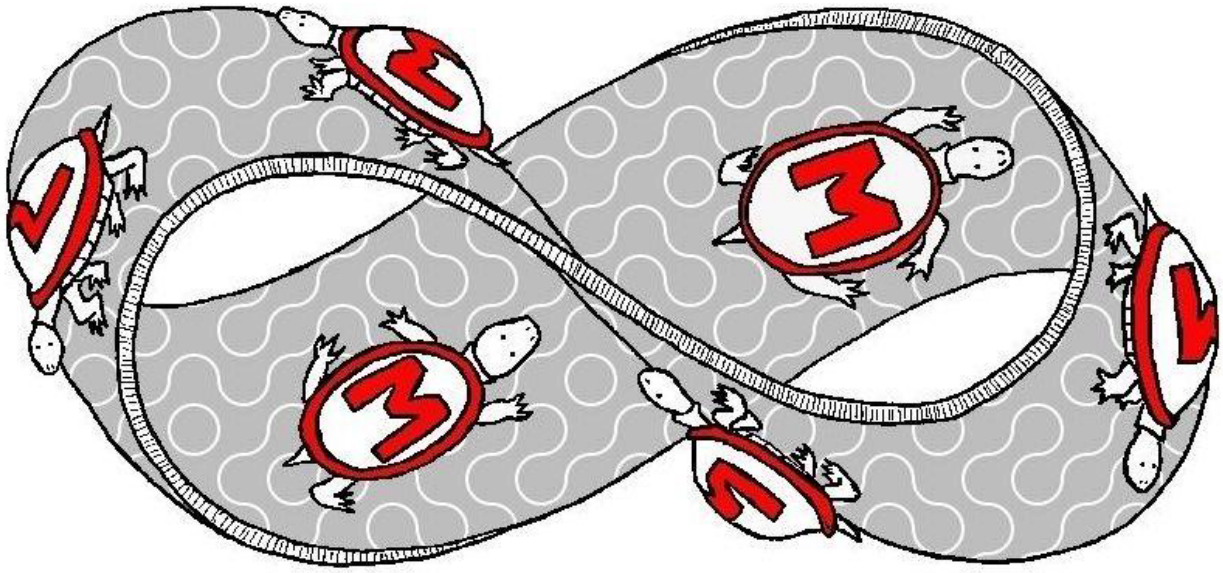
2023 University of Maryland High School Programming Contest


2
3
3
1
2
1

Problem 1: Relatively Prime Floating Mountains

The human scientists that are studying the floating mountains of Pandora have made another interesting observation. In many places, they have noticed that the mountains are arranged in a perfect circle. Furthermore, if one were to take the set of distances between each pair of adjacent mountains, any two of those distances are relatively prime.

Two integers are relatively prime if their greatest common divisor (GCD) is 1. Here is a simple pseudocode to find the GCD of two numbers, *a* and *b* (Euclid's algorithm):

```
function gcd(a, b):
   if b = 0:
        return a
   else:
        return gcd(b, a % b)
```

Here a%b (also written as `a mod b`) is the remainder after dividing a by b. The scientists would like your help to confirm this hypothesis. Specifically, given a set of numbers, you have to decide whether all pairs of numbers from that set are relatively prime, or whether there are two numbers (of those) that are *not* relatively prime.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
static boolean allPairsRelativelyPrime(int[] distances)
```

which returns an

```
boolean
```

Indicating whether all pairs of numbers are relatively prime.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be one number, *n,* denoting the number of distances, followed by *n* numbers that denote distances between adjacent mountains. You can assume that *n* <= *50,* and that all the distances are <= 10000.

Output:

For each test case, the program will output whether all pairs of numbers are relatively prime or not.

Note:

We have provided a skeleton program that reads the number of test cases and the input numbers. It also prints the output based on the `allPairsRelativelyPrime()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
3 3 2 3 5 5 2 5 7 10 11 7 10 17 21 23 79 83 103	 All pairs relatively prime There are some pairs that are not relatively prime All pairs relatively prime

Problem 2: Tsireya's Number Play

In Pandora, the citizens of the Water Nation are known for their love of numbers. They spend hours playing with numbers and challenging each other with puzzles and games. Tsireya, a math enthusiast, is one of the brightest minds of her generation. She has recently come up with a new problem that has been puzzling her friends and fellow clan members.

The problem goes like this: given a number and two more digits, find the largest number you can make by inserting those two digits into the number (anywhere).

For example, starting with 5716 and two digits, 1 and 9, the largest number we can get by inserting 1 and 9 into the number is: 957161. Other options are all smaller (e.g., 597116, or 517169, etc.). On the other hand, if you are given digits 0 and 0, then the largest number you can make is: 571600.

Can you help Tsireya by confirming her own answers, before she asks her friends?

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
findLargestNumber(int N, int digit1, int digit2)
```

