have $d = \left\lfloor \dfrac{8-1}{3-1} \right\rfloor = 3$.

For subtask 3, we have $d = \left\lfloor \dfrac{9996-1}{35-1} \right\rfloor = 293$.

Subtask	Answer
1	4277
2	3
3	293

Task 6: `Schedule`

Authored by: Brian Lee (penguin133)

Prepared by: Brian Lee (penguin133) and Chua Wee Chong (sheep)

Editorial written by: Brian Lee (penguin133)

Subtask 2

Additional constraints: $n = 30$, $k = 6$ and $s[i] = \texttt{X}$ for all $1 \leq i \leq n$

In subtask 2, Alice is undecided on every day. Alice should work for k days and rest for 1 day repeatedly to maximise the number of working days.

We can split the n days into blocks of $k + 1$ days, where in each block she works for k days. For the leftover days, Alice will also work since the last day before these leftover days is a rest day.

Hence, Alice will work for $6 \cdot \left\lfloor \frac{30}{6+1} \right\rfloor + \left(30 - (6+1) \cdot \left\lfloor \frac{30}{6+1} \right\rfloor\right) = 26$ days.

Subtask 1 and 3

Additional constraints for Subtask 1: $n = 10$ and $k = 3$

Additional constraints for Subtask 3: $n = 30$ and $k = 4$

To solve the full problem, we can adapt the above idea for an arbitrary string. We can perform the replacement as follows:

- Process each day one by one, in sequence. If the current day is undecided, make Alice work on that day if it does not result in her working more than k days consecutively. Otherwise, she should rest on that day.

This is known as a greedy algorithm, and we will prove that it will achieve the largest possible number of work days. (Note that rigorously proving the solution is not expected of contestants during the contest.)

Let the work schedule generated by the greedy algorithm be G, and let any optimal work schedule be O. Assuming that $O \neq G$, we will show that O can be transformed into G without reducing the number of days worked, which implies that the number of days worked in O is equal to the number of days worked in G.

Consider the first day i where $G[i] \neq O[i]$. If $G[i] = \texttt{R}$ and $O[i] = \texttt{W}$, it would mean that O violated the condition of not working for more than k days, since the greedy algorithm would have chosen to work on day i otherwise. If $G[i] = \texttt{W}$ and $O[i] = \texttt{R}$, then there must exist a day $j > i$ where $G[i] = \texttt{R}$ and $O[i] = \texttt{W}$ (otherwise, O has strictly less days worked than G, and O is sub-optimal). Choose the smallest such j. Then, swap $O[i]$ with $O[j]$ to obtain a new optimal solution O'. O' does not violate the condition,[2] and contains the same number of working days as O. By repeatedly performing these swaps, we can eventually transform O into G.

The above proof is known as an *exchange argument*, a common proof technique for greedy algorithms.

Using this algorithm, we obtain an answer of 7 for subtask 1 and 20 for subtask 3.

Subtask	Answer
1	7
2	26
3	20

[2] This is because $G[i+1] = O[i+1], G[i+2] = O[i+2], \ldots, G[j-1] = O[j-1]$.

Task 7: `Typing Test`

Authored by: Chua Wee Chong (sheep)

Prepared by: Chua Wee Chong (sheep)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $n = 9$, there is exactly one '`a`' in s

Input:

```
9
AAAAaAAAA
```

We use the one allowed change to change the middle '`a`' character into '`A`'.

This allows us to "bridge" the gap between the two continuous segments of '`A`', so we can complete the whole string with Caps Lock on.

However, since Caps Lock is initially off, we press it once to turn it on, hence the answer is 1.

Subtasks 2 and 3

Additional constraints: $n = 20$

Input for Subtask 2:

```
20
aAAaAAAaaaAAaaAAaaAA
```

Input for Subtask 3:

```
20
aaAAaaAAaaAAaaAAaaAA
```

Let's start by counting the number of Caps Lock presses each takes **before** modifying the string, which is the number of times we see a change in capitalisation between adjacent letters. (Note: if the starting letter was an '`A`', we would add one as well, since Caps Lock is initially off, but this was not in our inputs.)

This count is 9 for both inputs.

In subtask 2, we can use the same idea of "bridging" the gap, where we change the 4th character from '`a`' to '`A`', reducing our Caps Lock count by two, reaching an answer of 7.

In subtask 3, there are no "gaps" of size 1 that we can bridge.
Notice that if we change any letter, it would not change the number of changes in capitalisation between adjacent letters, as it would only move the place where it changes by one space.

Hence, any changes would not improve our answer, and we can either do a meaningless change or none at all, resulting in an answer of 9.

Subtask	Answer
1	1
2	7
3	9

Task 8: `Ducks, Rabbits and Weirdos`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Brian Lee (penguin133) and Chua Wee Chong (sheep)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $h = 8$ and $l = 40$

There are 8 heads and 40 legs. We can try every possible number of weirdos (from 1 to 8) and check if there exists some number of ducks and rabbits that satisfy h heads and l legs.

Trying every possibility gives an answer of 6, and there are 2 ducks and 0 rabbits.

Subtask 2

Additional constraints: $h = 20$ and $l = 119$

We need to narrow down our search range for the number of weirdos, since we now have 20 heads and 119 legs. Notice that if we have w weirdos and each weirdo has w legs, the weirdos would have a total of w^2 legs. This means that $w^2 \leq l$ for a valid solution to be possible, and hence we can narrow our search range to $1 \leq w \leq \sqrt{l}$.

Trying every possibility in this range gives an answer of 9, and there are 4 ducks and 7 rabbits.

Subtask 3

Additional constraints: $h = 407$ and $l = 2819$

h and l are now both huge. Immediately, using the idea from subtask 2, we can narrow down the range of w to $1 \leq w \leq \sqrt{l}$, or $1 \leq w \leq 53$.

Now, let us fix w, the number of weirdos, and let x be the number of ducks and y be the number of rabbits.

For w to produce a valid solution, it must hold that:

- $x + y = h - w$, and
- $2x + 4y = l - w^2$.

Looking at the second equation, since the left hand side is even, $l - w^2$ must be even for there to be integer solutions of x, y. Hence w must be odd.

Next, since we know that $2(h - w) \leq 2x + 4y \leq 4(h - w)$, it holds that $2(h - w) \leq l - w^2 \leq 4(h - w)$.

Hence, we can quickly derive a range of w that satisfies this using manual binary search or by trying all possibilities (if you are resilient).

The final answer for subtask 3 is 37, where a possible solution is 15 ducks, 355 rabbits and 37 weirdos.

Subtask	Answer
1	6
2	9
3	37

Level 2

Task 9: AANDNANDT

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Kang Yiming (kym2006)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $|s| = 29$

Since the length of the string is small, it is enough to manually examine the 1st, 4th, 7th, 10th, $\cdots$, 22nd, 25th, 28th characters for 'A'.

Subtask 2

Additional constraints: $|s| = 233$

To make it easier, you should split the process into finding all occurrences of "AND" and deleting them, then counting the number of 'A's left in the string.

Subtask 3

Additional constraints: $|s| = 199997$

There are multiple interesting solutions.

Solution 1: Ctrl + F

By opening the input in some text editors (unfortunately you'll need to find one that works for you), you can use the shortcut Ctrl + F to count all occurrences of "AND" and all occurrences of 'A'.

Then, the answer is just the number of 'A's minus the number of "AND"s, as each "AND" only contains one 'A'.

Solution 2: Code

You can perform the same occurrence checking as in the Ctrl + F solution, or just add one to a counter every time the 1st, 4th, 7th, $\cdots$ character is an 'A'.

Solution 3: Algebra

If we start with a string of length n, $n-1$ "AND"s are generated, resulting in a length of $n+3(n-1)=4n-3$.

As such, we know that the initial string length is $(199997+3)/4 = 50000$, and hence the number of "AND"s is 49999.

The number of "A"s can be found through some random website, which counts 74962. (I just searched `find number of each character` on Google)

Hence, the answer is $74962-49999=24963$.

Subtask	Answer
1	3
2	18
3	24963

Task 10: `90RP Manifestation`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $tc = 3$, $x = 0$, and $y = 0$

Since both $x = 0$ and $y = 0$, every operation aside from deleting a character is free.

It is easy to show that we need at least $n - 2$ cost, as we need to reduce the length of the string from n to 2, and a delete is the only way to do that, costing 1 for each character deleted.

Then, we can always achieve $n - 2$ cost, as we can just keep only the first two characters, and turn them into `90` without extra cost.

Hence, $n - 2$ is the answer for all test cases.

Subtask 2 and 3

Additional constraints for Subtask 2: $tc = 3$, $n = 10$, $x = 1$, and $y = 1$

Additional constraints for Subtask 3: $tc = 5$

Using what we learnt from Subtask 1, we will only keep two digits from the original string and add $n - 2$ to our cost.

We do not need to increase or decrease digits we will delete anyways, so we can safely ignore deleted digits.

Now, we only need to decide which two digits to keep. If you're doing subtask 2, you can simply try every option.

However, if you're doing subtask 3, we need to be more clever in how we pick these digits.

