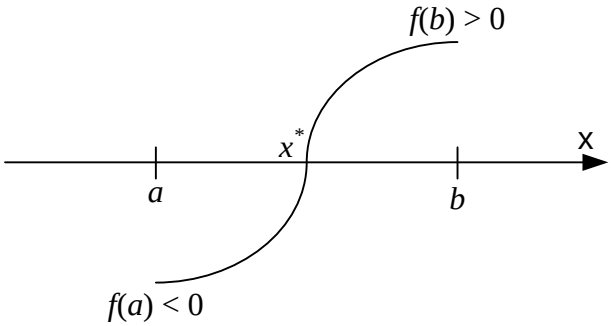
Binary search

Dichotomy is a method for numerically solving equations in a single unknown. Consider the equation f(x) = 0 with a continuous function ***f*** on the interval [a; b] which takes values of **different signs** at the end points of the interval and which has a single root x^* within [a; b].



To find x^* approximately, one divides [a; b] into halves and calculates the value of f(x_1) at the midpoint $x_1 = (a + b) / 2$. If f(x_1) ≠ 0, one takes the two intervals [a; x_1] and [x_1; b] and from them selects for the next dichotomy the one at the end points of which the values of the function **differ in sign**. This continued division into halves gives a sequence x_1, x_2, … which converges to the root x^* with the rate of a geometrical progression:

$$| x_n - x^* | \le (b - a) / 2^n, n = 1, 2, \ldots$$

E-OLYMP **3968. Square root** Find such number x that x^2 + SQRT(x) = C with no less than 6 digits after the decimal point. SQRT(x) is a square root of number x.

► Consider a function f(x) = x^2 + SQRT(x).

Its obvious that f(0) = 0, f(C) = C^2 + SQRT(C) > C. That is, the root of the equation

$$f(x) = C$$

lies on the interval [0; C]. Function f(x) is strictly increasing. Search for the root x with a ***binary search***.

Declare a constant EPS and function f(x).

```
#define EPS 1e-10

double f(double x)
{
  return x * x + sqrt(x);
}
```

The main part of the program. Read the value of C.

```
scanf("%lf",&c);
```

Set the boundaries of binary search [*left*; *right*] = [0; C].

```
left = 0; right = c;
```

Run the binary search.

```
while(right - left > EPS)
{
  middle = (left + right)/ 2;
  y = f(middle);
  if (y > c) right = middle; else left = middle;
}
```

Print the answer.

```
printf("%lf\n",left);
```

E-OLYMP **4420. The root of a cubic equation** Given a cubic equation $ax^3 + bx^2 + cx + d = 0$ ($a \neq 0$). It is known that this equation has exactly one root. Find it.

► Localize the root of equalization f(x) = 0. To do this, find r such that f(-r) * f(r) < 0. For example, having initialized r = 1, we will double r at each step until holds an inequality f(-r) * f(r) < 0. Thus, we will iterate over intervals [-1; 12; 2], [-4; 4], [-8; 8], …. until we find the interval where the root of the equation is located.

Set l = -r. Further on the interval [l; r] using a binary search (dividing a segment in half), we look for a root.

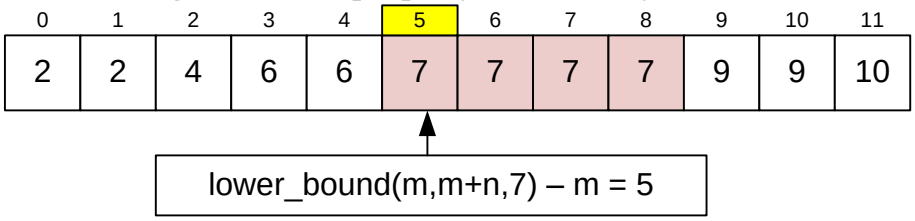
Discrete binary search

E-OLYMP **9016. Binary search** Sorted array of n integers is given. You must answer q queries: whether the given number x is in the array.