which returns an

```
int
```

which is the largest number you can make by inserting `digit1` and `digit2` into `N` anywhere.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be three numbers (*N, digit1, digit2*). You can assume that *1 <= N <= 9999999* (so the output numbers fit in an `int`). The two digits might be the same and are between 0 and 9 (inclusive).

Output:

For each test case, the program will output the largest number that can be constructed by inserting *digit1* and *digit2* into *N*.

Note:

We have provided a skeleton program that reads the number of test cases and the input numbers. It also prints the output based on the `findLargestNumber()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
3 5716 1 9 5716 0 0 5716 2 3	 957161 571600 573216

Problem 3: Interleaved Omangi Messages

The members of the Omangi clan use a unique method of encryption to send secret messages to each other. They start by taking a message (to be encrypted) and appending extra 'Z' characters to the end so that the total length of the message is a multiple of *k,* a pre-selected number. They then split the message into equal-length substrings of length *k* each, and interleave them together to create the encrypted message.

For example, if the message is "KATARA" and *k* = *4*, they would first add two Zs to get the message "KATARAZZ". They would then split the message into "KATA", and "RAZZ"; they would then take the first letter from each split word in order, and then the second letter, and so on, to create the encrypted message "KRAATZAZ". On the other hand, if *k* = *2,* then we don't need to do padding – the message would get split into "KA", "TA" and "RA", and the output message would be: "KTRAAA".

Your task is to write a program that can decrypt these messages. You will be given an encrypted message and the length of the substring used in the interleaving process. You need to find the original message that was encrypted.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
static String decrypt(String s, int k)
```

which returns a

```
String
```

which is the original message (padded with Zs in some cases).

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be one string, consisting of characters A-Z, and of length between 1 and 100 (inclusive), and a number, the length of the substring.

Output:

For each test case, the program outputs the decrypted message. Note that, there may be extra ('Z') characters at the end of the decrypted message (because of the padding). Those characters should be left as is (see examples below).

Note:

We have provided a skeleton program that reads the number of test cases and the input strings one by one. It also prints the output based on the `decrypt()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
4 4 KRAATZAZ 2 KTRAAA 3 AAAATCTWTKDN 4 BULERFYSZOEZ	 KATARAZZ KATARA ATTACKATDAWN BEYOURSELFZZ

Problem 4: Woodsprites' Patterns

A few days after his discovery of the interesting pattern of Ombratrees, Neteyam noticed a group of woodsprites arranged in a similar fashion, self-arranged in a series of lines. He isn't really surprised given all life on Pandora is interconnected. This pattern, however, is much larger since woodsprites are much smaller than Ombratees.

The pattern was slightly different though: if the first line has *N* woodsprites, then the second line has ⌊***N/2***⌋ = ***floor(N/2)*** woodsprites (i.e., largest integer that is smaller than or equal to *N/2*), the third line has ⌊***N/3***⌋ woodsprites, and so on until the last line with a single woodsprite. Neteyam would like your help to calculate the total number of woodsprites in this pattern.

For example, if *N = 10*, then the pattern goes:
⌊10/1⌋ = 10, ⌊10/2⌋ = 5, ⌊10/3⌋ = 3, ⌊10/4⌋ = 2, ⌊10/5⌋ = 2, ⌊10/6⌋ = 1, ⌊10/7⌋ = 1, ⌊10/8⌋ = 1, ⌊10/9⌋ = 1, ⌊10/10⌋ = 1, giving us a total of 27 woodsprites.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
long countWoodsprites(long N)
```

which returns an

```
long
```

that is the total number of woodsprites.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be a single number (*N*) indicating the number of woodsprites on the first line. You can assume that *1 <= N <= 2^60.* Make sure you test on large numbers before submitting.

Output:

For each test case, the program will output the total number of woodsprites.