Consider what happens if we pick digits a and b. (a comes before b)

We need to turn one of these two to 9, and the other to 0.
Notice that because we are able to reverse for free, it doesn't matter which turns into 9 and which turns into 0.
As such, we'll only consider a becoming 9.

If we turn a into 9, the extra cost incurred would be $(9 - a) \cdot x$.
If we turn b into 0, the extra cost incurred would be $b \cdot y$.

Since we can pick any two digits for a and b, it is optimal to choose the largest a possible, and the smallest b possible.

This is very easily doable in any programming language or even an excel sheet. Do note that if you are doing it in a programming language such as C++, characters may be stored as ASCII values, which means you can use `(s[i] - '0')` to get the numeric value of the i-th digit of s.

Subtask	Answer
1	44991
2	31
3	164187

Task 11: `Fisherman`

Authored by: Brian Lee (penguin133)

Prepared by: Brian Lee (penguin133)

Editorial written by: Brian Lee (penguin133)

Subtask 1

Additional constraints: $tc = 1$, $n = 3$, $l = 7$, $q = 4$, and $\sum t[i] = 49$

In this subtask, all the variables have very small values. We can simulate each of the fish moving manually, and check if the fish are at the desired position at the desired time for each query.

In this case, Ryan catches a total of 2 fish.

Subtask 2

Additional constraints: $tc = 1$, $n = 100$, $l = 500$, $q = 1000$, and $\sum t[i] = 100000$

In this subtask, the values are too large to work out by hand. Instead, we will convert our idea from subtask 1 into code. For each query, we will simulate each of the fish moving for $t[i]$ seconds, and find out whether the fish ends up at position $x[i]$.

Using this method, we will be able to obtain a final answer of 483.

Time complexity: $\mathcal{O}(n \cdot \sum t[i] + q)$

Subtask 3

Additional constraints: $tc = 2$, $n = 100000$, $l = 500000$, $q = 200000$, and $\sum t[i] = 10^{15}$

To solve this subtask, we will have to make a few optimisations to our algorithm.

Firstly, notice that since we only need to know whether a single fish can be caught for each query, we can instead simulate our desired position moving backwards $t[i]$ times, then checking

whether a fish exists at that initial position. Since we are only simulating the movement of 1 item instead of n fish, this removes the factor of n from the time complexity.

Additionally, notice that after the desired position is moved backwards l times, it will end up in exactly the same position it was at l seconds later. This means that we only need to consider the movement of the position for $t[i] \% l$ seconds, where the $\%$ sign here denotes the remainder of $t[i]$ when divided by l. From here on, we assume that $t[i] := t[i] \% l$.

This movement does not have to be simulated each second at a time. We know that if $x[i] > t[i]$ then a fish does not have to teleport to reach position $x[i]$ after $t[i]$ seconds, so we just need to check for a fish at position $x[i] - t[i]$. Otherwise we know that a fish has to teleport once, so its initial position has to be $x[i] - t[i] + l$.

To quickly check for the presence of a fish at any position, we can use a boolean array to keep track of the positions that initially have fish.

Combining these optimisations, we obtain the answer of 159943, and our code will run in $\mathcal{O}(n + q)$.

Subtask	**Answer**
1	2
2	483
3	159943

Task 12: `Pringles Sort`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Sun Beichen (TheRaptor)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $n = 6$

Since n is sufficiently small, you can try operations manually until you solve it.

For a more rigorous solution, see the observations and ideas below.

Subtask 2

Additional constraints: $n = 3000$

Firstly, let's understand what occurs when we use operations on multiple elements.

If we use an operation on X, then on Y, then on Z in the array $\{a, b, X, c, d, Y, e, Z, f\}$, we get the array $\{a, b, c, d, e, f, Z, Y, X\}$.

You will notice that the elements that we chose to use operations on, appear at the back of the array in **reverse** order.

Since we know that the final array must be $\{1, 2, \cdots, n-2, n-1, n\}$, we realise that if we use operations on k elements, they have to be the largest k elements in the array.

Since we know that the elements appear in reverse order of our operations, we just operate on these k elements from smallest to largest.

It is sufficient to try all possible k from 0 to n to solve the problem, as it takes $\mathcal{O}(n)$ time to construct the array resulting from each value of k, and doing this for all $\mathcal{O}(n)$ values of k gives us a final time complexity of $\mathcal{O}(n \times n) = \mathcal{O}(n^2)$.

Subtask 3

Additional constraints: $n = 1\,000\,000$

We further refine our observations by considering what the array looks like after k operations.

We realise that the last k elements are always correctly sorted, since we used operations in the correct order.

Therefore, the full array is sorted if and only if the other $n-k$ unselected elements are in correct order, which would be $\{1, 2, 3, \cdots, n-k-1, n-k\}$.

Since we want to minimise k, we want to maximise $n-k$, which would represent the number of unselected elements.

Hence, we want to find the longest sequence of elements that forms the 'staircase' pattern of $\{1, 2, 3, \cdots, n-k-1, n-k\}$, as these would become our unselected elements.

This is achievable by iterating through the array and keeping a counter of the next value we need in the staircase pattern, and adding 1 to the answer and the counter when the current value in the array matches the counter.

The final value of the counter will be 1 greater than the length of the staircase, so we can calculate the length of the staircase as $counter - 1$.

Then, the number of selected elements is the number of elements outside the staircase, which would then be $n - (counter - 1)$.

Outputting this value will give us the solution in $\mathcal{O}(n)$ time complexity.

Subtask	Answer
1	4
2	2179
3	565989

Task 13: `Bamboo`

Authored by: Lim Chang Jun (lcjly)

Prepared by: Brian Lee (penguin133)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $tc = 3$, $n = 3$, and $k \leq 40$

You can manually work out the minimum number of distinct heights achievable by trying to equate each pair of elements, and seeing if it's possible.

Then, grouping elements into sets of elements that must become equal, you find the minimum operations for each set and sum the operations to solve the problem.

For further elaboration, read below.

Subtask 2

Additional constraints: $tc = 2$, $n = 10000$, and $k = 1$

Firstly, notice that when $k = 1$, you can make any pair of elements a and b equal by adding 1 to the smaller element repeatedly.

As such, we know that the minimum number of distinct heights is exactly one, and all bamboo heights must be equal at the end.

Then, we must decide what the final bamboo height level will be.

Since increasing the final bamboo height level would increase the number of operations, we seek to minimise the final bamboo height level.

As the height of bamboo cannot decrease, the minimum height achievable is always at least the height of the maximum initial bamboo height.

Since we can always have the final height be the maximum initial bamboo height, that is opti-

mal.

The number of operations for a bamboo starting at height x and ending at height y is just $y - x$, so sum this value for all bamboo with $y = max(h)$ to get the final answer.

Subtask 3

Additional constraints: $tc = 5$, $n = 10000$, and $k \leq 10000$

Now that k is no longer equal to 1, we must consider which elements can end up equal.

Given a pair of elements a and b, we want to make them equal by adding k repeatedly to the smaller element.

However, sometimes this will not work, as for the case $a = 2,\ b = 5,\ k = 6$.

Trying some cases, or through some intuition, you may notice that the elements can only become equal if and only if the difference between the a and b is a multiple of k, which is equivalent to saying that a and b have the same remainder when divided by k.

Mathematically speaking (can be ignored if you do not understand math), we understand that by adding k to a, the remainder of $\frac{a}{k}$ does not change, and as such, the value of $a\ (mod\ k)$ and $b\ (mod\ k)$ is constant, and hence must be equal from the beginning.

$a \equiv b\ \pmod{k}$

$(a - b) \equiv 0\ \pmod{k}$

Back to solving the problem, we can then group elements by their remainder when divided by k.

Then, within each group, we apply the solution to Subtask 2 with slight changes, setting y to be the largest initial height **within the group**, and the number of operations for a bamboo starting at height x and ending at height y being $\frac{(y-x)}{k}$.

Since the number of elements within each group **totals** to n, and the time taken to process each group is $\mathcal{O}(m)$ where m represents the number of elements in the group, we get a total time complexity of $\mathcal{O}(n)$. (This time complexity is obtained through amortised analysis, it is good to look this up if you don't know what this is.)

Subtask	Answer
1	5
2	5015390891215
3	1545204087770

Task 14: `E-sign`

Authored by: Brian Lee (penguin133)

Prepared by: Brian Lee (penguin133)

Editorial written by: Brian Lee (penguin133)

Note

We can rewrite the problem as follows:

There are n people. For each person i, find how many times another person (other than person i) signed the document in between the time Ryan sent the document to person i for the first time and the time that person i signed the document.

Subtask 1

Additional constraints: $n = 6$

This subtask can be done by hand. Write down the number of times Ryan sent the document to each person on a piece of paper, and manually update the counts for each event.

Doing this, you will get an array a of $[1, 1, 2, 3, 1, 1]$, and the output value will be 32.

Subtask 2

Additional constraints: $n = 2000$

To answer this subtask, we can directly simulate each event using code. Maintain a list of all the people who Ryan has sent the document to but have yet to sign it, as well as a counter array to keep track of how many times the document was sent to each person.

