JSC INGENIUM TECHNICAL TEST

Answer the test in English, please.

You are part of the Resistance in the Star Wars universe.

You are at an old rebel base on planet Craig.

First Order forces will land on the planet in the next few hours.

You must write an algorithm in C# to load the right number of laser cannons into the only functional drone the Alliance has left in order to deploy them to the nearby peaks.

The algorithm is fed with a collection containing a finite number of radar sourced terrain heights. They are all greater than or equal to zero.

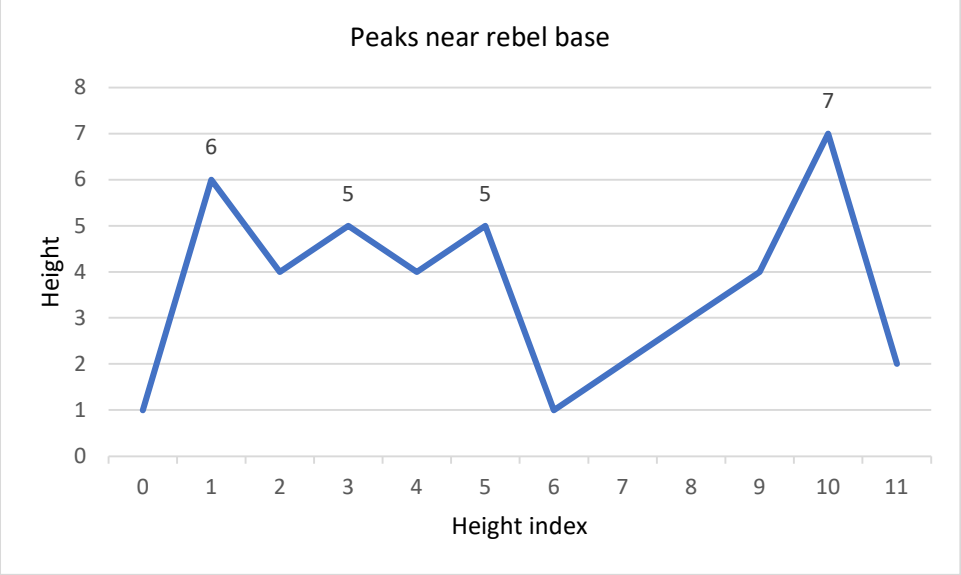
A peak is an element in the collection with both lower preceding and following neighboring heights.

The first and last elements in the collection are, by definition, not peaks, because they only have one neighboring height.

For example, the collection:

```
var heights = new uint[] { 1, 6, 4, 5, 4, 5, 1, 2, 3, 4, 7, 2 };
```

Can be depicted as:



And we find 4 peaks, corresponding to indexes 1, 3, 5 and 10 (heights 6, 5, 5 and 7, respectively).

You have to choose how many laser cannons the drone needs to load. The aim is to set the maximum number of cannons on the peaks, complying with some rules stated by R2D2:

1. The cannons must be deployed to peaks, and each peak admits a single cannon.
2. If you choose to load **k** cannons, and **k** > 1, the distance between any two cannons (once deployed) must be greater than or equal to **k**. If two cannons are deployed to the peaks on indexes P and Q, the distance between them is defined as the absolute value |P – Q|.

Let's analyze the example of collection with 12 elements:

- If you choose to load a single cannon, it can be deployed to any peak (indexes 1, 3, 5 or 10).
- If you choose to load 2 cannons, they can be deployed to:
    - Peaks at indexes 1 and 3.
    - Peaks at indexes 1 and 5.
    - Peaks at indexes 1 and 10.
    - Peaks at indexes 3 and 5.
    - Peaks at indexes 3 and 10.
    - Peaks at indexes 5 and 10.
- If you choose to load 3 cannons, they can only be deployed to the peaks at indexes 1, 5 and 10.
- Any other number of cannons do not comply with R2D2's rules.

The solution takes the form of a C# interface:

```
public interface ICannonLoader
{
    /// <summary>
    /// Calculates the number of cannons to be deployed to a collection of heights.
    /// </summary>
    /// <param name="heights">A collection of heights.</param>
    /// <returns>The number of cannons.</returns>
    int GetCannonCount(IReadOnlyList<uint> heights);
}
```

Your job is to provide at least one implementation of ICannonLoader.

GetCannonCount must return 3 for the example of collection shown.