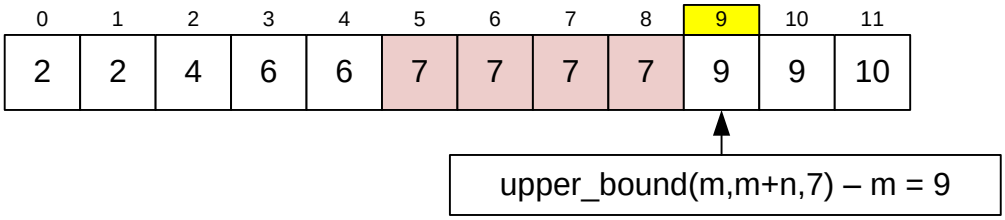
► In the problem we must find a number in a sorted array. This can be done using a binary search.

Let m be a sorted array of length n consisting of integers. Function ***binary_search***(m, $m + n$, x) returns ***true*** if number x is present in the array. The input array is read in O(n), q queries are executed in O($q\log_2 n$).

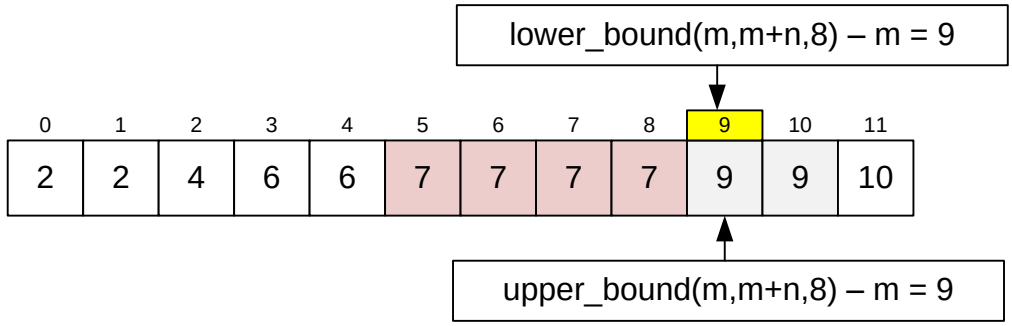
Function ***lower_bound*** returns a pointer to the first position where element x can be inserted without violating the sorted property of the array.



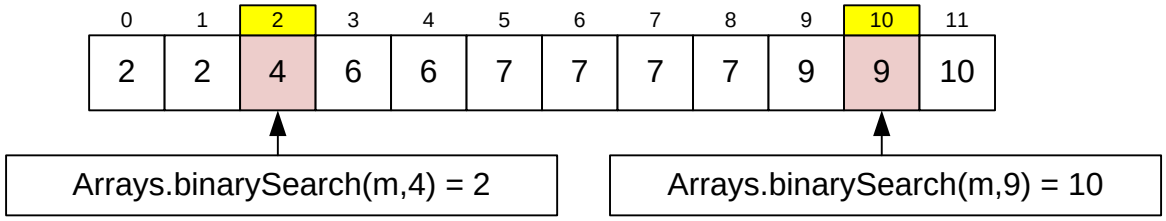
Function ***upper_bound*** returns a pointer to the last position where element x can be inserted without violating the array's sorted property.



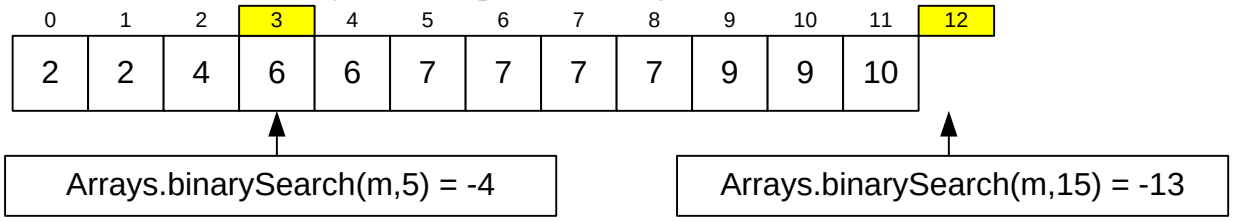
If ***lower_bound***(*m*, *m* + *n*, *x*) ≠ ***upper_bound***(*m*, *m* + *n*, *x*), then number *x* is present in array. Otherwise, there is no number *x* in the array.



