Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy

AIEP Send-off Python 2 Session 6

# BINARY SEARCH and BISECTION METHOD

Suppose that we have an alphabetical list of 10 anime characters and we would like to find the index of `Kikyo`.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `Amane` | `Arrietty` | `Conan` | `Kagome` | `Kanade` | `Kikyo` | `Kurisu` | `Mahiru` | `Mayuri` | `Okabe` |

A possible (albeit inefficient) approach would be to perform a linear search: start at the very first element, i.e., `Amane`, then iterate through the list until `Kikyo` is found. With this scheme, a total of 6 comparisons are required to find `Kikyo`. But is there a way to take advantage of the fact that the list follows an alphabetical sorting?

We can try starting at the middle of the list, i.e., `Kanade`. Since `Kikyo` comes after `Kanade`, we can discard the entire left half and focus only on all the entries after `Kanade`.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `Amane` | `Arrietty` | `Conan` | `Kagome` | `Kanade` | `Kikyo` | `Kurisu` | `Mahiru` | `Mayuri` | `Okabe` |

Ignoring the discarded elements, we take the middle of the list anew: `Mahiru`. Since `Kikyo` comes before `Mahiru`, we can discord all the entries from `Mahiru` onwards.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| `Amane` | `Arrietty` | `Conan` | `Kagome` | `Kanade` | `Kikyo` | `Kurisu` | `Mahiru` | `Mayuri` | `Okabe` |

Continuing with our strategy, we take the middle again: `Kikyo`. Voila, its index is 5 — and it took only a total of 3 comparisons to find it! With our toy example, 6 versus 3 comparisons may not seem like much of an improvement, but, as systems scale in size and complexity, this difference becomes significantly pronounced.

The algorithm that we have demonstrated is referred to as binary search, which belongs to a class of techniques under the divide-and-conquer paradigm. The qualifier "binary" encapsulates its core advantage: with each iteration, the search space is halved. For comparison, both linear search and binary search take $\Theta(1)$ space; neither of them uses an auxiliary data structure. However, given a list of size $n$, the worst-case time complexity of linear search is $\Theta(n)$ while that of binary search is only logarithmic: $\Theta(\log n)$.

Before we delve further, it must be emphasized that binary search is applicable only if the list is sorted. Trade-offs have to be weighed if it is initially unsorted, as comparison-based sorting algorithms can never perform faster than $\Theta(n \log n)$[1]. If searching will only be done once, a linear search will most likely suffice. On the other hand, if it will be done multiple times (as is typically the case in real-world and competitive programming problems), the overhead from a one-time sorting will be amortized by the speed binary search offers.

[1] For a discussion of this lower bound, you may refer to pages 1 to 3 of these lecture notes.

## RECURSIVE IMPLEMENTATION

Binary search lends itself well to a recursive implementation. `arr` pertains to the list while `key` pertains to the value that we are searching for.

```
def binary_search_recursive(arr, key, low, high):
    if low <= high:
        mid = low + (high - low) // 2
        if key < arr[mid]:
            return binary_search_recursive(arr, key, low, mid - 1)
        if arr[mid] < key:
            return binary_search_recursive(arr, key, mid + 1, high)

        return mid

    return NOT_FOUND

def binary_search(arr, key):
    return binary_search_recursive(arr, key, 0, len(arr) - 1)
```

In Session 4 of the qualifying phase of your AIEP Python 2 training, you were introduced to basic asymptotic analysis. Let us deviate slightly from our main topic and take this opportunity to gloss over the asymptotic analysis of recursive functions. Let $T(n)$ be the running time. Since each recursive call halves the search space, $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$.

The master theorem[2] provides a way to get the asymptotically tight bounds of relations of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. In this case, applying the master theorem yields $\Theta(\log n)$.

## ITERATIVE IMPLEMENTATION

The iterative version of binary search is as follows:

```
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = low + (high - low) // 2
        if key < arr[mid]:
            high = mid - 1
        elif arr[mid] < key:
            low = mid + 1
        else:
            return mid

    return NOT_FOUND
```

[2] For a discussion of the master theorem, you may refer to this page.