Note:

We have provided a skeleton program that reads the number of test cases and the input numbers. It also prints the output based on the `countWoodsprites()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
3 10 200 5000	 27 1098 43376

Problem 5: Unstable Floating Mountains

After receiving your program from Practice Problem 2, the scientists used it to try to verify their conjecture (that the weights of stacked floating mountains formed a generalized Fibonacci sequence). However, although they were able to verify the conjecture for a large number of cases, they discovered that there were stacked structures that did not satisfy the property.

Now they have another conjecture, inspired by some structures that they saw Kiri construct. They believe that the sequences of weights of those stacks of floating mountains follow what's called a **"subprime modular Fibonacci sequence**". As with generalized Fibonacci sequences, every number in the sequence (except first two) depends on the previous two numbers, but with a twist. Let `f(1)` and `f(2)` denote the first two numbers – then, for `n > 2`, we have the following recurrence:

```
let a = f(n-1) + f(n-2) mod 10007
if a is prime:
            f(n) = a
else:
            f(n) = largest factor of a (i.e., the largest b s.t. b divides a)
```

Note, 10007 is a fixed prime number dictated by some intrinsic gravitational properties of Pandora, and does not have any special significance for the code you have to write. This also means that all weights are between 0 and 10006, both inclusive (yes, there are mountains of weight 0).

The scientists are quite confident about this recurrence -- however, the actual sequences that they see don't appear to follow this pattern entirely. They believe that some mountains drifted away after forming an initial sequence that did satisfy the above property. They have worked out that at most 9 consecutive mountains can drift away without compromising the structure. They would like your help with verifying this conjecture.

Specifically, given a sequence of numbers, all between 0 and 10006, you must determine if the numbers are from a subprime modular Fibonacci sequence (called "original" sequence), such that all consecutive pairs of numbers in the input sequence are less than 10 apart (i.e., fewer than 9 items between any consecutive pair of numbers) in that original sequence.

For example, the following sequence satisfies the requirements:

1 5 7 11 17

These 5 numbers are part of a larger subprime modular Fibonacci sequence below:

1 2 3 5 4 3 7 5 6 11 17 (and also of: 1 0 1 1 2 3 5 4 3 7 5 6 11 17)

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
int[] solveSubprimeFibonacci(int[] sequence)
```

which returns an

```
int[]
```

Which should be “null” if the input sequence doesn’t satisfy the property, and it should be the first five elements of the original sequence if it does. If there are multiple original sequences that satisfy the requirements, you should output the one that has the smallest second element (since the first element in the output will be the first element in the provided sequence, if the property is satisfied). In the example above, the first five elements of the second sequence will be output (1 0 1 2 3).

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be one number, *n,* denoting the length of the sequence, followed by *n* numbers that form the sequence. You can assume that *n* <= *50,* and that all the sequence numbers are between 0 and 10006.

Output:

For each test case, the program will output whether the property is satisfied, and if yes, the first five numbers of the sequence.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
3 5 1 5 7 11 17 7 1 5 23 51 97 17 45 5 1 14 142 46 69	 YES: 1 0 1 1 2 YES: 1 0 1 1 2 NO

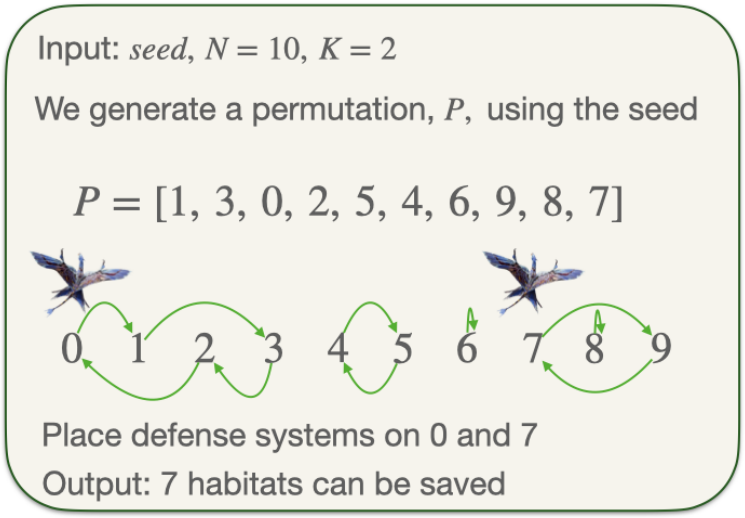
Problem 6: Path-Breaker

The Na'vi have received a message indicating the flight paths of a battalion of RDA attack ships. The ships plan to attack a set of Na'vi habitats. Each attack ship will fly a pattern where it visits a set of distinct habitats before ending its mission. Each habitat is attacked by a single ship, and every habitat is attacked by some ship.

More precisely, the flight paths for *all of the ships* can be described using a *single* permutation P on the integers [0 … N-1], representing habitats that the set of ships will attack. Given the set of habitats a given ship will attack, each ship will start its attack with the smallest habitat in its set of targets (i.e., the habitat with minimum index). Let's say that's habitat *i*. After attacking *i*, the ship will continue to *P[i]*, *P[P[i]],* and so on, until returning to *i*, at which point this ship ends its attack.

The Na'vi would like to disrupt the attacker's plans and save as many habitats as possible. However, they only have a fixed number of surface-to-air defense units, and must deploy them strategically. Each defense unit can disrupt a single attack ship. Placing a defense unit on habitat i ensures that i and any habitats the ship would have visited after visiting i are saved.

Determine the maximum number of habitats that can be saved by strategically placing K defense units. We show an example instance below:



In this example, there are 4 attack ships that will be deployed. The first ship attacks 0, P[0] = 1, P[P[0]] = 3, and P[3] = 2 in that order and then it returns back to P[2] = 0. The second ship attacks 4 and 5 in that order, and so on. To maximize the number of habitats saved, the K=2 defense units should be placed on habitats 0 and 7 to save a total of 7 habitats.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
int NumHabitatsSaved(Integer[] P, int, N, int K)
```