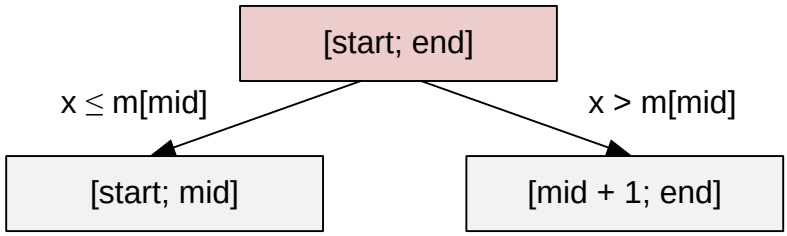
Java function ***Arrays.binarySearch***() returns the index of the element found in the array. If array contains multiple keys, then any of its indices are returned.



If *x* is not present in array m, then function ***Arrays.binarySearch***(m, *x*) returns value –*pos* – 1, where *pos* is the index of the first element greater than *x*. If *x* is greater than all the elements in array m, then *pos* = *m.length*.



Implementation of own binary search. Let the element *x* in the array m be searched for in the interval [*start*; *end*]. Divide the segment in half, put *mid* = (*start* + *end*) / 2. If *x* > m[*mid*], then *x* lies on the segment [*mid* + 1; *end*]. Otherwise, the search should be continued on the segment [*start*; *mid*].



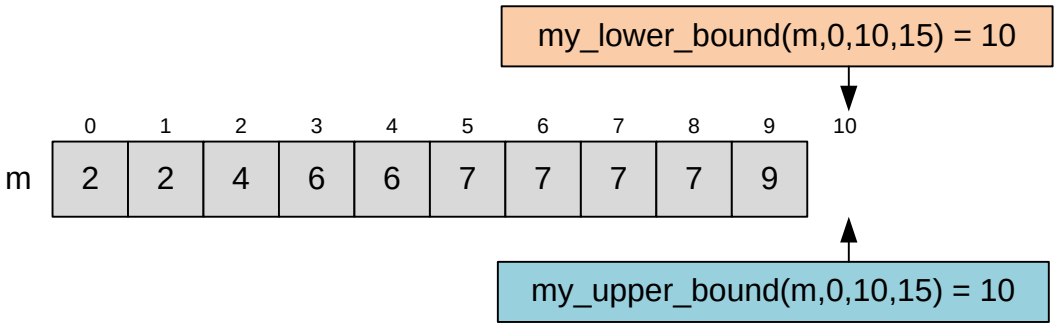

```
int my_bin_search(int *m, int start, int end, int x)
{
  while (start < end)
  {
    int mid = (start + end) / 2;
    if (x > m[mid])
```

```
      start = mid + 1;
    else
      end = mid;
  }
  return m[start] == x;
}
```

E-OLYMP **9017. Binary search - 1** Sorted array of n integers is given. You must answer q queries: how many times the given number x appears in the array.

► The number of times x occurs in the sorted array is ***upper_bound***$(m, m + n, x)$ – ***lower_bound***$(m, m + n, x)$. These functions can be taken from the STL template library or implemented independently (for **Java**).

Let all n elements of array m be in cells from 0 to $n - 1$. Then the ***lower_bound*** and ***upper_bound*** can return values from 0 to n. Therefore, binary search should be performed within these limits.




```
int my_lower_bound(int *m, int start, int end, int x)
{
  while (start < end)
  {
    int mid = (start + end) / 2;
    if (x <= m[mid])
      end = mid;
    else
      start = mid + 1;
  }
  return start;
}

int my_upper_bound(int *m, int start, int end, int x)
{
  while (start < end)
  {
    int mid = (start + end) / 2;
    if (x >= m[mid])
      start = mid + 1;
    else
      end = mid;
  }
  return start;
}
```

E-OLYMP **9018. n-th integer divisible by a or b** Given two integers a and b. Find the n-th positive integer that is divisible by either a or b.

