05. Binary Search

Binary Search

A basic technique to search something in a list of things.
You need a sorted array.

Problem.
a = [2, 5, 6, 10, 12, 18, 21]
x = 18. Find i such that a[i] = x

Maintain a segment of the array where we search for x. Maintain two pointers l and r with the following property: x belongs to a[l..r]

Inititally, l=0, r=n-1

Pick index m, at the middle of the segment. m = (l+r)/2

if a[m] < x, then all elements to the left of m are also less than x. So we search on the right half. Set l = m+1

else if a[m] > x, r = m-1, due to a similar reason.

else return m, as the element x as been found at position m

```
l=0, r=n-1
while r-l+1>=1 //While size of segment is greater or equal to 1
//Can also write 'while l<=r'
    if a[m]<x
        l=m+1
    else if a[m]>x
        r=m-1
    else
        return m
```

The above is not the best way to write binary search. Too many small details, such as m+1, m-1, 'while' condition is strange, there are three 'if' branches, if there are many elements = x, you might get any occurrence.

Moreover, the given problem can be solved using hashmaps. If we just need to find the value, we can just put elements in a hashmap and get the index of the element from the hashmap. Binary search isn't really needed.

Binary search can be used to solve more complicated problems.

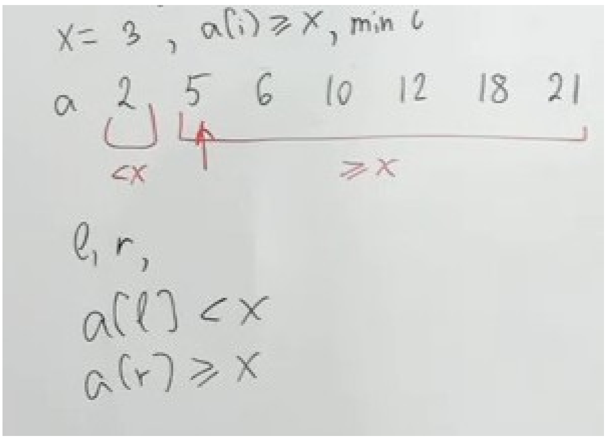
Problem.
a = [2, 5, 6, 10, 12, 18, 21]
x = 3. Find i such that a[i] >= x, with minimum i.

We can adjust the above algorithm to solve the problem, but it's slightly complicated, and we might make a bug.

Instead, let's write a new algorithm from scratch.

Maintain two pointers l, r, with the following property: a[l]<x, a[r]>=x

Initially, we can put the l and r pointers here.

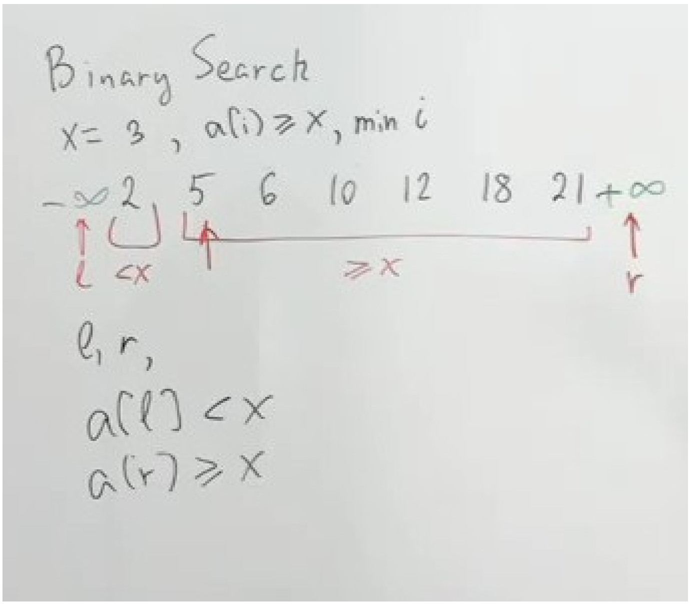


The leftmost position of the second segment (of pointer r) is what we're looking for.