Each time Ryan sends the document to a new person, add that person to the list. Then, when a person signs the document, increment the counter array for the people in the list by 1. This will ensure that our final counter array will be equal to our desired array a.

Using this array, we can then obtain an output of 1353562740.

This code should run in a time complexity of $\mathcal{O}(n^2)$.

Subtask 3

Additional constraints: $n = 200\,000$

To solve this subtask, we need to speed up our solution for subtask 3.

Solution 1

We can maintain a running counter of how many people have signed the document at each point in time. When Ryan sends the document to a new person x, record the number of people who have signed so far under some array $b[x]$. When person x does finally sign the document, increment the counter, and record this new counter value under another array $c[x]$.

From here, an observation can be made that for person x, the number of times that he was sent the document is exactly equal to $c[x] - b[x]$, since the number of people who signed the document before person x signs the document is $c[x]$ and the number of people who signed the document before person x receives the document for the first time is $b[x]$.

Using this, we can obtain $a[x] = c[x] - b[x]$ for all x and we can obtain our final output value of 1333442114745347.

This solution runs in $\mathcal{O}(n)$.

Solution 2

For a person i, we notice that $a[i]$ is exactly equal to the number of 1 events that occur in between the event "$0\ i$" and the event "$1\ i$". To find $a[i]$, we can perform some precomputation and calculate $a[i]$ using prefix sums.

This solution also runs in $\mathcal{O}(n)$, and gives the same output value of 1333442114745347.

Subtask	Answer
1	32
2	1353562740
3	1333442114745347

Task 15: `Investment`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Kang Yiming (kym2006) and Brian Lee (penguin133)

Editorial written by: Ernest Kiew (Shor the Duck)

Note

Due to the interesting constraints of this problem, I won't go over how to do each subtask individually, but rather give an overview of the problem.

Subtask 1, 2 and 3

Additional constraints for Subtask 1: $n = 3$, $d = 100$, $x = 80$, $y = 65$, $r = 3$, and $t = 200$

Additional constraints for Subtask 2: $n = 50$, $d = 1$, $x = 2\,000\,000$, $y = 0$, $r = 2$, and $t = 2\,000\,000$

Additional constraints for Subtask 3: $n = 30$, $d = 10$, $x = 10^8$, $y = 4$, $r = 2$, and $t = 10^9$

The problem asks you to find the minimum number of thrifty months you need to retire at the end.

The key is to realise that months that you are thrifty for, should come at the start.

The proof: we can consider being thrifty for a month as adding $x - y$ to your current money, then all months have a fixed expenditure of x.

Then, it is clear that being thrifty is equivalent to a boost of current money.

We need to prove that it is optimal to boost our money in the earlier months.

Since $x - y$, the saving for being thrifty, is **constant** for all months, if we boost in month 1, that money will be invested and give us even more money for future months.

However, if we boost in later months, we will not have the opportunity to invest that money as many times, and hence we only lose out in money by shifting boost later on.

Therefore, it is optimal to always be thrifty for the first x months, for some value of x.

Since x is small, between 0 to n, we can try every possible x and find the minimum working x.

The use of spreadsheets can significantly simplify the solving process, leading to a no-code solution.

Subtask	**Answer**
1	-1
2	22
3	28

Task 16: Bamboo 2

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Ernest Kiew (Shor the Duck)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $n = 1, k = 8$

Manually calculate the outcome of the bamboo.

Subtask 2

Additional constraints: $n = 1000$ and $k = 13$

For each bamboo, iterate through all 13 days and find the maximum height reached, resulting in an $\mathcal{O}(nk)$ time complexity solution.

Subtask 3

Additional constraints: $n = 100\,000$

Since k can be up to 10^9, we have to calculate the answer with more brainpower.

We observe that for large k, the heights of a bamboo tend to get stuck in a cycle:

For $a = 3, b = 53, c = 5$, the heights on each day are:
{ 3, 6, 12, 24, 48, 96 (cut to 5) ,
10, 20, 40, 80 (cut to 5) ,
10, 20, 40, 80 (cut to 5) ,
10, 20, 40, 80 (cut to 5) , $\cdots$ }

As you can see from the colouring, there are two parts of the lifecycle of a bamboo.

At first, it starts with $\{a, 2a, 4a, \cdots\}$, before any cut occurs.

Then, once a cut occurs, it continues with $\{c, 2c, 4c, \cdots\}$, resetting with each cut.

We notice that because of how the growth of bamboo is exponential, a cut will occur within the first 32 days, as $a \times 2^{32} > b$ for any value of a and b.

Then, the next cut for all cuts will also occur within 32 days, as $c \times 2^{32} > b$ for any value of c and b.

Now, to find the maximum height reached, we only need to find the maximum height reached in the pink section and in **one** of the green sections, as all green sections are identical.

Since it is guaranteed that within 64 days, the bamboo has gone through the pink section and at least one green section, we can ignore large values of k, setting k to no more than 64.

Hence, we can solve the problem in $\mathcal{O}(64n)$.
(If you wish to be pedantic, it is $\mathcal{O}(n \log(\max(b)))$.)

Subtask	Answer
1	16
2	944954990278
3	94941123772256

Task 17: `Compressor`

Authored by: Chua Wee Chong (sheep)

Prepared by: Lim Chang Jun (lcjly)

Editorial written by: Lim Chang Jun (lcjly)

Subtask 1

Additional constraints: $3 \leq n \leq 5000$

This subtask can be solved by doing a naive simulation as described in the problem statement. The overall time complexity of such a solution is $\mathcal{O}(n^2)$.

Subtask 2

Additional constraints: $3 \leq n \leq 10^6$ and there are only two `'0'`s in s

This subtask is meant to help participants find the full solution for this problem. Let i, j (1-indexed and $i < j$) be the indices of the two 0s.

Case 1 ($j - i \geq 3$):
Notice that after the first compression, the resulting string will consist only of the character 1, resulting in the final remaining character being 1.

Case 2 ($j - i = 2$):
After the first compression, there will be exactly one occurrence of the character 0 at index $i+1$, while the rest of the characters (possibly none) will be 1. If $n = 3$, the character 0 will be the only character after one compression, which means that the final character will be 0. However, if $n > 3$, since there is only one occurrence of character 0, it would not be the most frequent character in any substring of length 3. Therefore, the string after two rounds of compression is guaranteed to consist only of the character 1, hence the final character will be 1.

Case 3 ($j - i = 1$):
Let m be the current length of the string. If $i \neq 1$ and $j \neq m$, then after one compression there will still be two occurrences of the character 0 at indices $i - 1$ and $j - 1$. However, if $i = 1$ or $j = m$, then after one compression, there will only be one occurrence of the character 0, which will disappear after another round of compression. The zeros will disappear after

$\min(j, m-i+1)$ rounds of compression. Since each compression reduces the string length by 2, there can only be $\frac{n-1}{2}$ rounds of compression in total. Therefore, $\min(j,\ m-i+1) > \frac{n-1}{2}$ implies that the final remaining character will be 0.

Let $mid = (n+1)/2$. It follows that the final remaining character is 0 if and only if $i = mid$ or $j = mid$.

Subtask 3

Additional constraints: $3 \leq n \leq 10^6$

If the character at index mid is the same as at least one of its adjacent characters (i.e. $mid-1$ or $mid+1$), then there exists a contiguous block of length at least 2 consisting of either only 1s or only 0s. Any substring of length 3 intersecting this block will have this character as the majority, so it will never be removed in future compressions. In this case, the final remaining character is already determined and the process is finished.

Otherwise, we must have $s[mid-1] \neq s[mid]$ and $s[mid+1] \neq s[mid]$. This means that $s[mid-1] = s[mid+1] \neq s[mid]$, meaning that the string is alternating around the middle. Now consider the next compression. The new middle position, denoted by mid', is produced from the substring of length 3. This substring is either `010` or `101`, so the majority character is not equal to the middle one, and hence the new middle character satisfies $s[mid'] = s[mid-1]$.

We now repeat the same check at mid'. If the character at mid' matches one of its adjacent characters, then we are done as before. Otherwise, both adjacent characters of mid' must be different from $s[mid']$, which again forces them to be equal to each other. Tracing this back to the original string, this implies that $s[mid-2] \neq s[mid-1]$ and $s[mid+2] \neq s[mid+1]$, so the alternating pattern must extend one step further outward.

By continuing this argument inductively, we see that as the center substring's characters alternate between 0s and 1s, the middle character would alternate as well. Eventually, either the alternation breaks at some position, or the alternation extends to the entire string. The final remaining character would be the first character of the longest possible alternating string centered around the middle. This algorithm runs with a time complexity of $\mathcal{O}(n)$.

Subtask	Answer
1	10000
2	11011
3	10010

Level 3

Task 18: `Jokers`

Authored by: Ernest Kiew (Shor the Duck)

Prepared by: Ernest Kiew (Shor the Duck)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $m = 1, b[1] = 1$

Given that there is only one `mult` joker, we can just try both having it, and not having it.

Then, in either case, it is optimal to take the highest untaken `chips` joker until we fulfil the total score requirement.

Time complexity: $\mathcal{O}(n \log n)$, with the bottleneck being sorting for highest chips jokers.