An oddity that you might have noticed in both recursive and iterative implementations is the computation of `mid`. The numerator of the straightforward computation,

```
mid = (low + high) // 2,
```

may result in an overflow when the addends are large[3]. In order to avoid this pitfall, we strategically rewrite it as

```
mid = low + (high - low) // 2.
```

For a historical sidenote, you may read this 2006 report from Google. Fortunately, unlike in C++ and Java, integers in Python enjoy **arbitrary precision**, i.e., the only limit to the range of supported values is the hardware (the computer memory) itself. Ergo, integer overflow is not a concern for Python programmers. But this feature already takes a toll on the language's performance; abusing it will only exacerbate the penalty.

In practice, the iterative version of binary search is preferred since recursion consumes more memory and runs the risk of creating too many stack frames, resulting in a so-called "**stack overflow**" — especially if the recursion becomes excessively deep. To mitigate this issue, some compilers implement techniques such as **tail call optimization**. Interestingly, Python does not (and will most likely never) support tail call optimization[4].

## 💡 VARIATION #1: Lower Bound

Consider the following alphabetical list of anime characters. Our task is to find the index of the first occurrence of `Nana`. Consistent with our implementation of binary search, we initialize `low` to 0 (the index of the first element). However, instead of initializing `high` to be equal to the index of the last element, we set it to the length of the list. The computation of `mid` remains unchanged.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Amane | Kikyo | Kikyo | Kikyo | Mahiru | Mahiru | Mahiru | Nana | Nana | Okabe | |
| low | | | | | mid | | | | | high |

`Nana` comes after `Mahiru`. Therefore, we search only the right half.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Amane | Kikyo | Kikyo | Kikyo | Mahiru | Mahiru | Mahiru | Nana | Nana | Okabe | |
| | | | | | | low | | mid | | high |

We have found `Nana`, but we have to be careful. As humans, we know that this is the second occurrence of `Nana`, but computers do not possess a panoramic visual system. In fact, the system only knows that it has found `Nana`, but it is oblivious to which occurrence of `Nana` has been found. Hence, we have to play the skeptic's role, assume that this `Nana` is not the first in the list, and march to the left.

[3] For a discussion of integer overflow and how it leads to unintended behaviors, you may refer to this module.
[4] For more information on Python and tail call optimization, you may refer to this Stack Overflow discussion.


Once again, we have found `Nana` — but, to reiterate, the computer does not know whether this `Nana` is really the first occurrence, so we continue going to the left.


`Mahiru` comes before `Nana`; it seems that we have gone far too left, prompting us to search the right half. Interestingly, the resulting value of `low` this time is not below `high`, signaling that we have reached the terminating condition and that we have obtained the index of the first occurrence of `Nana`.

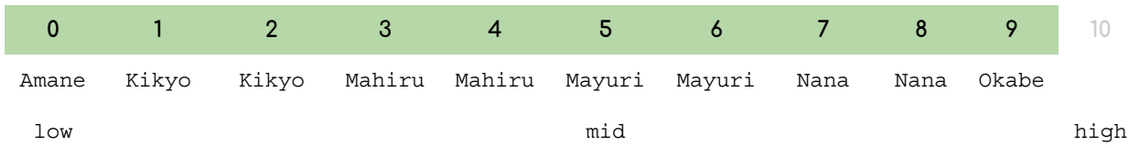

The value of `low` is referred to as the **lower bound** since it is the lowest position where the search key can be inserted without destroying the ordering (in this case, the alphabetical arrangement). Assuming an ascending order, the lower bound can also be viewed as the index of the first value that is **greater than or equal** to the specified search key.

The implementation of the algorithm for getting the lower bound is as follows:

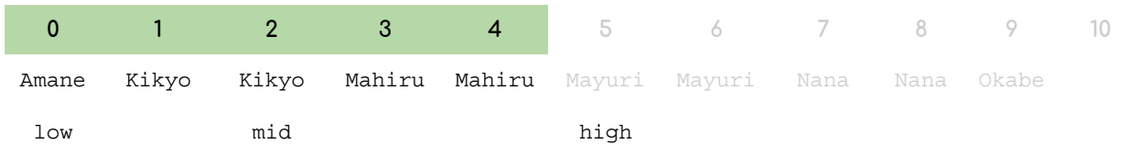
```
def lower_bound(arr, key):
    low = 0
    high = len(arr)

    while low < high:
        mid = low + (high - low) // 2
        if key <= arr[mid]:
            high = mid
        else:
            low = mid + 1

    return low
```