which returns a

```
int
```

Indicating the number of habitats that can be saved by placing K defense units.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be one number representing a random seed to generate the permutation, one number specifying the number of habitats in total, and one number representing the number of defense units (K). We have provided the code that generates the permutation.

Output:

For each test case, the program should output the number of habitats that can be saved.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
3 42 10 1 43 10 1 2 100 3	 Num Habitats Saved: 5 Num Habitats Saved: 7 Num Habitats Saved: 90

Problem 7: Akula Topple

Spider is competing in the annual Akula Topple competition on Pandora. Don't worry, he doesn't actually have to topple any Akulas – rather the game is played with small Akula figurines. This is a two-player game, played with 5 distinct Akulas, labeled A to E. The 5 Akulas are initially arranged in a random order on a board from top to bottom. Each player is also assigned 3 of the Akulas randomly, and the goal of the game is to rearrange the Akulas so that your Akulas occupy the top positions at the end of the game.

Each player has 6 action cards: 2 "Akula 1 up" cards, 2 "Akula 2 up" cards, and 2 "Akula topple" cards. The "Akula up 1" card allows you to move any Akula (except the top one) up one position in the sequence, while the "Akula 2 up" card allows you to move any Akula up two positions in the sequence (this card cannot be used for the first two Akulas). The "Akula topple" card allows you to move the top Akula to the bottom of the sequence. In each case, the other Akulas move up or down accordingly while keeping their order.

After both players have played all of their cards, whichever player has the most Akulas in top 3 positions (out of their 3 Akulas) wins. If both players have the same number of Akulas in top 3, then Player 1 wins (since Player 2 plays the last move which is a significant advantage).

As an example, let's say the initial order is D C E A B (D at top), player one's Akulas are A C D, and player two's Akulas are A B E. Following shows a first few possible steps:

(1) Player 1 plays "Akula 2 Up" card on E, to get to: E D C A B
(2) Player 2 plays "Akula Topple" card to get to: D C A B E
(3) Player 1 plays "Akula 1 Up" card on E to get to: D C A E B
(4) Player 2 plays "Akula Topple" card to get to: C A E B D

If the game were to end here (we still have 8 cards to play in total), then both the players have 2 Akulas in the top 3, so Player 1 wins.