► Let function f(n) computes the number of positive integers in the interval [1; n], divisible by either a or b. This number is

$$n / a + n / b – n / \text{LCM}(a, b)$$

Let x be the n-th number. Then x must be the ***smallest positive integer*** such that f(x) = n. We will search for it with a binary search, starting from the segment [*left*; *right*] = [1; 10^9]. Let *mid* = (*left* + *right*) / 2. If f(*mid*) ≥ n, then the search should be continued on the segment [*left*; *mid*], otherwise on the segment [*mid* + 1; *right*].

	1		2	3	4		5		6		7		8	9	10
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Consider the first sample, where a = 2, b = 5. It is necessary to find the smallest x for which f(x) = 10. Let's calculate some values:

- f(15) = 15 / 2 + 15 / 5 – 15 / 10 = 7 + 3 – 1 = 9;
- f(16) = 16 / 2 + 16 / 5 – 16 / 10 = 8 + 3 – 1 = 10;
- f(17) = 17 / 2 + 17 / 5 – 17 / 10 = 8 + 3 – 1 = 10;
- f(18) = 18 / 2 + 18 / 5 – 18 / 10 = 9 + 3 – 1 = 11;

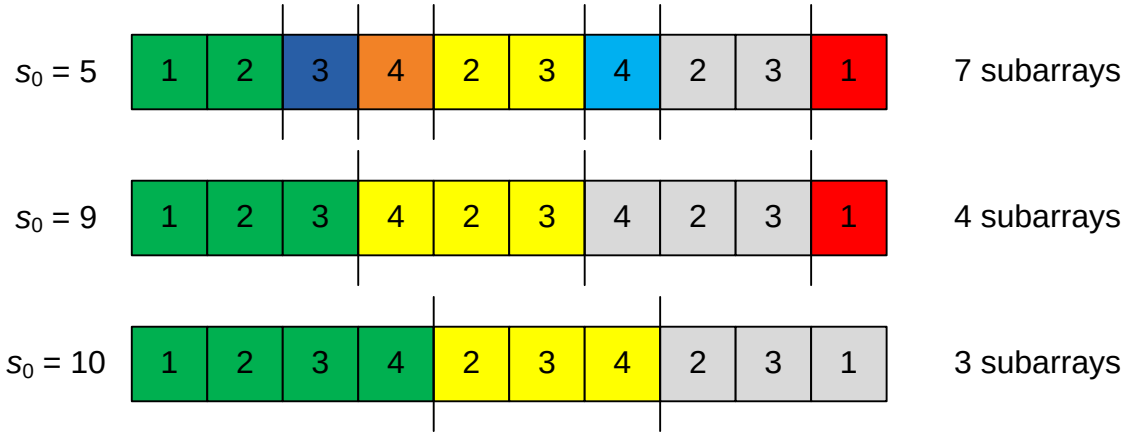
The smallest solution to the equation f(x) = 10 is x = 16.

E-OLYMP **9020. Split into k sub-arrays** Given array of n elements and number k. Split the given array into k sub-arrays (they must cover all the elements). The maximum sub-array sum achievable out of k sub-arrays formed, must be minimum possible. Find that possible sub-array sum.

► The answer can be found by the binary search. Obviously, it is not less than 1 (the integers in array are positive) and no more than the sum of all numbers. Set *left* = 1, *right* = the sum of the numbers in the array. Then the minimum possible sub-array sum lies on the interval [*left*; *right*].

Consider a subproblem: is it possible to split an array into k subarrays so that the maximum of the sums of all k subarrays will be no more than s_0. If array contains at least one element greater than s_0, then such partition is impossible. Split the original array into subarrays, the sum of which elements is no more than s_0. If there are no more than k such subarrays, the split is possible.

Consider in the second example the required partitions into subarrays for different values of s_0:



E-OLYMP **9021. Count the integers of the form 2^x * 3^y** Find the number of integers from the range [*a*, *b*] that can be represented as $2^x * 3^y$ ($x \geq 0, y \geq 0$).

► The amount of required numbers f(*a*, *b*) on a segment [*a*, *b*] is f(0, *b*) – f(0, *a* – 1). The query f(0, *q*) means the amount of numbers of the form $2^x * 3^y$, not greater than *q*. We look for the answer using a ***binary search*** on array v.