## 💡 VARIATION #2: Upper Bound

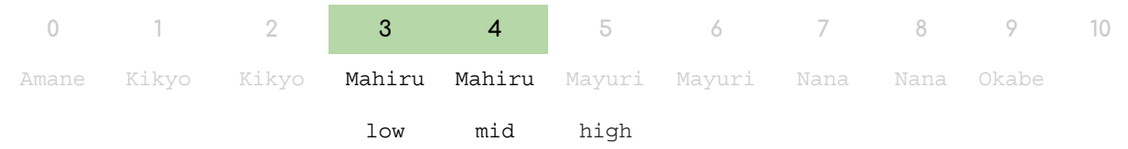
Using a slightly different but still alphabetical list, suppose that our task this time is to find the index of the entry immediately after the last occurrence of `Kikyo`.



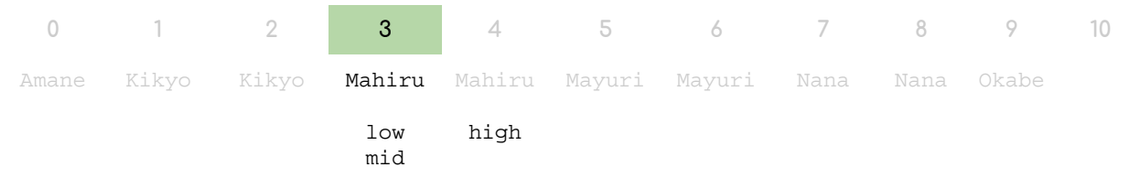

`Kikyo` comes before `Mayuri`; thus, we search only the left half.




We have found `Kikyo`, but, since we want the first entry after it, we march to the right.




We have found `Mahiru`, which is after `Kikyo`. As humans with panoramic visual systems, we know that the end-goal is equivalent to finding the first occurrence of `Mahiru`. However, insofar as the computer is concerned, the only argument we passed (aside from the list) is `Kikyo`; it does not have any idea whatsoever that it has to look for `Mahiru`. Therefore, we are prompted to zero in on the left half.




Once again, we have found `Mahiru` — but, to reiterate, the computer is agnostic to this discovery, so we continue going to the left. Interestingly, the resulting value of `low` this time is not below `high`, signaling that we have reached the terminating condition and that we have obtained the index of the entry immediately after the last occurrence of `Kikyo`.


The value of `low` is referred to as the upper bound since it is the highest position where the search key can be inserted without destroying the ordering (in this case, the alphabetical arrangement). Assuming an ascending order, the upper bound can also be viewed as the index of the first value that is greater than the specified search key.

### 💭 Breakpoint A

1. In getting the lower and upper bounds, what is the wisdom behind initializing `high` to be equal to the length of the list (rather than the index of the last element)?
2. What will the lower and upper bounds be if the search key is not in the list?
3. In the interest of consistency, how would we rewrite our implementation of the usual binary search if `high` were to be initialized to be equal to the length of the list?

The implementation of the algorithm for getting the upper bound is as follows (take note of the highlighted line of code vis-à-vis the one in `lower_bound`):

```
def upper_bound(arr, key):
    low = 0
    high = len(arr)

    while low < high:
        mid = low + (high - low) // 2
        if key < arr[mid]:
            high = mid
        else:
            low = mid + 1

    return low
```

## Illustrative Example Where is the Marble?

*University of Valladolid (UVa) Online Judge 10474*

Raju and Meena love to play with Marbles. They have got a lot of marbles with numbers written on them. At the beginning, Raju would place the marbles one after another in ascending order of the numbers written on them. Then Meena would ask Raju to find the first marble with a certain number. She would count 1...2...3. Raju gets a point for a correct answer, and Meena gets the point if Raju fails. After some fixed number of trials the game ends and the player with maximum points wins. Today it's your chance to play as Raju. Being the smart kid, you'd be taking the favor of a computer. But don't underestimate Meena, she had written a program to keep track of how much time you're taking to give all the answers. So now you have to write a program, which will help you in your role as Raju.

**Input**

There can be multiple test cases. Total no of test cases is less than 65. Each test case consists begins with 2 integers: $N$ the number of marbles and $Q$ the number of queries Mina would make. The next $N$ lines would contain the numbers written on the $N$ marbles. These marble numbers will not come in any particular order. Following $Q$ lines will have $Q$ queries. Be assured, none of the input numbers are greater than 10000 and none of them are negative.