Subtask 2

Additional constraints: $n, m \leq 1\,000$, $\sum n \leq 10\,000$ and $\sum m \leq 10\,000$

Similarly to the idea that we should only take the highest untaken `chips` joker, we can do the same for `mult` jokers.

To clarify, if we sort the `chips` and `mult` jokers by value in **descending** order, the highest score achievable with some number of jokers can be achieved with some prefix of `chips` jokers and some prefix of `mult` jokers. (Both prefixes possibly being empty)

We can naively try every prefix of `chips` and `mult` jokers, as long as we maintain the total sum of the prefix while iterating through each prefix, resulting in an $\mathcal{O}(n^2)$ time complexity.

Subtask 3

Additional constraints: $n, m \leq 1\,000\,000$, $\sum n \leq 10\,000\,000$ and $\sum m \leq 10\,000\,000$

To speed up the search, we can use the technique of Binary Search the Answer (informally known as BSTA).

We can prove that if some number of jokers x is sufficient to achieve the score requirement, then all numbers of jokers greater than x can achieve the score requirement as well, as adding one joker can only increase the score.

Similarly, we can prove that if some number of jokers x is insufficient to achieve the score requirement, then all numbers of jokers lesser than x cannot achieve the score requirement as well, as removing one joker can only decrease the score.

This proves *monotonicity* of our answer, as there is a point at which all answers at or after that point is valid, and all answers before that point are invalid.

By maintaining the range of possible but unchecked answers, we can "guess" an answer in the middle, and by determining if that middle answer is valid or not, we can choose to only look at smaller answers or larger answers, and hence cutting the range of possible but unchecked answers in half.

Since our answer lies in the range of [0, $n + m$], we know that the number of answer checks we will need will not exceed $\mathcal{O}(\log(n + m))$, as that is the number of times we can cut the range in half before it reaches a size of 0.

Now that we know Binary Searching the Answer is valid, we can work on the checking problem:

Given an answer g, determine if g jokers is sufficient to reach the score requirement.

This is a simpler problem than the original one, and by iterating through the number of `chip` jokers from 0 to g, you can also determine the number of `mult` jokers as $g - chips$, where $chips$ is the number of chip jokers.

By maintaining the sum of the prefixes, this can be done in $\mathcal{O}(n + m)$ time.

Since the time complexity of Binary Search the Answer is $\mathcal{O}(\log(numAns) \times timeCheck)$ where $numAns$ is the number of possible answers and $timeCheck$ is the time complexity of the checking function, we conclude with a time complexity of $\mathcal{O}((n + m)\log(n + m))$, which is fast enough to pass.

Subtask	Answer
1	8111599
2	70419
3	76106593

Task 19: `Musical Chairs`

Authored by: Chua Wee Chong (sheep)

Prepared by: Ryan Goh (rgca) and Chua Wee Chong (sheep)

Editorial written by: misalignedDiv and Ernest Kiew (Shor the Duck) (editing)

Note

The input is already sorted in increasing $c[i]$, and for brevity, Position will be abbreviated as Pos.

Furthermore, we will treat the input as part of $\{d[i], c[i]\}$ pairs.

Subtask 1

Additional constraints: $d[i] = \texttt{L}$ for all $1 \leq i \leq n$ We can greedily shift the people to the leftmost position.

For example, for the input:

n = 51, m = 6
$\{L,\ 9\},\ \{L,\ 11\},\ \{L,\ 20\},\ \{L,\ 23\},\ \{L,\ 34\},\ \{L,\ 51\}$

Shift the person at Pos 9 to Pos 1, the person at Pos 11 to Pos 2, $\cdots$, the person at Pos 51 to Pos 6.

Simply summing the difference in starting position and ending position for each person will give you the answer.

Although this is not relevant for now, if the input is the opposite, where all $d[i] = \texttt{R}$, we should shift them all to the rightmost position.

Subtask 2

Additional constraints: All persons with direction `R` are initially to the left of all persons with direction `L`

The left directions will go against the right directions. Lending inspiration from what we learnt from Subtask 1, we should shift the left people to form a consecutive block of people starting at index j, and all right chairs to form a consecutive block of people ending on index $j-1$ to ensure no more moves can be made.

To clarify:

```
People:          R   R   R   L   L   L
Indices: ...   j-3 j-2 j-1  j  j+1 j+2   ...
```

j should be $\geq$ the pos of the rightmost right chair, and $\leq$ the pos of the leftmost left chair.

We could brute force all possible j to get the optimal j, but since there are $\mathcal{O}(n)$ possible values of j and calculating each one takes $\mathcal{O}(m)$ time, this will take $\mathcal{O}(nm)$ time, which would take too long. How do we speed this up?

Let's say there are a `R`s and b `L`s.

By shifting j rightwards by one index, we get:

```
People:          R   R   R   L   L   L
Indices: ...   j-2 j-1  j  j+1 j+2 j+3   ...
```

We have moved the `L` people one step closer to their original pos and the `R` chairs one step farther, this means that our result will decrease by a (The a `L` people move less) and increase by b (The b `R` people move more), resulting in a net change of $b-a$.

If $a > b$ this would result in less shifts.
If $a < b$ this would result in more shifts.
If $a = b$, there will be no change in shifts.

We realise that by shifting j 1 step rightwards, it will always change only based on the value of $b-a$.

Hence, to ensure the max possible result if $b > a$, the optimal j would be as far right as possible.

if $b \leq a$, the optimal j would be as far left as possible.

The furthest right position of j is the furthest left `L`'s position, and the furthest left position of j is the furthest right `R`'s position plus one.

With this we can solve this problem in $\mathcal{O}(n)$ time.

Subtask 3

Using what we learnt from Subtask 1, we can ignore the first `R`s until the first `L` appears, as we can shift them all to the start without interacting with other people.

Similarly, we can ignore the last `L`s until a `R` appears, as we can shift them all to the end without interacting with other people.

This would clean up the data, and would result in alternating sections of `R`s and `L`s like this (ignoring spaces between people):
`RRRRLLLLRRRRLLLLL ··· RRRLLL`

Let's call a set of people forming this pattern $R \cdots RRRLLL \cdots L$ as a block.

We can show that each block does not interact with any of the other blocks. As such, we can solve for each block independently.

Using what we've learnt from Subtask 2 we can calculate this distance moved for each block to get the answer in $\mathcal{O}(n)$.

Subtask	Answer
1	360358556037
2	199541624813
3	7401359

Task 20: `Volcano`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Zhao Yaoqi (zyq1810)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $tc = 3$, $1 \leq l \leq 20$, and $1 \leq n \leq 6$

This subtask is meant for contestants to understand the problem by working it out manually.

Drawing out the 6 by 6 grid, we realise that the i-th volcano creates a square piece of land of width and height equal to $2 \times h[i] + 1$, centred on the volcano's location. (The square might be cutoff by the boundaries of the grid.)

By drawing out the squares created, we can visually tell how many islands are created.

Subtask 2

Additional constraints: $tc = 10$, $1 \leq l \leq 1000$, and $1 \leq n \leq 30$

Now that the size of the grid is larger, we will need computer assistance.

For each square on the l by l grid, we can iterate through each volcano to see if that volcano would cover this square (we check ifthe horizontal and vertical distance both do not exceed $h[i] + 1$).

This part of the code takes $\mathcal{O}(tc \times l^2 \times n)$ time, which is sufficient for running in less than 30 seconds. (We can speed this up through usage of Breadth First Search (BFS) or even further sped up through Multisource BFS.)

Then, we need to count the number of islands. We do this by using BFS as well. For every unvisited but covered square, we perform a BFS that visits all covered squares that are connected to it, and add one to our answer.

While initially it may seem that this algorithm takes $\mathcal{O}(l^2 \times l^2)$ time, since our BFS actually

runs in linear time with respect to the number of nodes visited, and we only visit l^2 nodes in total, we have an amortised time complexity of $\mathcal{O}(l^2)$ for this part of the code.

Combining these two parts allows us to solve this subtask fully.

Subtask 3

Additional constraints: $tc = 10$, $1 \leq l \leq 10^9$, and $1 \leq n \leq 5000$

Now, we have to speed up our algorithm significantly.

An important clue is that l is very large, and suggests we cannot have our time complexity be linear or worse with respect to l.

Furthermore, $n \leq 5000$ typically suggests a quadratic solution.

As such, we will attempt to find a solution whose runtime is independent of l, but quadratic in n.

We notice that the pieces of land two volcanoes i and j make, will connect if and only if the difference in both x-coordinates and y-coordinates is no more than $(h[i] + h[j] + 1)$.

This idea allows us to sidestep the large size of the grid, as we no longer need to work on the grid to determine the number of islands. We can make an equivalent definition of island as a maximal group of volcanoes such that each volcano is connected, directly or indirectly, to every other volcano in the group.

We can then perform the same algorithm of repeated BFS on unvisited nodes, on a graph where each vertex is a volcano, and an edge exists between volcanoes i and j if the land they create connects.

In this case, our BFS runs in $\mathcal{O}(n^2)$ time, as there are $\mathcal{O}(n^2)$ edges in the worst case.