Initially, we might think of putting l=0 and r=n-1. But

1. If all elements < x, then r=n-1 breaks the property.
2. If all elements >= x then l=0 breaks the property.

So we add element +INF after the end, and -INF before the beginning of the array, and make l and r point there.



While we can actually but a very big number there, here it's sufficient to imagine their existence. In other words, simply l=-1, r=n.

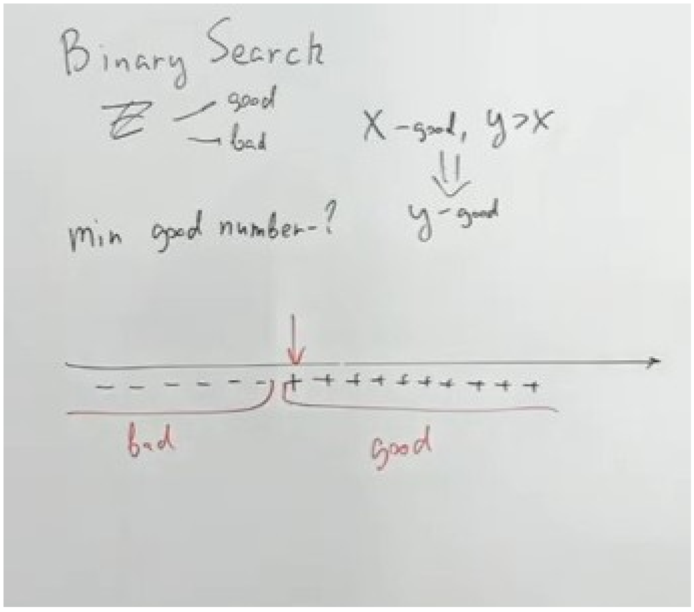
```
l=-1
r=n
while r>l+1 //we need to stop the loop at r=l+1
    m=(l+r)/2
    if a[m]>=x
        r=m
    else
        l=m
return r //r=n means no element >=x
```

Problem.
a = [2, 5, 6, 10, 12, 18, 21]
x = 15. Find i such that a[i] <= x, with maximum i.

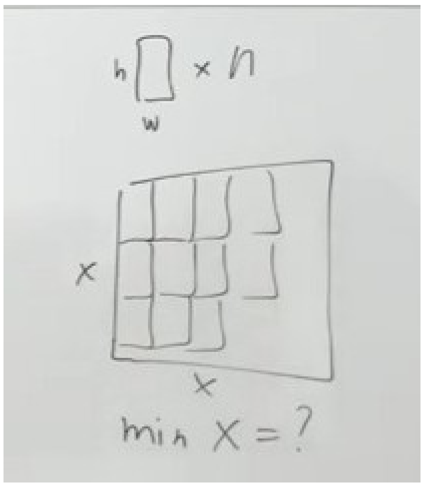
Then invariants: a[l]<=x and a[r]>x

```
l=-1
r=n
while r>l+1
    m=(l+r)/2
    if a[m]>x
        r=m
    else
        l=m
return l
```

Problem. We have a set Z of all integers. Some of them are good, some of them are bad. If a number is good, all greater numbers are good. i.e. x is good, y>x, then y is good.
Find minimum good number.



Typical problem where you can use binary search:
We have n rectangles of height h, width w. We want to put all rectangles in a square of side x.
Find minimum x for this to be possible.



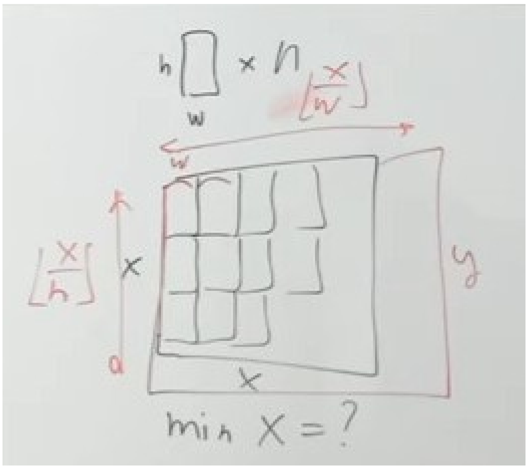
Here, x is good if we can put n rectangles in square of size x.