Input is terminated by a test case where $N = 0$ and $Q = 0$.

**Output**

For each test case output the serial number of the case.

For each of the queries, print one line of output. The format of this line will depend upon whether or not the query number is written upon any of the marbles. The two different formats are described below:

- '$x$ `found at` $y$', if the first marble with number $x$ was found at position $y$. Positions are numbered 1, 2, ..., $N$.
- '$x$ `not found`', if the marble with number $x$ is not present.

Look at the output for sample input for details.

| Sample Input | Sample Output |
|---|---|
| 4 1<br>2<br>3<br>5<br>1<br>5<br>5 2<br>1<br>3<br>3<br>3<br>1<br>2<br>3<br>0 0 | CASE# 1:<br>5 found at 4<br>CASE# 2:<br>2 not found<br>3 found at 3 |

📝 **Discussion**

Let us use the given in Case #2. The marbles are numbered 1, 3, 3, 3, and 1. Since Raju arranges them in ascending order, the ordered list of numbers is [1, 1, 3, 3, 3]. The problem asks for the one-based index of the first marble numbered $x$. The key idea is to recognize that this task is equivalent to getting the lower bound (and adding one since Meena starts counting at 1).

The slightly tricky part is deducing that an element is not in the list. We could perform the usual binary search and check if its return value is `NOT_FOUND`, but, since we are already employing binary search to solve for the lower bound, it would be felicitous (and elegant) to leverage the lower bound itself. Going through some examples may spark ideas:

- Suppose that we would like to get the lower bound of 2. The earliest zero-based position where it can be inserted without destroying the ordering is 2. However, the number at index 2 is 3. To generalize, if the lower bound is $y$ but the value at index $y$ is not equal to $x$, then $x$ is not in the list.

- Suppose that we would like to get the lower bound of 10. The earliest zero-based position where it can be inserted without destroying the ordering is 5, i.e., at the end of the list. However, trying to get the number at index 5 is an out-of-bounds memory access. To generalize, if the lower bound is $y$ but accessing the value at index $y$ is illegal (i.e., $y$ is the length of the list), then $x$ is not in the list.

The core function can, thus, be implemented as follows:

```
def solve(marble_numbers, queries):
    marble_numbers.sort()

    for query in queries:
        pos = lower_bound(marble_numbers, query)

        if pos == len(marble_numbers) or marble_numbers[pos] != query:
            print(query, "not found")
        else:
            print(query, "found at", pos + 1)
```

Note that the order of the expressions in the conditional

```
pos == len(marble_numbers) or marble_numbers[pos] != query
```

is important. If the order were to be reversed and `pos` turns out to be equal to the length of the list, `marble_numbers[pos]` would be an out-of-bounds memory access, halting the execution of the program. The way we wrote our conditional prevents this issue by taking advantage of **short-circuit Boolean evaluation**[5].

In brief, if there is a risk of an out-of-bounds memory access, it is imperative to check the index first before using it to retrieve a list element.

### 💭 Breakpoint B

1. Read about Python's `bisect` module. Are there library methods for getting the upper and lower bounds?
2. After experimenting with the `bisect` module, it may be insightful to take a look at its source code. How do its implementations of the methods for obtaining the upper and lower bounds differ from ours? What do you think are the motivations for these differences?

[5] For a discussion of short-circuit evaluation and how this is exploited in so-called "guardian patterns," you may refer to this text.

# BISECTION METHOD

So far, we have only discussed binary search in the context of discrete spaces, but you may have realized that the overarching principle can be applied to continuous spaces as well. We will go over the details as we dive straight into a contest problem.

## Illustrative Example Bootstrapping Number

*IDI Open 2021 (Norwegian University of Science and Technology)*

Neelix has been working on making the Altonian warp bubble power itself up. While waiting for the system checks to complete, he saw a number on his screen and wondered if there is a number that powers itself up to this number.