Subtask	Answer
1	6
2	73
3	7146

Task 21: `Card Draw`

Authored by: Ernest Kiew (Shor the Duck) & Brian Lee (penguin133)

Prepared by: Zhao Yaoqi (zyq1810)

Editorial written by: Ernest Kiew (Shor the Duck)

Subtask 1

Additional constraints: $tc = 10$, $p = 1$, and $2 \leq n \leq 2 \times 10^5$

Firstly, we must observe that it is always optimal for Shor to play the highest card in his hand.

This can be quickly proven by Greedy Stays Ahead (but not necessary to prove in-contest), as playing the highest card means you have a larger prefix of the draw pile in your hand, which gives you more options of which cards to play while also contributing to the goal of holding every card.

Any strategy that involves not playing the highest card, can have the lower card played be replaced by the highest card for an equal or better result.

Simply maintaining the highest card currently in your hand, and simulating playing that card is sufficient to pass, as there are only $\mathcal{O}(n)$ cards in total we can draw, so our time complexity is an amortised $\mathcal{O}(n)$.

Subtask 2

Additional constraints: $tc = 10$, $p = 2$, and $2 \leq n \leq 1000$

It is actually optimal to play the highest card here too, despite having to return the card to the draw pile.

There is no reason to save the highest card for a later turn, as that means you draw less cards now, and it will take longer for the highest card to return to your hand.

There is also the possibility that we never play the highest card, but by replacing a lower card for the highest card, we can prove that this situation's outcome is always equal or worse than playing the highest card.

Hence, since saving the highest card for later and never playing the highest card are both equal or worse to playing the highest card, we should always play the highest card.

However, our time complexity proof for Subtask 1 breaks down.

Consider what happens if our draw pile is $\{\underbrace{1, \cdots, 1}_{n-1 \text{ times}}, 2\}$

Our hand size starts at 1, but because we return our played card into the draw pile, we play a 1 and return a 1, resulting in our hand size staying at 1.

This repeats $n-2$ times, until we finally draw a 2. We can then play the 2, but this results in our hand consisting of two 1s.

We then have to play 1s $n-2$ times before we see a 2 again.

This cycle of playing cards results in $\mathcal{O}(n^2)$ cards played.

Luckily, Subtask 2's constraints are small and will allow a simple simulation solution to pass.

We simulate by maintaining a priority queue consisting of all cards in our hand, as we need to add elements into the priority queue, while removing the largest element.

This gives us a time complexity of $\mathcal{O}(n^2\, log(n))$, which passes.

Subtask 3

Additional constraints: $tc = 10$, $p = 2$, and $2 \leq n \leq 2 \times 10^5$

Notice that our hand size never decreases, since the lowest card is a 1, which when played, keeps our hand size constant.

I will refer to 1 cards as small and non-one cards as big.

Every time we play a big card, our hand size increases by 1.

Since our hand size is no more than n and can never decrease, we know that our hand size only increases at most $n-1$ times, and hence we can only play $n-1$ big cards.

If we can somehow quickly deal with the small cards, we can actually have a reasonable time complexity.

This gives rise to the idea of compression. Instead of dealing with small cards one-by-one, we merge adjacent small cards into a group of small cards.

Since we process at most n big cards, and there cannot be two consecutive groups of small cards (it violates our definition of a group), the maximum number of small card groups we need to process is where, between every two processed big cards, there is a group of small cards, which means at most n groups of small cards are processed.

Hence, we can perform simulation after this group compression is done, and since we only process n big cards and n small card groups, if we process each item in $\mathcal{O}(\log n)$ time (priority queue is still necessary), we get a final time complexity of $\mathcal{O}(n \log n)$.

Subtask	**Answer**
1	323413
2	160304
3	5517262337

Task 22: `Cup Stacking`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Zhao Yaoqi (zyq1810)

Editorial written by: Sun Beichen (TheRaptor)

Note

A common mistake when attempting this problem is forgetting that a cup cannot be slotted onto a lower cup if there are 2 cups being supported by the lower cup.

Subtask 1

Additional constraints: $tc = 3$, $1 \leq a[i] \leq 5$ for all $1 \leq i \leq n$, and $1 \leq n \leq 5$

This subtask is meant for manual trial and error to find the best way to collapse the cups. The experimentation can in fact provide valuable insights to the full solution.

Subtask 2

Additional constraints: $tc = 10$, the array a contains one '1' and $n - 1$ '0's, and $1 \leq n \leq 10^6$

In this subtask, the goal is to maximise the final number of cups at one specific index. Let this index be x. Notice that on the second lowest layer, although 2 cups use the cup on index x as support, at most 1 cup can be slotted onto index x (it is impossible to move both cups to index x due to the restrictions on slotting cups). This applies to every layer, so the maximum number of cups that can be at index x in the end is n. The existence of such a construction is provable through induction, but the proof is omitted here. Therefore, the answer is to simply output n.

Subtask 3

Additional constraints: $tc = 10$, $1 \leq a[i] \leq 10^6$ for all $1 \leq i \leq n$, and $1 \leq n \leq 10^6$

Participants who attempted subtask 1 may notice an interesting result – the set cup numbers in the end is a set of all numbers from 1 to n. In fact, the final arrangement will always be a permutation of numbers from 1 to n, and all permutations can be achieved. This is provable by induction. As a result, the answer can be found by sorting array a and greedily assigning larger cup numbers to larger array values.

Inductive proof: We will first prove that for n layers, all final cup arrangements will form a permutation of numbers from 1 to n.

Base case is true because for a pyramid of 1 layer, the final arrangement is $[1]$.

Assume that it is true for a pyramid of n layers. Consider a pyramid of $n + 1$ layers. We will first collapse the top n layers, generating a permutation of 1 to n, before collapsing them onto the final layer. We can see that exactly 1 cup in the $(n + 1)^{th}$ layer that will not have anything slotted onto it, while all other cups will have exactly 1 cup stack slotted onto it. This guarantees the final arrangement being a permutation of $[1, 2, ..., n + 1]$. Thus, by induction, the statement is proven.

We will now show that all permutations are achievable.

Again, the base case is 1 layer, which can achieve all permutations of $[1]$.

Assume that this is true for a pyramid of n layers. Say we have a target permutation of $[1, 2, ..., n + 1]$ that we want to achieve with $n + 1$ layers. We can construct this by locating the index of the element with value 1, removing it from the array, then reducing all other elements by 1 to generate a permutation of $[1, 2, ..., n]$. By achieving this permutation using the top n layers, we can collapse them onto the $(n + 1)^{th}$ layer (while avoiding the index that is supposed to have value 1) to achieve the target permutation. Since any target permutation can be constructed through this method, the statement is proven.

Subtask	**Answer**
1	121
2	7552421
3	3157623464083820735

Task 23: `Mildly Angry Ducks`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Chua Wee Chong (sheep)

Editorial written by: misalignedDiv and Ernest Kiew (Shor the Duck) (editing)

Note: Disintegrate and explode are used interchangeably in this editorial :p

Subtask 1

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

Implementation wise, we can use 2 arrays, a representing the row of ducks on the current day and b representing the row of ducks on the next day. By scanning through a, we can append the survived ducks to b, and swap the 2 arrays with each other after scanning. The swap operation will only take $\mathcal{O}(1)$ time as we are just swapping their memory addresses.

With this process, there are at maximum $\mathcal{O}(n)$ days and $\mathcal{O}(n)$ scans per day, achieving an $\mathcal{O}(n^2)$ time complexity, with the worst case scenario being:

$[1, 1, 1, 1, 1, \cdots, 2, 2, 2, 2]$

This is because we have to go through the entire list which takes $\mathcal{O}(n)$ per removal. As there are n removals, this results in an $\mathcal{O}(n^2)$ time complexity.

Subtask 2

Additional constraints: $1 \leq n \leq 2 \times 10^6$ and if $d[i] = d[j]$ and $i < j$, then all $d[i], d[i+1], \cdots, d[j]$ are equal to $d[i]$

The condition of "If $d[i] = d[j]$ and $i < j$, then all $d[i], d[i+1], \cdots, d[j]$ are equal to $d[i]$" is saying that all elements that are equal must be consecutive.

So instead of listing each duck one by one, we can group consecutive ducks with the same ID into ranges.

For example, $[1, 2, 2, 2, 2, 3, 1, 3, 3]$ would become $[\{1, 1\}, \{2, 4\}, \{3, 1\}, \{1, 1\}, \{3, 2\}]$,

where the 0th number of each curly brace represents the ID of the duck and the 1st number represents the number of ducks in the range. (Note that this example does not follow the subtask constraint.)

We realise that the range of ducks at the beginning and end decrease in length by 1 each day, and the rest decrease by 2.
Since all equal elements are consecutive, we know that these ranges of ducks will never merge or interact with each other in any other way.
We can use this information to deduce the 1st and the 2nd longest lasting range of ducks (i.e, largest number of days it lasts), and then calculate the number of ducks left like this.

Subtask 3

Let's look at our representation of grouping consecutive ducks with the same ID into ranges.

At first glance, this might seem like another $\mathcal{O}(n^2)$ algorithm in time complexity, but it's actually an $\mathcal{O}(n)$ algorithm.