If x is good, then y>x is also good. Since if all rectangles can be fit in a square, then they can also be fit in a larger square.

We use binary search. One pointer will point to a bad number, another will point to a good number. good(l) = 0, good(r) = 1
The idea is to make them meet in adjacent positions.

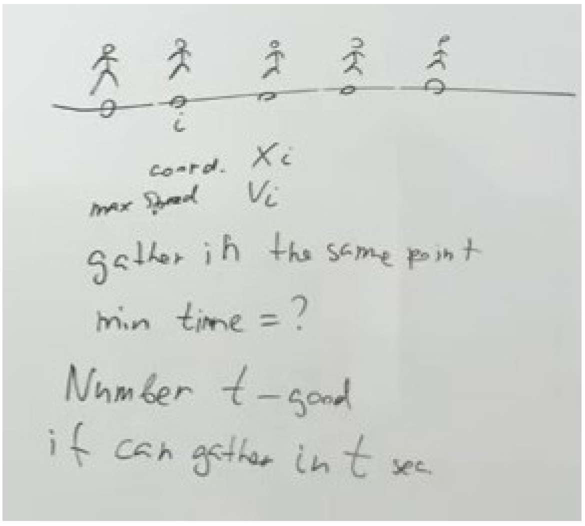
```
l=0 //Bad number
r=max(w,h)*n
while r>l+1
    m=(l+r)/2
    if good(m)
        r=m
    else
        l=m
return r

bool good(x):
    return (x/w)*(x/h)>=n //Explanation in below diagram
```



Need to be careful about integer overflows in the above program.
One way to prevent integer overflows is to use python. Integer overflows won't happen but time limit might exceed in some part of the program.
We can use float to prvent integer overflow too, but that raises accuracy problems.
The best way to solve it is to not set r as the highest possible number, and instead choose a good enough big number.

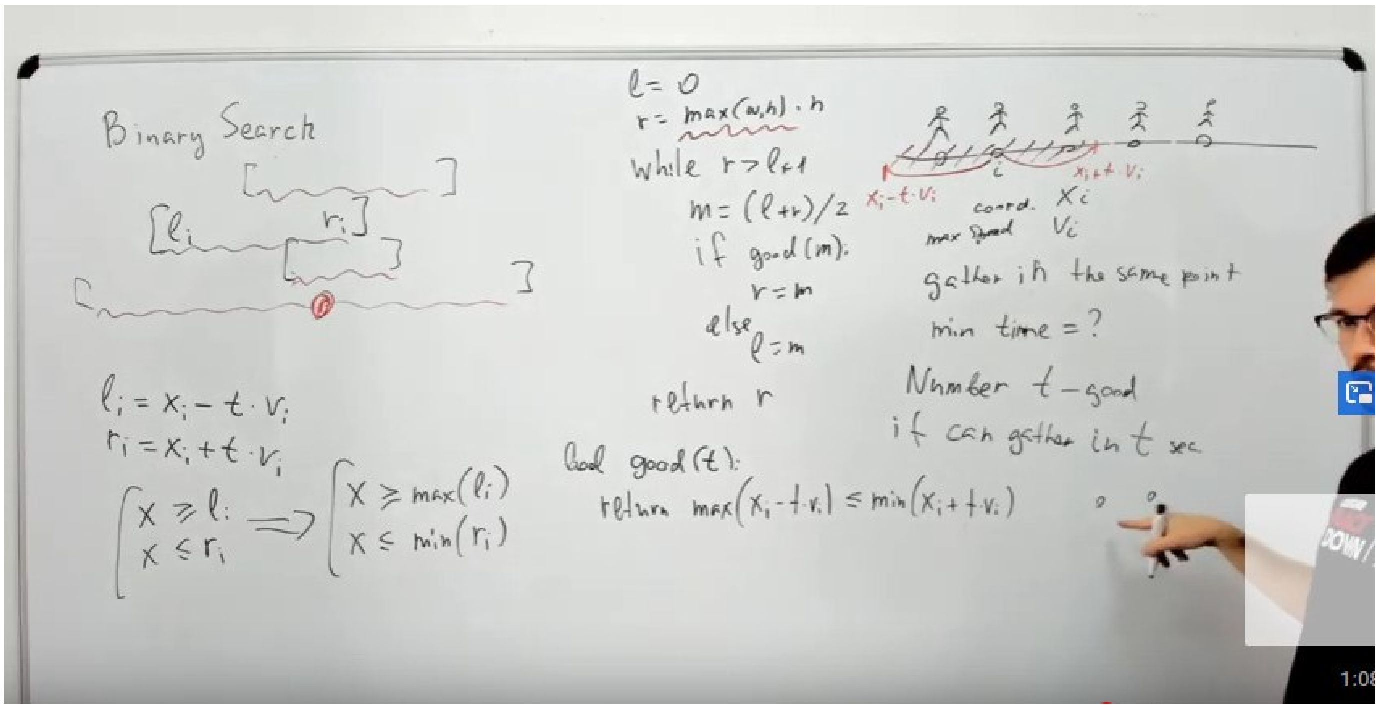
Problem. We have a straight line and n people on this straight line. We know the co-ordinates and maximum speed of every person. These people want to meet at the same point. Find minimum time for this to happen.