Spider would like your help in deciding what is the best next card to play. This is a "complete information" game with no ties, and as such, one of the players is guaranteed to have a winning strategy for any given starting arrangement.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
boolean firstPlayerWins(char[] initial, char[] player1, char[] player2,
                                                    ArrayList<String> moves)
```

where initial is an array of length 5 indicating the initial arrangement (from top to bottom), and player1 and player2 are two arrays of length 3 containing the Akulas assigned to each player.

Your code should also populate the “moves” array to capture a sequence of moves assuming each player plays their best move. If a player does not have a winning move, then they play the first available move in the sequence: “Akula 2 Up <position 3>”, “Akula 2 Up <position 4”>, …, “Akula 1 Up <position 2>”, “Akula 1 Up <position 3>”, …, “Akula Topple”. Your code should populate the specific Akula being moved (and not list the position). More details in the Skeleton code file.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be three character arrays, of length 5, 3 and 3 respectively.

Output:

For each test case, the program will output who wins the game (assuming both the players play their best moves), and also a sequence of moves.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you). There is another example provided in the test input file. Since Player 2 doesn’t have winning moves, its moves are in the order listed above (Akula 2 Up moves first, and so on).

Important: We recommend that you submit your solution if you believe it is right, even if it does not match the provided output precisely.

Input:	**Output:**
1 CDEAB ACD ABE	========== Test Case 0 ========== Initial State: CDEAB Player 1: ACD Player 2: ABE *** First Player Wins *** Start --> CDEAB Akula Topple --> DEABC Akula Up 2 A --> ADEBC Akula Up 2 E --> EADBC Akula Up 2 D --> DEABC Akula Up 2 A --> ADEBC Akula Up 1 D --> DAEBC Akula Up 1 A --> ADEBC Akula Up 1 D --> DAEBC Akula Up 1 A --> ADEBC Akula Topple --> DEBCA Akula Topple --> EBCAD Akula Topple --> BCADE

Problem 8: Hidden Pathways

The Na'vi have taken refuge in their secret lair. To plan for emergency evacuations, they have set up a sequence of hidden paths that provide safety while escaping from their home. Being mathematically minded, they have devised the paths based on the following construction:

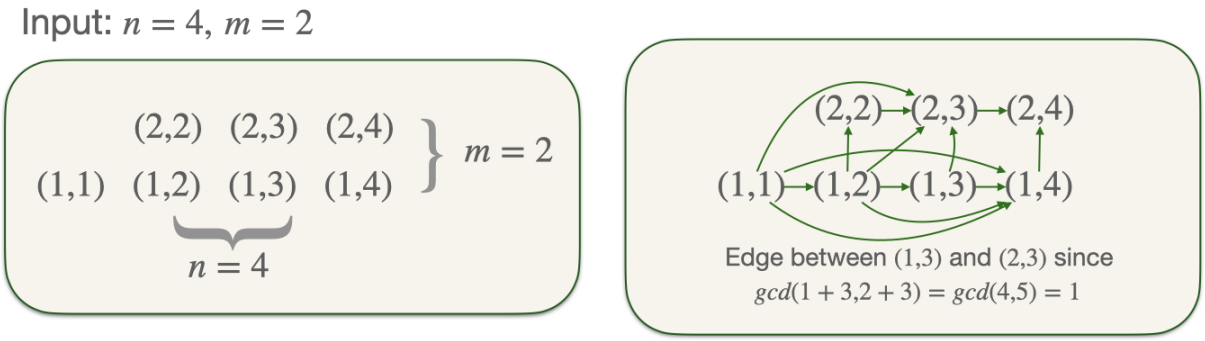
Given two numbers n and m, consider the rectangular grid (or matrix) with m rows and n columns with coordinates (a, b) where 1 <= a <= m and 1 <= b <= n. We will consider only coordinates (a, b) where a <= b to be valid locations.

Over the years, the Na'vi have built many hidden pathways between locations on the grid. They constructed these pathways using a scheme where a pathway exists between location (x, y) to (x', y') if the number x + y is relatively prime with x' + y' and x + y < x' + y'.

Observe that pathways only go "up" and "to the right" on the grid.

The Na'vi start at location (1, 1), and must travel to location (m, n) via a sequence of pathways to reach safety. Please assume that n + m > 2.

Your task is to count the number of paths that exist between (1, 1) and (m, n). Consider the following example:



In the example above, the input has n=4, m=2. The grid of locations is shown in the first panel, and the hidden pathways between locations is shown in the panel on the right. The total number of hidden paths in this example is 7.

1,1 → 1,4 → 2,4
1,1 → 1,2 → 1,4 → 2,4
1,1 → 1,2 → 1,3 → 1,4 → 2,4
1,1 → 1,2 → 1,3 → 2,3 → 2,4
1,1 → 1,2 → 2,3 → 2,4
1,1 → 1,2 → 2,2 → 2,3 → 2,4
1,1 → 2,3 → 2,4

The provided skeleton handles parsing the values m and n. You need to implement the method