### Input

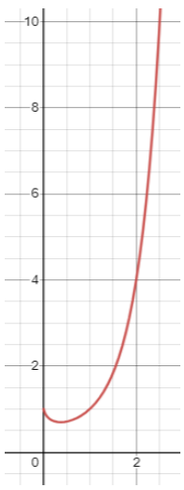
The first and only input line consists of a single integer $y$, the number Neelix saw on his screen. *(Note: The variable was renamed for pedagogy)*

### Limits

$1 \leq y \leq 10\,000\,000\,000$

| Sample Input | Sample Output |
|---|---|
| `4` | `2.0` |
| `42` | `3.20720196137` |

### 📝 Discussion




The problem is equivalent to finding the value of $x$ such that $x^x$ is equal to the input $y$. Let us begin our investigation by graphing the equation $y = x^x$. As seen in the figure, the graph suggests that $y = x^x$ is increasing in our interval of interest; this can be proven using calculus (although, for most intents and purposes, there is no need for such rigor).

Following the first sample input, suppose we want to find $x$ such that $x^x = 4$. We can take an initial guess and adjust it accordingly by exploiting the fact that, as $x$ increases, $y$ increases — or, as $x$ decreases, $y$ decreases.

The binary search principle makes this guessing-and-adjusting process more informed. The lowest value that $x$ can take is 1 (since $1^1 = 1$) while the highest possible value is 10 (since $10^{10} = 10\,000\,000\,000$). We set our initial guess to the "midpoint," i.e., $\frac{1+10}{2} = 5.5$, but, because $5.5^{5.5} \approx 11803.065 > 4$, we have to decrease our guess. Using the vocabulary of divide-and-conquer, we search the "lower half" and discard the "upper half" of the domain. The table below traces the first few iterations (for simplicity, let `low` and `high` be the endpoints of the search space, and let `mid` be the midpoint):

| Iteration # | low | high | mid | $\texttt{mid}^{\texttt{mid}}$ | Decision |
|---|---|---|---|---|---|
| 1 | 1 | 10 | 5.5 | ~ 11803.065 | Search lower half |
| 2 | 1 | 5.5 | 3.25 | ~ 46.092 | Search lower half |
| 3 | 1 | 3.25 | 2.125 | ~ 4.962 | Search lower half |
| 4 | 1 | 2.125 | 1.5625 | ~ 2.008 | Search upper half |
| 5 | 1.5625 | 2.125 | 1.84375 | ~ 3.089 | Search upper half |
| 6 | 1.84375 | 2.125 | 1.984375 | ~ 3.896 | Search upper half |
| ... | ... | ... | ... | ... | ... |

The next stumbling block is the terminating condition: When do we stop iterating? Waiting for $\texttt{mid}^{\texttt{mid}}$ to be equal to $y$ might seem to be an appealing suggestion, but this approach will not work if the roots are not rational, as in the second sample input ($x^x = 42$). Moreover, directly checking for equality between floating-point numbers should never be performed in light of floating-point precision errors[6].

Instead, we have two options:

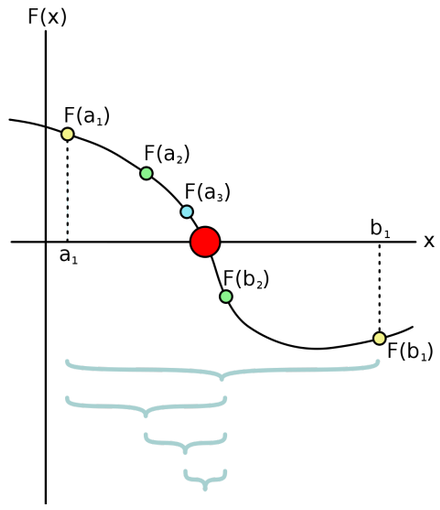
a) Set the number of iterations in advance.

b) Set a certain threshold or tolerance for error. Keep iterating while `high - low` > $\epsilon$, where $\epsilon$ is a very small value ($\epsilon$ is the Greek letter epsilon, which is conventionally used in math to denote extremely small values). Halim and Halim[7] suggest setting $\epsilon$ to $10^{-9}$ as a good rule of thumb.

## IMPLEMENTATION AND THEORY

Using the second option (i.e., via setting a tolerance for error), the core function to solve Bootstrapping Number via the bisection method can, thus, be implemented as follows:

```
def solve(y):
    EPSILON = 1e-9
    low = 1
    high = 10

    while high - low > EPSILON:
        mid = (high + low) / 2
        if mid ** mid > y:
            high = mid
        else:
            low = mid

    return mid
```




https://upload.wikimedia.org/wikipedia/commons/thumb/8/8c/Bisection_method.svg/1200px-Bisection_method.svg.png

[6] For a discussion of floating-point precision errors, you may refer to the Python documentation.
[7] Halim, S., & Halim, F. (2010). *Competitive programming 3: The new lower bound of programming contests.* National University of Singapore.