If all people can meet in t seconds, t is good. If they can meet in t seconds, they can meet in >t seconds too. This property is good for binary search.

For every person, we can obtain a segment [li..ri] which they can reach in time t.
l=xi-(t*v) and r=xi+(t*v)

A point X belongs to the segment if X>=li and X<=ri. Or, X>=max(li) and X<=min(ri)



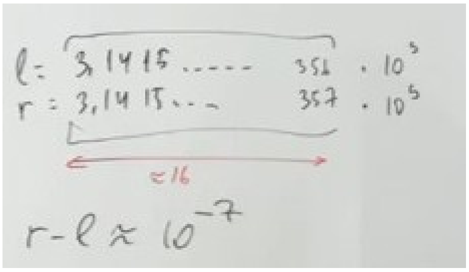
$$l_i = x_i - t \cdot v_i$$
$$r_i = x_i + t \cdot v_i$$
$$\begin{cases} x \geq l_i \\ x \leq r_i \end{cases} \Longrightarrow \begin{cases} x \geq \max(l_i) \\ x \leq \min(r_i) \end{cases}$$

```
l = 0
r = max(w,h) · h
while r > l+1
    m = (l+r)/2
    if good(m):
        r = m
    else
        l = m
return r

bool good(t):
    return max(x_i - t·v_i) ≤ min(x_i + t·v_i)
```

```
bool good(t):
    return max(xi - t*vi) <= min(xi + t*vi)
```

There's a catch though. optimal t might not be integer. For floats, the basic idea is the same.
Change the while condition to `while r-l>epsilon`.
This is not a good way though. If minimum t is 10^9 and epsilon is 10^-9 then we'd need to store 18 decimal places for accuracy.