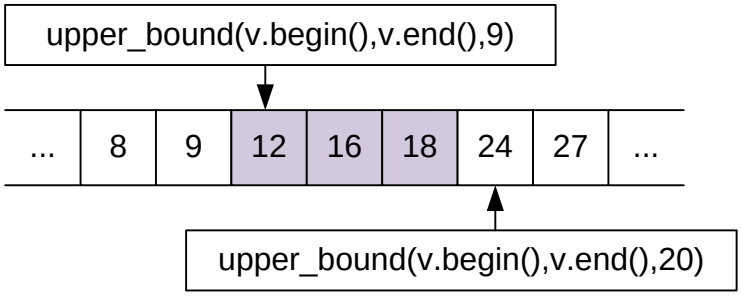
Generate all numbers of the form $2^x * 3^y$, no more than 200.

y	0	1	2	3	4
x = 0	1	3	9	27	81
x = 1	2	6	18	54	162
x = 2	4	12	36	108	
x = 3	8	24	72		
x = 4	16	48	144		
x = 5	32	96			
x = 6	64	192			
x = 7	128				

$2^x * 3^y$

Segment [1; 10] contains 7 numbers (highlighted in blue).
Segment [100; 200] contains 5 numbers (highlighted in green).

Find a solution for segment [10; 20]:
upper_bound(v.***begin***(), v.***end***(), 20) – ***upper_bound***(v.***begin***(), v.***end***(), 9) = 3

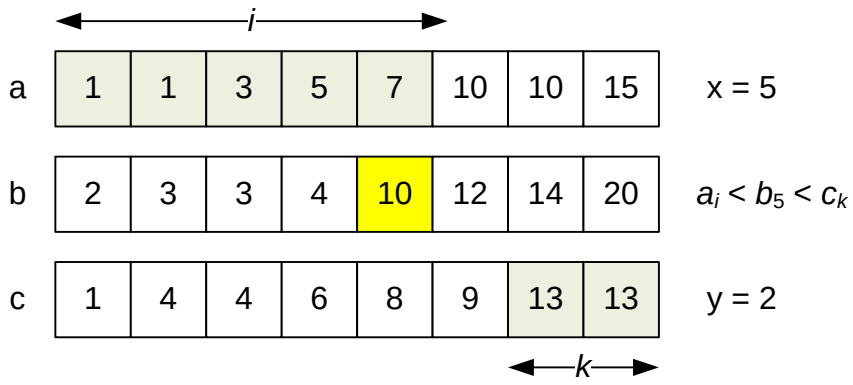


Indeed, segment [10; 20] contains 3 numbers of required form:
12, 16, 18

E-OLYMP **9022. Count the tripples** Given three array *a*, *b* and *c* of *n* integers each. Find the number of triplets (a_i, b_j, c_k) such that $a_i < b_j < c_k$.

► Sort the arrays. For each value of b_j, using a binary search, find the amount of numbers x from array a that are less than b_j, as well as the amount of numbers y from array c that are greater than b_j. Then, for a fixed value of b_j, there are $x * y$ desired triples (a_i, b_j, c_k).

Consider the following sorted arrays. Let us calculate the number of required triples, in which $b_5 = 10$. We have: $a_i < b_5$ for $i \le 5$, $c_k > b_5$ for $k \ge 7$. That is, the inequality $a_i < b_5 < c_k$ holds for $1 \le i \le 5$ and $7 \le k \le 8$. The number of triplets (a_i, b_5, c_k) is 5 * 2 = 10.



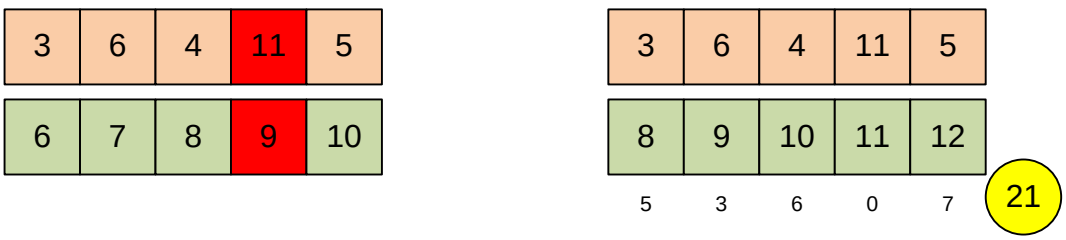
E-OLYMP **9023. Minimum increments** Given array with n positive integers. Find the minimum number of operations required so that an Arithmetic Progression with the array elements is achieved with common difference as 1. In a single operation, any element can be incremented by 1.