Let's define some variables:
Let n be the number of ducks.
Let k be the length of the range list. (the grouping in Subtask 2)

1. Let's assume the length of the range list is constant for now.
2. Going through the whole list takes $\mathcal{O}(k)$ time.
3. We note that the number of days is not $\mathcal{O}(n)$, it's actually $\mathcal{O}(\frac{n}{k})$, since ducks explode at the boundaries of the range, at least k ducks will explode everyday, resulting in $\mathcal{O}(\frac{n}{k})$ days
4. Hence, the whole algorithm achieves an $\mathcal{O}(k \times \frac{n}{k}) = \mathcal{O}(n)$ time complexity

From here, we can see that for each duck explosion, it only affects at most 2 ranges, so we can handle each explosion in $\mathcal{O}(1)$ time. As there are at most n explosions, this algorithm will therefore achieve an $\mathcal{O}(n)$ time complexity.

Do note the important case where $[\cdots, \{3,6\}, \{1,1\}, \{3,7\}, \cdots]$ will convert to $[..., \{3,9\}, ...]$ the next day and **not** $[..., \{3,4\}, \{3,5\}, ...]$.
Remember to merge ranges of the same ID together!

Subtask	Answer
1	4954
2	25590
3	2257469

Task 24: `Tick Tock`

Authored by: Lim Chang Jun (lcjly)

Prepared by: Ryan Goh (rgca)

Editorial written by: Ryan Goh (rgca)

Subtask 1

Additional constraints: $n = 3, m \leq 10$

The subtask exists for one to try on paper. It's tedious but it's free points. It's also a subtask that can catch most errors in the latter subtasks.

Subtask 2

Additional constraints: $n, m \leq 500$

We formulate a simple DP. Let $dp(idx, hour, minute)$ be the minimum number of turns required to kill monsters idx to m, when starting the hour hand at $hour$ and the minute hand at $minute$. We ignore the parameter $hour$ if $minute$ is not 0.

We have 2 simple transitions: either go to $dp(idx, new\_hour, minute+1)$ or $dp(idx, new\_hour, minute-1)$, both with cost of 1. In the event that the minute or hour hand exactly falls onto the idx-th monster, we then transition to $dp(idx + 1, new\_hour, new\_minute)$ with cost 0.

Notice this does not form a directed acyclic graph as one can move the minute hand clockwise and then anticlockwise again forming an infinite loop. Hence, we do BFS instead of DP. However, since weights can be 1 or 0, we can use either Dijkstra or modified BFS known as 0-1 bfs, using a priority_queue or deque respectively. We start our BFS / Dijkstra from $idx = 0$ (assuming zero-indexed), $hour = n$ and $minute = n$.

There might be other ways to solve this subtask with better constants or time complexity, we only explained the most intuitive one.

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

Subtask 3

Additional constraints: $n \leq 5000$, $m \leq 500$

We can cut down the number of states in the DP. Let $dp(idx, hour, last\_killed)$ be the minimum number of turns required to kill monsters idx to m, when starting the hour hand at $hour$. We also keep track on whether monster $idx - 1$ is killed by the minute hand or the hour hand using a boolean.

Interestingly, with just this information, we can figure out where the hour and minute hand is right now. If the monster is killed by the hour hand, the minute hand must be at zero; if not the minute hand is at the position of the last monster (or at n if there is no last monster).

For our transitions, we have 4 cases: move the minute hand clockwise or anticlockwise; or move the hour hand clockwise or anticlockwise to kill the idx-th monster. Use some careful math and case work (for example, whether any of the hands move past number n on the clock) for all the cases. We then call our dp with $idx = 0$ (assuming one-indexed) and $hour = n$.

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

Subtask	Answer
1	82
2	3457519
3	32873314

Task 25: `Mahjong`

Authored by: Brian Lee (penguin133)

Prepared by: Brian Lee (penguin133)

Editorial written by: Brian Lee (penguin133)

Note

In the following solution, we will denote a *winning tile* as a tile which when drawn, allows Brian to win, and a *winning hand* as a hand in which Brian can win.

$\sum n$ denotes the sum of n over all test cases in a subtask.

Subtask 1

Additional constraints: $n \leq 6000$ and $\sum n = 30000$

In this subtask, n is small enough for solutions of time complexity resembling $\mathcal{O}(n^2)$ or $\mathcal{O}(n^2 \log n)$ to pass. To solve this subtask, we can check if every tile is a winning tile by adding it to the hand, and then checking if the current hand is a winning hand.

To check if a tile is a winning tile, notice that the only way to use tiles of type 1 is to form consecutive runs of tiles of types $1, 2, 3, \ldots, k-1, k$. This means that the number of tiles of types $2, 3, 4, \ldots, k-1, k$ must be greater than or equal to the number of tiles of type 1. We can then form the runs as required, and subtract the number of tiles used for each type from $2, 3, \ldots, k-1, k$.

After which, this is equivalent to solving the problem with only tiles ranging from 2 to n. We can repeat this process until all tiles are used, or when the condition above is violated.

This idea can be implemented with a segment tree supporting range minimum query and range subtract operations, which would have a time complexity of $\mathcal{O}(n^2 \log n)$, or the simulation could also be done with a deque / queue or prefix sums of the sort, having a time complexity of $\mathcal{O}(n^2)$. Other viable solutions include coding the basic $\mathcal{O}(n^2 k)$ brute force and running it on your computer for a sufficient amount of time.

Subtask 2

Additional constraints: $\sum n \leq 2 \times 10^6$ and $k \leq 10$

We will start by deriving a few observations of the winning hand.

Observation 1: In a winning hand, there is exactly one way to group the tiles into consecutive runs of size k.

This can be seen from the observation that we made in subtask 1 when we check for a winning hand. If we consider tile types in increasing order, at each point there is exactly 1 way for the tiles of the smallest type to be grouped, and we can continuously repeat this until all tiles are used or it is impossible to group tiles.

This allows us to make the second observation:

Observation 2: Drawing a tile of type x will not affect the groupings of tiles of type y where $1 \leq y \leq x - k$ or $x + k \leq y \leq n$.

We will show each side of the inequality separately. Consider tiles of type y where $1 \leq y \leq x-k$. Then suppose we perform our checking algorithm upwards from tile 1. Since the number of groups starting at each tile i is fixed for all i for a winning hand, increasing the number of tiles for any type that is $\geq y + k$ will definitely not affect the groupings for tiles of type y, since all such groupings involve no tiles of type $\geq y + k$.

The second inequality can also be shown, by performing the checking algorithm in decreasing tile types from tiles of type n to 1. From here, it is not hard to derive that tiles of type $\leq y - k$ will not affect the groupings of type y.

Using the above 2 observations, we notice that drawing tile x will only affect the groupings of tile types y where $x-k+1 \leq y \leq x+k-1$. Hence, we can presimulate the checking algorithm from both ends, and for each index i, store the number of remaining tiles for each type j where $i-k+1 \leq j < i$ and $i < j \leq i+k-1$. Then, we can try drawing a tile of type i and simulate the checking algorithm with the relevant $2k-1$ indices, which would work in $\mathcal{O}(k)$ time.

This algorithm works in $\mathcal{O}(nk)$ and is sufficient to solve the second subtask.

Subtask 3

Additional constraints: $\sum n \leq 2 \times 10^6$.

In this subtask, we need to further speed up our algorithm by making more observations.

Observation 3: Let a winning tile be x. Then all winning tiles y have the form $y = x + ak$, where a is an integer.

To show this, for each $0 \leq i < k$, consider the number of tiles that have a type y such that $y \equiv i \pmod{k}$. In each group of k tiles, all k tile types have a distinct remainder when divided by k, and hence there must be an equal number of tiles that have a type y such that $y \equiv i \pmod{k}$ for all $0 \leq i < k$. Hence it can be seen that all winning tiles must have the same remainder when divided by k, and in other words, all winning tiles y are of the form $pk + q$, where p, q are non-negative integers and q is constant.

From here, we can make the following observation:

Observation 4: The values of a which produce a winning tile form a contiguous range.

The idea for this is to consider how the drawn tile will contribute to the group. There are 2 main ways – either by merging 2 disjoint runs of tiles to form a contiguous run of length k, or by extending a run of length $k - 1$ in either direction.

If tile x merges 2 disjoint runs, then tile x will be the only winning tile since any other tile would still result in there being 2 disjoint runs of length $< k$.

If tile x extends a run of length $k - 1$, let the original length of the run be l, where $l = pk - 1$ (since the run should be one off from being able to group all the tiles together). Then there are $(l + 1)/k = p$ values of x which win, and notably, there can only one of such runs since having more than one would mean that drawing one tile is not enough to win the game.

Hence, given the above observation, we can formulate an algorithm as follows:

1. Find a winning tile x.
2. Find the lower and upper bounds of a such that $x + ak$ is a winning tile.
3. Output the number of valid a.

The first step can be done through the simulation by finding the first tile type that has insufficient to cover the tiles before it.

The second step can be done through binary search and simulation for checking.

This algorithm works in $\mathcal{O}(n \log n)$ and is sufficient to pass this subtask.