### Breakpoint C

1. There are equations with no real roots, such as $x^2 = -4$. How would you modify our implementation of the bisection method to catch these cases?
2. How would you modify our implementation if the function is decreasing (instead of increasing)?
3. What is the disadvantage of setting $\epsilon$ to a value that is "not small enough"? What is the downside of setting it to a value that is "too small"?

The approach demonstrated is known as the **bisection method**, also referred to as the **dichotomy method** or the **interval halving method** — the application of the binary search principle to continuous search spaces. Underlying this numerical approximation method is the **intermediate value theorem** from calculus[8].

Analogous to how binary search requires the list to be sorted, the search space has to be **monotonic**, i.e., either increasing or decreasing. Monotonicity is rigorously proven using calculus, but, in competitive programming, heuristics (e.g., graphing the function or trying out a series of values) usually suffice. Likewise, quick scripts can be written to determine the endpoints of the domain based on the problem constraints or limits.

The bisection method has a space complexity of $\Theta(1)$. Given a search space with endpoints $a$ and $b$ and tolerance $\epsilon$, it takes $O\left(\log \frac{b-a}{\epsilon}\right)$ iterations to converge to an answer. Knowing this is useful for carrying out ballpark estimates of the program's efficiency; for instance, in our illustrative example, the answer can be obtained in roughly $\log_2\left(\frac{10-1}{10^{-9}}\right) \approx 33$ iterations (or "guesses"). In contrast, performing something akin to a linear search will take, at worst, $\frac{10-1}{10^{-9}} \approx 9 \times 10^9$ iterations. The difference is beyond palpable!

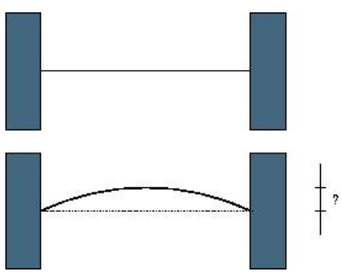
Let us now take a look at some examples of how the bisection method can be applied to an exciting selection of **interdisciplinary programming problems** that appeared in recent competitions.

## Geometry Dash Expanding Rods

*June 2004 University of Waterloo/Alberta Programming Contest*

When a thin rod of length $L$ is heated $n$ degrees, it expands to a new length $L' = (1 + n \cdot C) \cdot L$, where $C$ is the coefficient of heat expansion.

When a thin rod is mounted on two solid walls and then heated, it expands and takes the shape of a circular segment, the original rod being the chord of the segment.




Your task is to compute the distance by which the center of the rod is displaced.

[8] For a more mathematical treatment of the bisection method from the perspective of numerical analysis, you may refer to this slide deck.

### Input

The input contains at most 20 lines. Each line of input contains three non-negative numbers:

- An integer $L$, the initial length of the rod in millimeters ($1 \leq L \leq 10^9$),
- An integer $n$, the temperature change in degrees ($0 \leq n \leq 10^5$),
- A real number $C$, the coefficient of heat expansion of the material $0 \leq C \leq 100$, at most 5 digits after the decimal point).

The input is such that the displacement of the center of any rod is at most one half of the original rod length. The last line of input contains three –1's and it should not be processed.

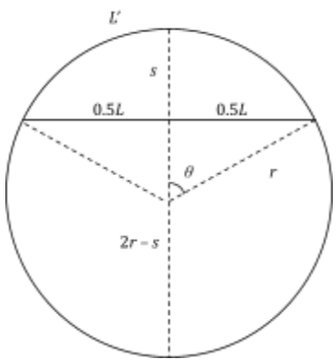
### Output

For each line of input, output one line with the displacement of the center of the rod in millimeters with an absolute error of at most $10^{-3}$ or a relative error of at most $10^{-9}$.

| Sample Input | Sample Output |
|---|---|
| 1000 100 0.0001<br>15000 10 0.00006<br>10 0 0.001<br>-1 -1 -1 | 61.328991534<br>225.020248568<br>0.000000000 |

### 📝 Discussion

The crux of the problem is deriving an equation that relates the displacement of the center of the rod to the given quantities. Let $s$ be the displacement and $r$ be the radius of the circle. By the intersecting chords theorem, $s(2r - s) = (0.5L)^2$. This implies that $r = \frac{0.25L^2+s^2}{2s} = \frac{L^2}{8s} + \frac{s}{2}$.