► Let the required least number of operations transform the original array m[0], m[1], …, m[n – 1] into x, x + 1, x + 2, …, x + n – 1. Such the smallest value of x will be found by binary search. Obviously, that m[0] ≤ x ≤ max(m[0], m[1], …, m[n – 1]).

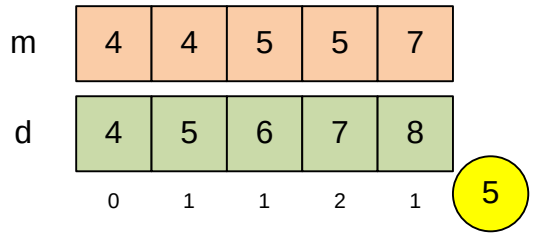
A sequence d = [x, x + 1, x + 2, …, x + n – 1] is called ***suitable*** if array m can be transformed into array d (for this, m[i] ≤ d[i] must hold for all i from 0 up to n – 1).

Consider the first test case. Let d = [x, x + 1, x + 2, …, x + n – 1] be an appropriate sequence with the minimum sum of elements. Obviously, m[0] = 3 ≤ x ≤ 11 = max(m[i]). Using binary search, we look for the smallest x for which the sequence d is suitable.

For example, for x = 6 the sequence d will not be suitable, but for x = 8 it will.



Consider the second test. Start the binary search on the interval 4 ≤ x ≤ 7. For example, for x = 4, the sequence d is already suitable.



E-OLYMP **1302. Almost prime numbers** Almost prime numbers are the non-prime numbers which are divisible by only a single prime number. In this problem your job is to write a program which finds out the number of almost prime numbers within a certain range.

► *Almost prime* numbers have the form p^k, where p is a prime number, $k \geq 2$. For example, the almost prime numbers are 4, 8, 16, 32, 9, 81, … . As $high < 10^{12}$, then from the definition of almost prime number $p < 10^6$. Generate the array *primes* if length $1000000 = 10^6$, where *primes*[*i*] = 1 if *i* is prime and primes[*i*] = 0 otherwise. Then generate and sort in array *m* all the almost primes (their amount equals to 80070).

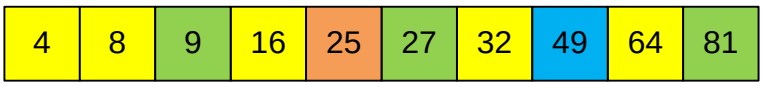
Let f(*a*, *b*) be the number of *almost primes* in the range [*a*..*b*]. Then

$$f(low, high) = f(1, high) - f(1, low - 1)$$

The number of almost primes in the range [1 .. *n*] equals to the position (index), where it is possible to insert *n* into the sorted array *m* by upper bound preserving the sorted order. The index number can be found using binary search on array *m* by means of function ***upper_bound***.

Let's generate all the almost prime numbers in the range from 1 to 100. First write the powers of 2 not greater than 100. Then the powers of 3, 5 and 7. The square of 11 is greater than 100, so there are no powers of 11 among the almost primes in the range [1..100].

Sort the array:



Consider the second test case. In the range from 1 to 20 there are 4 almost prime numbers: 4, 8, 9, 16.

E-OLYMP **7329. Construction** After finishing his chalet construction, Stepan still has *n* wooden boards, with lengths l_1, ..., l_n. He decided to build a fishing deck on a lake using these boards.

Stepan believes that the longer deck is, the more fish he will catch!

Moreover, Stepan, as all fishermen, is very superstitious and trusts all omens. One of it – the deck can be build using whole boards only (boards can be cut but not joined). Now Stepan is interested what maximum deck length *d* can be obtained, if needs to use *m* boards.