```
long NumHiddenPaths(int m, int n)
```

which returns a

```
long
```

Indicating the number of hidden paths between (1, 1) and (m, n).

Examples:

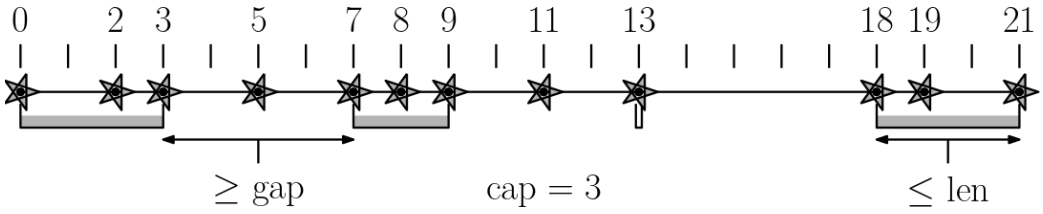
The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
2 2 4 3 4	 Num Hidden Paths: 7 Num Hidden Paths: 22

Problem 9: Starfish Collecting

Kiri is diving to collect starfish. The starfish are arrayed in a line along the bottom of the sea. Kiri makes multiple dives, and collects a sequence of starfish with each dive, which she puts in her knapsack. Her knapsack holds a fixed capacity of starfish. She can only stay underwater for a short time, which limits the length of the interval between the first and last starfish she collects on each dive. Finally, it takes time to resurface, and so there is a minimum gap between the end of each interval and the start of the next.

Help Kiri get the most starfish. You are given her knapsack capacity (cap), the maximum length of any dive interval (len), and the minimum gap between two consecutive intervals (gap). You are also given the starfish coordinates as a sequence x of distinct, nonnegative integers. The objective is to compute a set of disjoint intervals, each starting and ending at a point of x (possibly the same point), such that no interval contains more than cap starfish, the length of any interval is at most len, and any two consecutive intervals are separated by a distance of at least gap.



Example: cap=3, len=5, gap=4 with 12 points x = {0, 2, 3, …, 21}.
She can collect 10 starfish with the four dive intervals [0,3], [7,9], [13,13], and [18,21].

Write a program that computes the set of intervals that contains the maximum number of starfish. If there are multiple solutions, output the one whose sorted sequence of endpoints (start and finish) is minimum in lexicographical order. The provided skeleton handles the input of the values and the output messages. You only need to implement the method:

```
static ArrayList<Integer> getStarfish(int cap, int len, int gap, int[] x)
```

It returns an ArrayList consisting of the interval endpoints, sorted from left to right. In the above example, the 4 intervals would be recorded in the ArrayList as [0, 3, 7, 9, 13, 13, 18, 21].

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, each test case begins with a line containing three integers with the values cap, len, and gap. The next line contains the number of starfish n. This is followed by n lines, each of which contains the integer coordinates of a

starfish sorted in increasing order. You may assume that cap is strictly positive, and len and gap are nonnegative. All starfish coordinates are nonnegative, and there are no duplicates. All quantities are less than or equal to 1000.

Output:

For each test case, the program will output the number of intervals. For each interval, it outputs the interval endpoints, the interval's length, the number of points in the interval, and (except for the first interval) the size of the gap between this interval and the prior one. Finally, it outputs the total number of points covered.

Note:

Our skeleton program will read the input and generate the output. You need only implement the `getStarfish()` function.

Examples:

The example below shows a single test case.

Input:	**Output:**

1
3 5 4
12
0
2
3
5
7
8
9
11
13
18
19
21
	Test case: 1
	  Capacity: 3, Length: 5, Gap: 4
	  Points: 0 2 3 5 7 8 9 11 13 18 19 21
	  Found 4 intervals:
	  [0, 3] of length 3 containing 3 points
	  [7, 9] of length 2 containing 3 points gap size 4
	  [13, 13] of length 0 containing 1 points gap size 4
	  [18, 21] of length 3 containing 3 points gap size 5
	  Total covered: 10