R2D2 adds these last-minute hints:

- For the following collection, the answer is 14:
  { 206, 11, 645, 601, 770, 755, 858, 637, 7, 540, 986, 935, 77, 968, 478, 18, 943, 352, 242, 454, 514, 196, 592, 926, 164, 153, 149, 605, 458, 193, 22, 263, 309, 198, 921, 160, 699, 933, 207, 337, 90, 552, 474, 490, 81, 788, 671, 76, 464, 340, 717, 394, 634, 795, 639, 10, 418, 719, 436, 617, 605, 252, 179, 813, 517, 723, 625, 879, 46, 318, 319, 9, 145, 904, 363, 962, 721, 101, 845, 376, 482, 73, 141, 117, 475, 627, 629, 513, 558, 911, 741, 980, 497, 590, 731, 837, 372, 660, 973, 357, 162, 791, 29, 145, 21, 708, 249, 729, 432, 676, 288, 316, 873, 59, 135, 532, 111, 61, 337, 889, 93, 112, 320, 135, 576, 339, 512, 554, 173, 525, 589, 61, 128, 487, 213, 690, 43, 995, 96, 767, 378, 900, 81, 881, 623, 424, 106, 437, 455, 199, 726, 311, 322, 83, 264, 50, 471, 894, 568, 682, 196, 695, 555, 906, 87, 226, 188, 385, 845, 444, 488, 115, 964, 570, 511, 193, 31, 438, 511, 152, 232, 74, 99, 973, 798, 278, 738, 44, 223, 163, 219, 149, 426, 414, 571, 683, 344, 174, 794, 396 }
- Remember to validate the inputs. Find a suitable static method in the ArgumentNullException class to do so.
- The list of heights is indexable (you can use the indexer operator []). You also know its Count.
- You must find a solution among a set of possible solutions. Remember that the less the candidate number of cannons is, the less the distance between peaks can be. Don't fall into the trap of trying a single value.
- Use well-known algorithms. In this case, as the set of numbers of cannons to try form an ordered sequence, R2D2 suggests you: linear search, binary search. The latter is better for random data, because it discards half the alternatives on each iteration, so it rapidly converges no matter where the solution is.

- If you choose binary search, remember that recursion is always a concern for the risk of StackOverflowException (in languages, like C#, with no tail recursion optimization). Consider turning recursion into iteration (a loop) instead.
- Consider that there is an upper bound on the number of peaks for a given number of heights. It is an order of magnitude less than the latter. If you need to eagerly gather all the peaks into a List<T>, remember you can pass the upper bound as the *capacity* argument to the constructor. Depict the most compact peak arrangement and deduce the relationship between the number of heights and the maximum number of peaks. Calculating the needed *capacity* in advance will prevent the list from having to recreate its internal array on the way (something that can happen if the list is initialized with the default capacity).
- Consider that there is an upper bound on the number of cannons for a given number of heights. Depict the most compact cannon arrangement (complying with the distance rule) and deduce how many other heights you need around and between the cannon peaks. Solve the equation for the maximum number of cannons. It is advisable to locate the first peak and save its index, because ignoring the heights before the first peak will reduce the calculated upper bound. Remember that, for any k > 1, k (k – 1) >= (k – 1) (k – 1). And that square root is monotone. Obviously, if you gather all peaks first, there is a second upper bound on the number of cannons, which is the number of peaks. You can use the upper bound regardless of the search algorithm you use, because in both cases you will reduce the set of values to try.
- Consider not gathering all peaks in advance, especially if you use binary search. In that case, calculating the first peak and saving it is advisable, because the first peak is a good starting point for each cannon number check. Remember that binary search discards many values on each try and each tried value forces an increasing distance between peaks, which allows you to discard an increasing number of heights to check for peaks. Searching for peaks as you are checking each number of cannons is a perfectly viable approach in those circumstances and avoids having to gather them all in advance.
- Test you program against a fair number of different inputs. Remember that good unit tests should include edge cases: the list is *null*, the list is empty, the list has only one height, the list has a plateau… Consider an xUnit.net project for performing those tests, and even designing the tests first and only then the algorithm (TDD, Test Driven Development). Remember that you will be evaluated mainly by the correctness of the results against different inputs, so it is not advisable to be satisfied with a few biased test inputs.
- If you want a more production-ready solution, you should follow Microsoft's advice to design APIs receiving collections around IEnumerable<T> and not IReadOnlyList<T>, ICollection<T>… As some actual types implementing IEnumerable<T> are amenable to more efficient algorithms than others (indexing, translation to Span<T>, Vector<T>…), Microsoft suggests dynamically casting the collection (in modern C#, pattern matching) in order to use different implementations for different actual collection types:

  https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/guidelines-for-collections#collection-parameters

  You can see this approach in .NET runtime's own code:

  https://github.com/microsoft/referencesource/blob/master/System.Core/System/Linq/Enumerable.cs#L15

Good luck and remember you are the spark to light the fire that will eradicate the First Order!