Alternative solutions include noticing that only $\mathcal{O}(\frac{n}{k})$ tiles need to be checked for winning tiles, and checking for such tiles in $\mathcal{O}(k)$ time, either by using prefix sums or a segment tree to maintain the values of the k tile types before and after. The checking algorithm is similar to the one in subtask 2.

This algorithm would run in $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$.

Subtask	Answer
1	115
2	174019
3	9112

Task 26: `Slimes`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Chua Wee Chong (sheep)

Editorial written by: Chua Wee Chong (sheep)

Subtask 3

Additional constraints: None

We will consider the full solution first.

Define an interval $[l, r]$ to be good if and only if $a[l]+a[l+1]+\cdots+a[r] \geq max\{a[1], a[2], \ldots, a[n]\}$. Define the length of an interval $[l, r]$ to be $r - l + 1$, written as $\text{len}([l, r]) = r - l + 1$.

We can restate the problem: For all slimes, find the length of the smallest good interval that contains it.

Consider the following algorithm:

1. For all i, find the minimum $r[i]$ such that $[i, r[i]]$ is a good interval. Call this set of intervals X. If $r[i]$ does not exist, we insert $[i, \infty]$ instead.
2. For all i, find the maximum $l[i]$ such that $[l[i], i]$ is a good interval. Call this set of intervals Y. If $l[i]$ does not exist, we insert $[\infty, i]$ instead.
3. For all i, find the smallest interval in $X \cup Y$ such that i is contained within that interval.

We will prove that the algorithm is correct.

Suppose that there exists some i such that $[A, B]$ forms a good interval and $1 \leq A \leq i \leq B \leq n$. Further, suppose that $\text{len}([A, B])$ is strictly smaller than all other intervals in $X \cup Y$ that contain i. Also, suppose $[A, B]$ is not an interval in $X \cup Y$.

From X, we have the interval $[A, r[A]]$. If $r[A] \geq i$, then $[A, r[A]]$ would contain i. Due to the assumption that $\text{len}([A, B]) < \text{len}([A, r[A]])$, this would imply that $B < r[A]$. Since $[A, B]$ is a good interval, this violate the minimality of $r[A]$.

Hence, it must be that $r[A] < i$. Now, consider the interval $[A, i]$. It must be a good interval, as $A < r[A] < i$, that is, $[A, i]$ contains $[A, r[A]]$ as a sub-interval. An upper bound on the length of the smallest interval containing i would be len($[A, i]$).

Since we assume that len($[A, B]$) is strictly smaller than all the intervals in $X \cup Y$ that contain i, it must be $B < i$. However, we must also have that $[A, B]$ contains i, that is, $B \geq i$. We arrive at a contradiction.

Hence, for all i, there does not exist any interval not within $X \cup Y$ that has a shorter length than all other intervals that contain i in $X \cup Y$.

The above proof shows that considering those $2n$ intervals is sufficient to find the answer for all i. To construct the intervals, we can perform a sliding window twice: once forwards and another backwards. The details are left to the reader as an exercise. This can be done in $\mathcal{O}(n)$.

After finding the required intervals, tag each of them with its own length. To find the minimum length interval for each i, either a sweep line + set method or a segment tree can be used. Again, the details are left as an exercise to the reader. This can be done in $\mathcal{O}(n \log n)$.

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

Subtask 1

Additional constraints: $n \leq 500$

This subtask is for less optimised solutions.

Some possible less optimised solutions are:

- Computing all $\mathcal{O}(n^2)$ possible intervals.
- Without using a sliding window, computing intervals in $\mathcal{O}(n^2)$ time total.
- Less optimised methods of finding the minimum length interval for each i, $\mathcal{O}(n)$ time each, $\mathcal{O}(n^2)$ time total.

Subtask 2

Additional constraints: $n \leq 10000$

Refer to Subtask 1.

Subtask	Answer
1	48389
2	21985998
3	222900402216

Level 4

Task 27: `Bad Addition`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Chua Wee Chong (sheep)

Editorial written by: Chua Wee Chong (sheep)

Notes

Define $\overline{ab}$ to be string concatenation of a and b.
Define $len(a)$ to be the number of digits in a.
Define $\star$ to be the bad addition operator. That is, $n \star m$ means to do bad addition on n to m.
A formal definition of bad addition is: Given integers a, b of n digits written as $\overline{a_1 a_2 \dots a_n}$ and $\overline{b_1 b_2 \dots b_n}$, $a \star b = \overline{(a_1 + b_1)(a_2 + b_2) \dots (a_n + b_n)}$

If either of them is shorter than length n, we can left-pad it with zeros.

Subtask 1

Additional constraints: $k = 10$

Since k is small, one can work this out by simulating the process either by hand or by using Python which supports arbitrarily large integers.

However, this solution might not work as well for larger k as the number of digits the integer becomes a significant bottleneck.

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

Subtask 2

Additional constraints: $k = 272727$

Because $1 \leq n \leq 99999$, n is at most 5 digits. Let $\overline{ab}$ be some integer with $len(\overline{ab}) = x + 5$ digits, such that $len(b) = 5$ and $len(a) = x$. If $\overline{ab}$ is too short, we can left-pad it with zeroes.

Observation: If $n \star b = \overline{a'b'}$, where a' is y digits long and b' is 5 digits long, then $n \star \overline{ab} = \overline{aa'b'}$.

This observation comes about because n itself is at most 5 digits, so further digits will not be affected by bad addition. Hence, we can write numbers in the form $\overline{ab}$ and use the observation. Due to modulo properties, we can store a mod $10^9 + 7$ instead of a. This is also because a is unaffected by bad addition.

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

Subtask 3

Additional constraints: $k = 407600000000000000$

Now, k is very large and it is infeasible to simulate bad addition.

Define $f_x(b) = ((b \star n) \star n) \cdots \star n)$ such that $\star$ appears x times.

Observation: It can be shown by induction that $f_x(\overline{ab}) = \overline{aa_xb_x}$ for an arbitrary integer x.

Further, if $f_x(b_0) = \overline{a_xb_x}$ and $f_y(b_x) = \overline{a_{x+y}b_{x+y}}$, then $f_{x+y}(\overline{a_0b_0}) = \overline{a_0a_xa_{x+y}b_{x+y}}$. In other words, we can chain bad addition operations together easily.

To get to our desired k, we first pre-compute operations by powers of two. That is, for all $0 \leq b \leq 99999$, we first find $f_1(b)$. Then, using our observation above, we can calculate $f_2(b)$ by setting $x = 1, y = 1$. Similarly for $f_4(b), f_8(b) ... f_{2^{59}}(b)$. This step takes $\mathcal{O}(n \log k)$ time.

Afterwards, we can decompose k into binary digits and apply the same trick. For example, if $k = 6 = 4 + 2$, we would do $f_{4+2}(0) = f_4(f_2(0))$. In this way, we will only take $\mathcal{O}(n \log k)$ time.

As an implementation detail, because the values get large quickly, we have to store the results of f as $(a, len(a), b)$. In this way, we can handle both the string concatenation and modulo operations properly. Without $len(a)$, simulating string concatenation would not be possible.

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

Subtask	Answer
1	134927836
2	430188270
3	664305046

Task 28: `Company Management`

Authored by: Sun Beichen (TheRaptor)

Prepared by: Sun Beichen (TheRaptor)

Editorial written by: Sun Beichen (TheRaptor)

Subtask 1

Additional constraints: $p[i]= 1$ for all $1 \leq i \leq n$

The constraints form a star graph.

If the root has x hours, then all leaves should have $m - x$ hours. While the answer can be computed by trying all values of x, an interesting observation is that an optimal distribution of hours can be attained by either assign all m hours to the root, or to all leaves. This is because if both the root and the leaves have hours, then we can always reduce hours for the side with lower total value and increase hours for the others.

Subtask 2

Additional constraints: $m = 1$

Nodes with an hour assigned to them must not share ancestor-descendant relationships. The maximum sum of values of nodes without ancestor-descendant relationships can be calculated with dynamic programming. Let $DP(a)$ represent the answer for the subtree of node a. Let array c represent the children of node a and let its size be d. By considering whether an hour is assigned to node a, we can see that $DP(a) = max(k[a], DP(c[0]) + DP(c[1]) + ... + DP(c[d-1]))$. This calculates the answer in $\mathcal{O}(n)$.

Subtask 3

Additional constraints: No additional constraints

A key observation is that even for larger values of m, there will exist an optimal solution where all pairs of ancestor-descendant nodes will not both have nonzero hours. This is because, when

given some optimal distribution of hours that does not satisfy this condition, we can first locate the node that:

1. has non-zero hours assigned
2. has descendant(s) with non-zero hours
3. has the greatest depth out of all nodes satisfying the previous requirements

The reasoning mentioned in subtask 1 can be applied to this node and its descendants, thus reducing either the node's hours or the descendant's hours to zero. This reduces the number of ancestor-descendant pairs with non-zero hours, and the process can be applied repeatedly until no such pairs remain. Thus, there exists an optimal solution with no such pairs. The answer can be found by applying the dynamic programming algorithm from subtask 2, then multiplying the result by m.