$l = 3.1415 \ldots\ldots 356 \cdot 10^5$
$r = 3.1415 \ldots\ldots 357 \cdot 10^5$
≈ 16
$r - l \approx 10^{-7}$

One way to fix this is to break when m=l or m=r

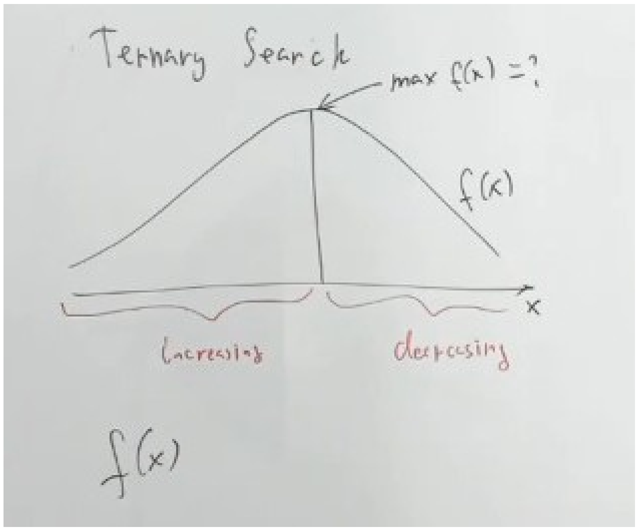
One of the safest way of implementing is to make fixed number of iterations.

```
l=0
r=10^10
for i=0..100:
    m=(l+r)/2
    //Code follows
```

100 iterations means size gets halved by 2^100. Approximately 10^30.
No infinite loop and good enough accuracy.
If it gives time limit exceeded, then do only 50 or so iterations.

Ternary Search

Suppose we have a function we need to maximize. This function depends on x.
The function is first increasing and then decreasing.



We can check the value of function at any point.

If we knew whether the function is increasing or decreasing at a given point, we could have binary searched the peak.
But we can only check the value of the function at a point.

We maintain two pointers, l and r, and maintain the following property: At l, the function is increasing, and at r, the function is decreasing.
We then split the segment into three parts.
FOR EXAMPLE, m1 = (2l+r)/3, m2 = (l+2r)/2

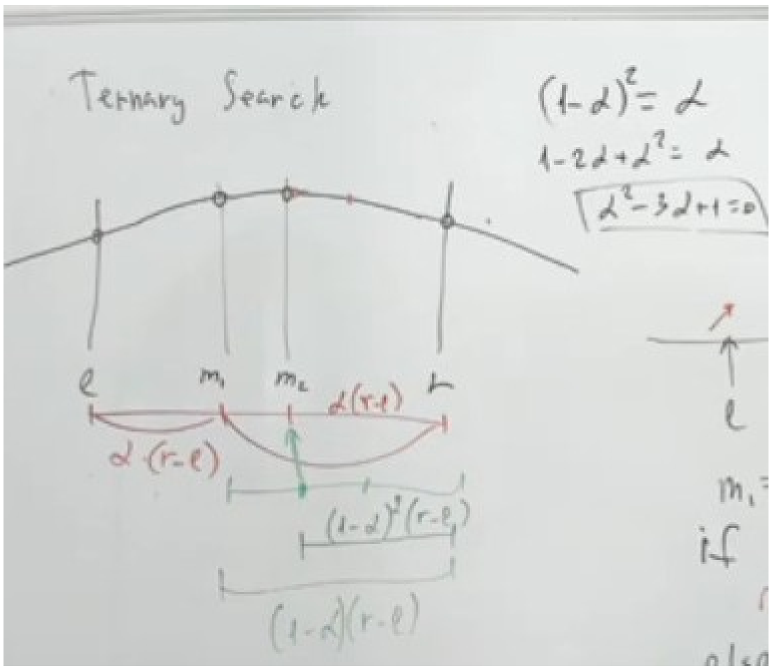
1. if f(m1)>f(m2), then function is decreasing at some point between m1 and m2. Move the right pointer r to m2.
2. Else, the function is increasing at some point between m1 and m2. Love the left pointer l to m1.

There might be accuracy problems if m1 and m2 are too close.

Making Ternary Search faster

The slowest part of the algorithm is calculating f(m1) and f(m2).
(This is true for binary search too, where finding f(mid) might be slow.)

We can pick the ratio of the split in such a way that we can reuse an already calculated value.



By solving the above quadratic equation, we can pick alpha optimally.