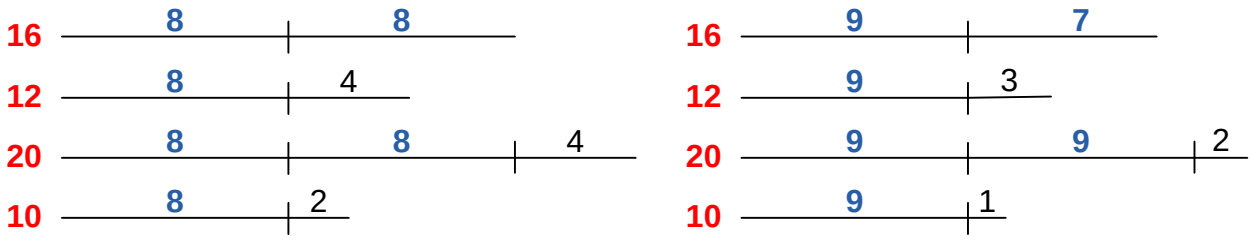
► Let's find the length of the deck *d* with binary search. Initially let $d \in [Left; Right] = [0; INF]$. Consider the next subtask: is it possible to construct the required

bridge with length exactly *Mid* = (*Left* + *Right*) / 2 ? To do this, count how many boards of length *Mid* we can get. From the board of length l_i we can get $\lfloor l_i / Mid \rfloor$ such boards. By cutting you can get

$$\sum_{i=1}^{n} \lfloor l_i / Mid \rfloor$$

boards of length *Mid*. If in total we can get *m* boards, set *Left* = *Mid* + 1. Otherwise set *Right* = *Mid*. The answer will be the value of *Left* – 1.

In the given sample there are 4 boards. It is necessary to build a deck with 6 boards. If the length of the deck will be 8, then the existing boards can be cut as follows, getting 16/8 + 12/8 + 20/8 + 10/8 = 6 boards of length 8.

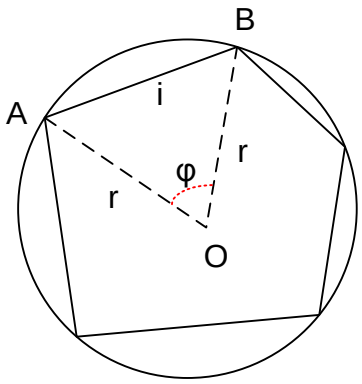


If the length of the deck will be 9, then we can get only 16/9 + 12/9 + 20/9 + 10/9 = 5 boards of length 9, which is less than the required 6.

Binary search in geometry

E-OLYMP **3532. Interesting polygon** Vasya drew a convex polygon with sides of length 1, 2, 3, ..., *n* (4 ≤ *n* ≤ 30), and then built the circle around it. Find the radius r of the circle drawn around the polygon by Vasya.

► We'll search for the radius by binary search. It's obvious that $r > n / 2$ and $r < n^2$.



Let the length of the side AB be equal to *i*. Write the cosine theorem for triangle AOB:

$$i^2 = r^2 + r^2 - 2 * r * r * \cos\varphi$$

Or the same as $i^2 = 2r^2 - 2r^2\cos\varphi$, whence $\cos\varphi = (2r^2 - i^2) / 2r^2$. Compute the sum of angles subtending the sides of the polygon with lengths 1, 2, 3,…, *n*. If the resulting sum of angles is greater than 2π, then the radius should be reduced. Otherwise, the

radius should be increased. Find the value of the radius, for example, with an accuracy of 10^{-11}.

Declare the constants.

```
#define EPS 1e-11
#define PI acos(-1.0)
```

Read the number of sides *n*. Set the boundaries of the binary search for the radius of the circumscribed circle [*Left* .. *Right*] = [*n* / 2 .. n^2].

```
scanf("%d",&n);
Left = n / 2; Right = n * n;
```

While the length of the segment [*Left* .. *Right*] is not less than 10^{-11}, continue the binary search.

```
while(fabs(Right - Left) > EPS)
{
```

Divide the segment [*Left* .. *Right*] in half with the *Middle* point. Assuming the radius of the circle to be *Middle*, calculate in the variable *fi* the sum of the angles that subtend the sides of the polygon.

```
  Middle = (Left + Right) / 2;
  for(fi = 0, i = 1; i <= n; i++)
  {
    double angle = (2*Middle*Middle - i*i)/(2*Middle*Middle);
```

If a triangle with sides *i*, *Middle*, *Middle* cannot be constructed, then the cosine value of the corresponding angle may go beyond the segment [-1, 1]. Therefore, it is necessary to reduce the value of the radius of the circumscribed circle.

```
    if ((angle < -1.0) || (angle > 1.0))
    {
      fi = 100; break;
    }
    fi += acos(angle);
  }
```

Depending on the found sum of angles *fi*, adjust the search segment.