Subtask	**Answer**
1	428036307
2	34480011
3	573574206

Task 29: `Mixing Sauces`

Authored by: Lim Rui Yuan (oolimry)

Prepared by: Ryan Goh (rgca)

Editorial written by: Shor the Duck (Ernest Kiew)

Subtask 1

Additional constraints: $n = 2, m = 10$ We can solve this subtask by considering equations.

If we use x parts of base 1 and y parts of base 2, such that $x + y = 1$, our final result is the pair $\{x \times c[1] + y \times c[2], x \times d[1] + y \times d[2]\}$.

If we want the saltiness to be equal C and sweetness to be equal D, then we form the equations:
$x \times c[1] + y \times c[2] = C$
$x \times d[1] + y \times d[2] = D$

Since $c[1], c[2], d[1]$, and $d[2]$ are all constants, both equations are linear equations.

Using some school Mathematics, we can solve for x and y. If a pair of solutions x and y exist, then it is possible to make that sauce.

Subtask 2

Additional constraints: $n = 3, m = 6$

It is generally a good idea to visualise problems when you do not have ideas.

In this case, a 2-dimensional visualisation where the saltiness is the x-coordinate and sweetness is the y-coordinate is particularly helpful.

Using this visualisation, we plot three points corresponding to our base sauces.

We define a reachable sauce as a sauce that can be made using only our base sauces.

We must now figure out the area of sauces that are reachable. Then, a sauce can be made if its coordinate is reachable.

Firstly, it is obvious that the points on the base sauces are reachable.

We can simply use only that base sauce.

But what if we want to make any other sauce?

Let's start the sauce-making process with a choice of the first base sauce.

This puts our position at the position of the first base sauce.

The key observation is that when we add some amount of the second base sauce, we are moving our current position towards that second base sauce.

In fact, every position **on the line** between the positions of both base sauces is reachable this way.

Now, when a third base is added, we can start with any point on the line between the first two sauces, and move towards the third base.

This forms a triangle of reachable sauces. (See Figure 1 for illustration)

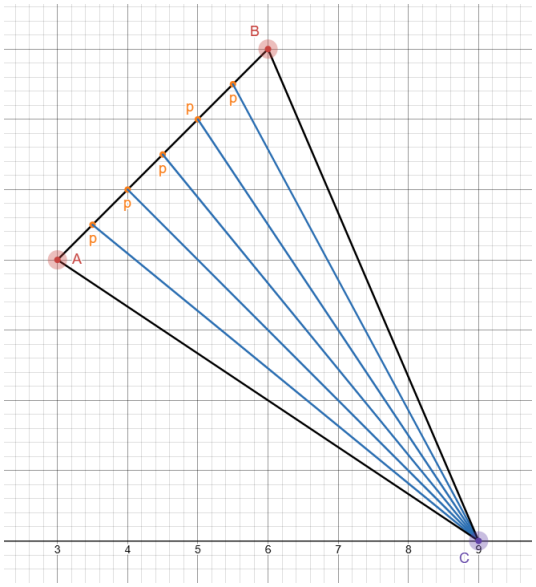

Figure 1: Starting from a point p between points A and B, we can fill in the space within the triangle.

There is an infinite number of possible points p between points A and B, but by picking an appropriate p, we can reach any point inside the triangle.

Hence, we just output 1 if the point is within the triangle ABC, and 0 otherwise.

Subtask 3

Additional constraints: $n = 100$, $m = 40$

If we consider the triangle formed by every triplet of points, we know that any area covered by any triangle is reachable.

If we combine the area formed by all triangles, we actually get a Convex Hull!

You can refer to Wikipedia to learn more.

The outermost sauces form a convex shape, beyond which we cannot reach by any combination of sauces.

By checking if each point is within the Convex Hull, we can solve the problem. (See Figure 2 for illustration)

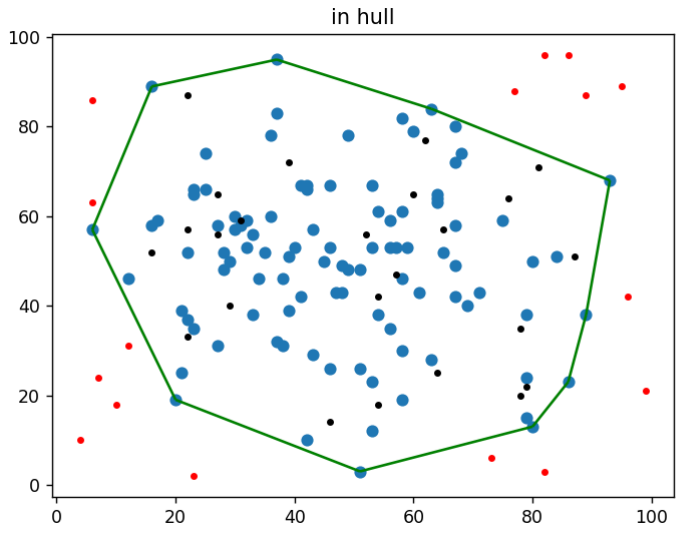

Figure 2: Subtask 3's input visualised, kindly provided by the problem preparer

Subtask	Answer
1	1001100011
2	110100
3	10001111010011101010111100001011011111110

Task 30: `Mountain Ranges`

Authored by: Yaw Chur Zhe (pavement)

Prepared by: Yaw Chur Zhe (pavement)

Editorial written by: Yaw Chur Zhe (pavement)

Subtask 1

Additional constraints: $1 \leq n \leq 20$

In this subtask, it is enough to iterate over all non-empty subsequences of the given array. Checking if a subsequence is beautiful can be done straightforwardly in $\mathcal{O}(n^2)$ time, or with a monotonic stack in $\mathcal{O}(n)$ time.

Time complexity: $\mathcal{O}(n2^n)$

Subtask 2

Additional constraints: $1 \leq n \leq 2000$

Let us analyse the structure of beautiful mountain ranges. Let a be a beautiful mountain range. We can make the following observations:

1. $|a|$ (the length of a) is odd. Since the shortest $\left\lfloor \frac{|a|}{2} \right\rfloor$ mountains will never be able to see strictly more than half of all mountains (and hence are not vantage points), there can never be strictly more than $\frac{|a|}{2}$ vantage points if $|a|$ is even.
2. Let m be the median of a (the median is unique because $|a|$ must be odd). Then, all mountains shorter than or equal to m must form a contiguous subarray of mountains in a. Otherwise, m would not be a vantage point, resulting in strictly less than $\frac{|a|}{2}$ vantage points, a contradiction.
3. For mountains taller than m, they form a bitonic sequence surrounding the contiguous subarray of mountains formed by mountains shorter than or equal to m. This is the only possible configuration where all the mountains taller than m are vantage points.

Figure 3: Structure of a. The red block is strictly decreasing, and the blue block is strictly increasing.

Graphically, the structure of a will look like Figure 3.

If a mountain range of length $l > 1$ exists, a mountain range of length $l - 2$ must surely exist. This motivates an approach that involves binary searching on the maximum length l of a mountain range. Subsequently, we will assume that l has been fixed.

Let x be the index of the rightmost element in the red block, and let y be the index of the leftmost element in the blue block. Observe that at least one of x or y must exist. This is because, if both red and blue blocks were empty, then m would not be the median in the black block (assuming $|a| > 1$). Without loss of generality, assume that x exists (reverse the array if only y exists), and if y also exists, further assume that x is shorter than y ($h[x] < h[y]$).

We can iterate over possible values of x. Let $\text{lds}[x]$ be the length of the longest decreasing subsequence (in the given array h) that ends at x.[3] This tells us the maximum length of the red block. We need the blue block to have length $l_{\text{blue}} = \max(0, \lfloor \frac{l}{2} \rfloor - \text{lds}[x])$. Let $\text{lis}[y]$ be the length of the longest increasing subsequence (in h) that starts at y. In other words, we need $\text{lis}[y] \geq l_{\text{blue}}$. We do no worse by setting y to be the rightmost possible index that satisfies $\text{lis}[y] \geq l_{\text{blue}}$ and $h[x] < h[y]$. Finding such a y can be done in $\mathcal{O}(n)$.[4]

Now that we have our candidate values of x and y, we can count the number of values in $h[x+1], h[x+2], \ldots, h[y-1]$ that are strictly less than $h[x]$. If this count is at least $\lceil \frac{l}{2} \rceil$, then there exists a mountain range with length l. Notice that there is no need to explicitly determine the value of m.

Since there are n candidates values for x, and each of them require $\mathcal{O}(n)$ time to process, the overall time complexity is $\mathcal{O}(n^2)$.

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

Subtask 3

Additional constraints: $1 \leq n \leq 200\,000$

[3] Pre-computation of lds requires $\mathcal{O}(n \log n)$ time, a standard longest increasing subsequence problem.
[4] Remember to handle the edge case where y does not exist ($l_{\text{blue}} = 0$).

In the last subtask, we may use data structures to speed up the process of finding y. By performing a binary search over a Segment Tree and sweep-line techniques, the time complexity can be reduced to $\mathcal{O}(n \log n)$.

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

Subtask	Answer
1	60
2	7388
3	441672