The next task is relating the radius $r$ and the arc length $L'$; this can be done via the arc length formula and a bit of trigonometry. Let $\theta$ be the measure of the angle (in radians) shown in the adjoining figure. Since $L' = 2r\theta$ and $\sin\theta = \frac{0.5L}{r} = \frac{L}{2r}$, $L' = 2r \arcsin\left(\frac{L}{2r}\right)$.

Summarizing what we know so far:

a) It is given that $L' = (1 + n \cdot C) \cdot L$.

b) We have derived that $L' = 2r \arcsin\left(\frac{L}{2r}\right)$, where $r = \frac{L^2}{8s} + \frac{s}{2}$.

Theoretically, we can equate these two equations and isolate $s$, but a cursory look shows that this will only introduce unnecessary complications. To organize our solution, let us create two separate functions corresponding to these two equations for $L'$:

```
def getLPrimeByHeatCoeff(L, n, c):
    return (1 + n * c) * L
```

```
def getLPrimeByDisplacement(L, s):
    r = (L ** 2) / (8 * s) + (s / 2)
    return 2 * r * math.asin(L / (2 * r))
```

We are now ready to apply the bisection method. The problem's constraints guarantee that the displacement is at most half of the original rod length:

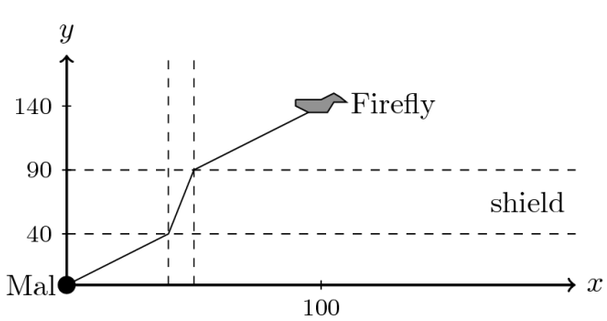
```
def solve(L, n, c):
    EPSILON = 1e-9
    low = 0
    high = L / 2

    while high - low > EPSILON:
        mid = (high + low) / 2
        if getLPrimeByDisplacement(L, mid) > getLPrimeByHeatCoeff(L, n, c):
            high = mid
        else:
            low = mid

    return mid
```

## Physics Simulation Careful Ascent

*2016 ACM-ICPC Northwestern Europe Regional Contest*
*(Association for Computing Machinery – International Collegiate Programming Contest)*

That went well! As police sirens rang out around the palace, Mal Reynolds had already reached his lifting device outside of the city.

No spaceship can escape Planet Zarzos without permission from the High Priest. However, Mal's spaceship, Firefly, is in geostationary orbit well above the controlled zone and his small lifting device can avoid being recognised as an intruder if its vertical velocity is exactly 1 km/min.




There are still two problems. First, Mal will not be able to control the vehicle from his space suit, so he must set up the autopilot while on the ground. The vertical velocity must be exactly 1 km/min and the horizontal velocity must be set in such a way that Mal will hit the Firefly on the resulting trajectory. Second, the energy shields of the planet disturb the autopilot: They will decrease or increase the horizontal velocity by a given factor. The original horizontal velocity is restored as soon as there is no interference. For this problem we consider Firefly to be a single point — the shape shown in the figure is merely for decorative purposes.

**Input**

The input consists of:

- One line with two integers $x$, $y$ ($-10^7 \le x \le 10^7$, $|x| \le y \le 10^8$ and $1 \le y$), Firefly's coordinates relative to Mal's current position (in kilometres).
- One line with an integer $n$ ($0 \le n \le 100$), the number of shields.
- $n$ lines describing the $n$ shields, the $i^{\text{th}}$ line containing three numbers:
  - An integer $l_i$ ($0 \le l_i < y$), the lower boundary of shield $i$ (in kilometres).
  - An integer $u_i$ ($l_i < u_i \le y$, the upper boundary of shield $i$ (in kilometres).
  - A real value $f_i$ ($0.1 \le f_i \le 10.0$), the factor with which the horizontal velocity is multiplied during the traversal of shield $i$.