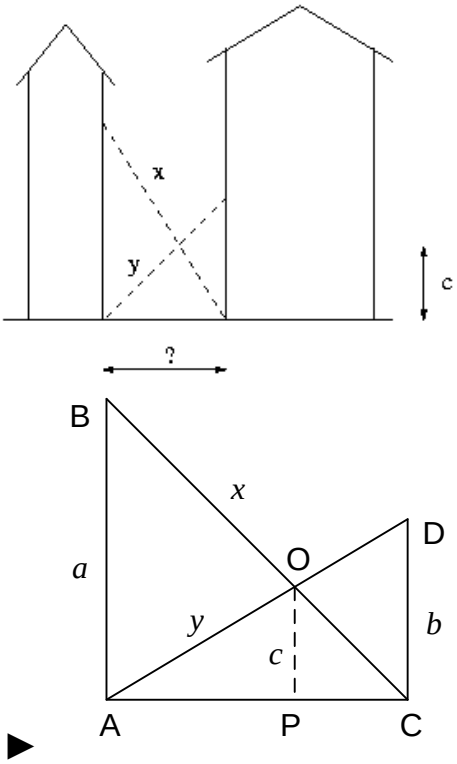
```
  if (fi > 2*PI) Left = Middle; else Right = Middle;
}
```

Print the answer.

```
printf("%.8lf\n",Left);
```

E-OLYMP 1506. Crossed ladders A narrow street is lined with tall buildings. An *x* foot long ladder is rested at the base of the building on the right side of the street and leans on the building on the left side. An *y* foot long ladder is rested at the base of the

building on the left side of the street and leans on the building on the right side. The point where the two ladders cross is exactly *c* feet from the ground. How wide is the street?



Triangles AOP and ADC are similar: $\frac{c}{b}=\frac{AP}{AC}$.

Triangles COP and CBA are similar: $\frac{c}{a}=\frac{CP}{CA}$.

$\frac{c}{b}+\frac{c}{a}=\frac{AP}{AC}+\frac{CP}{AC}=\frac{AC}{AC}=1$. Whence $c\left(\frac{1}{a}+\frac{1}{b}\right)=1,\ c=\frac{1}{\frac{1}{a}+\frac{1}{b}}=\frac{ab}{a+b}$.

We'll search for the required street width *d* = AC with ***binary search***.

Initially set $0 \le d \le \min(x, y)$. Given *d*, *x*, *y* find *a*, *b* and *c*. The larger *d* (with fixed *x*, *y*), the smaller *c*.

Read the input data for each test case.

```
while(scanf("%lf %lf %lf",&x,&y,&c) == 3)
{
```

Set *left* = 0, *right* = min(*x*,*y*). Further in the ***while*** loop, holds the inequality *left* ≤ *d* ≤ *right*.

```
    left = 0;
    if (x < y) right = x; else right = y;
    d = (left + right) / 2;
    do
    {
```

Compute the values of *a*, *b*, *c*.

```
a = sqrt(x*x - d*d); b = sqrt(y*y - d*d);
c1 = 1/(1/a + 1/b);
```

If the found value of *c*1 is less than the desired *c*, then it is necessary to reduce the upper bound *d*. Otherwise, you should increase its lower bound.

```
if (c1 < c) right = (left + right) / 2;
       else left = (left + right) / 2;
d = (left + right) / 2;
```

Carry out the calculations to the accuracy required in the problem statement (four decimal digits).

```
  } while (fabs(c1 - c) > 0.00001);
```

Print the answer.

```
  printf("%.3lf\n",d);
}
```