  It is guaranteed that shield ranges do not intersect, i.e., for every pair of shields $i \ne j$ either $u_i \le l_j$ or $u_j \le l_i$ must hold.

All real numbers will have at most 10 digits after the decimal point.

**Output**

Output the horizontal velocity in km/min which Mal must choose in order to reach Firefly. The output must be accurate to an absolute or relative error of at most $10^{-6}$.

| Sample Input | Sample Output |
|---|---|
| 100 140<br>1<br>40 90 0.2000000000 | 1.0 |
| 100 100<br>3<br>0 20 2.0000000000<br>50 100 0.1000000000<br>20 50 0.2000000000 | 1.96078431373 |

📝 **Discussion**

Our strategy centers on simulating Mal's careful ascent — but, first and foremost, we have to find a way to express the horizontal velocity in terms of the given variables. Suppose the angle of motion is $\theta$ and the destination is at $(x_{dest}, y_{dest})$.

Recalling the principles of two-dimensional kinematics, the components of the velocity $v$ can be expressed as:

a) Horizontal Velocity: $v_x = v\cos\theta$

b) Vertical Velocity: $v_y = v\sin\theta$

It follows that $\frac{v_x}{v_y} = \cot\theta$. Since $v_y = 1$, $v_x = \cot\theta = \frac{x_{dest}}{y_{dest}}$, implying that $x_{dest} = y_{dest} v_x$. The key idea is that we are going to guess a value for $v_x$ and calculate the resulting $x_{dest}$ (although this is not that straightforward due to the presence of shields). If Mal's final $x$-coordinate is less than that of Firefly's, we increase our horizontal velocity; otherwise, we decrease it.

The crux of the problem is now accounting for the effects of the shields. Suppose the height of shield $i$ is $h_i = u_i - l_i$. The modified horizontal velocity is $f_i v_x$. Consequently, Mal's $x$-coordinate after passing through shield $i$ is $h_i f_i v_x$. After passing through all the shields (and nothing but shields), Mal's $x$-coordinate is now $\Sigma\, h_i f_i v_x$.

Since the total height of the remaining zones outside the interference of shields is $y - \Sigma\, h_i$ (note that $y$ is Firefly's $y$-coordinate), Mal's final $x$-coordinate is $\Sigma\, h_i f_i v_x + (y - \Sigma\, h_i) v_x$. Armed with these insights, it is now possible to implement the core functions.

```
def getMalFinalX(l_shields, u_shields, f_shields, y_firefly, v_x):
    malFinalX = 0
    y = y_firefly

    for i in range(0, len(l_shields)):
        h_shield = u_shields[i] - l_shields[i]
        malFinalX += h_shield * f_shields[i] * v_x

        y -= h_shield

    malFinalX += y * v_x
    return malFinalX


def solve(l_shields, u_shields, f_shields, x_firefly, y_firefly):
    EPSILON = 1e-9
    low = -1 * y_firefly
    high = y_firefly

    while high - low > EPSILON:
        mid = (high + low) / 2
        if (getMalFinalX(l_shields, u_shields, f_shields, y_firefly, mid) >
        x_firefly):
            high = mid
        else:
            low = mid

    return mid
```

It is actually possible to solve this problem without using the bisection method. This more efficient approach is left as an exercise for the reader.

# 👩‍💻 EXERCISES

## Binary Search

1. Room Painting (October 2012 University of Waterloo Programming Contest)
2. Exact Sum (UVa Online Judge 11057)
3. Out of Sorts (2019 ICPC East-Central North America Regional Contest)
4. The Stern-Brocot Number System (UVa Online Judge 10077)

## Bisection Method

5. Solve It (UVa Online Judge 10341)
6. Electric Bill (UVa Online Judge 12190)
7. Traveling Monk (2017 Virginia Tech High School Programming Contest)
8. Leaps Tall Buildings (in a single bound) (UVa Online Judge 10372)

The exercises were selected from the list of practice problems suggested by S. Halim and F. Halim in *Competitive Programming 3: The New Lower Bound of Programming Contests.*

```
-- return 0; --
```