

# **Lecture 2.**

## **Sorting algorithms (continued), Binary search.**

**Algorithms and Data Structures  
Ivan Solomatin  
Harbour.Space@UTCC Bangkok.**

# Outline

- Linear  $O(N)$  sorting algorithms
  - Why do we need additional limitations to get better complexity than  $O(N \log N)$
  - Counting sort
- Binary Search
  - Binary search in an array
  - L/R binary search (in an array)
  - Binary search on a function
  - Binary search by answer

# Linear $O(N)$ sorting algorithms

- **Additional limitations (why?)**
- Counting sort

# Linear $O(N)$ sorting algorithms

Additional limitations

Actually, sorting problem **cannot** be solved faster than  $O(N \log N)$  if using problem statement given on previous lecture.

Let's prove it using problem statement in terms of permutations:

Given input:

$$x_0, x_1, \dots, x_{N-1}: x_i \in X$$

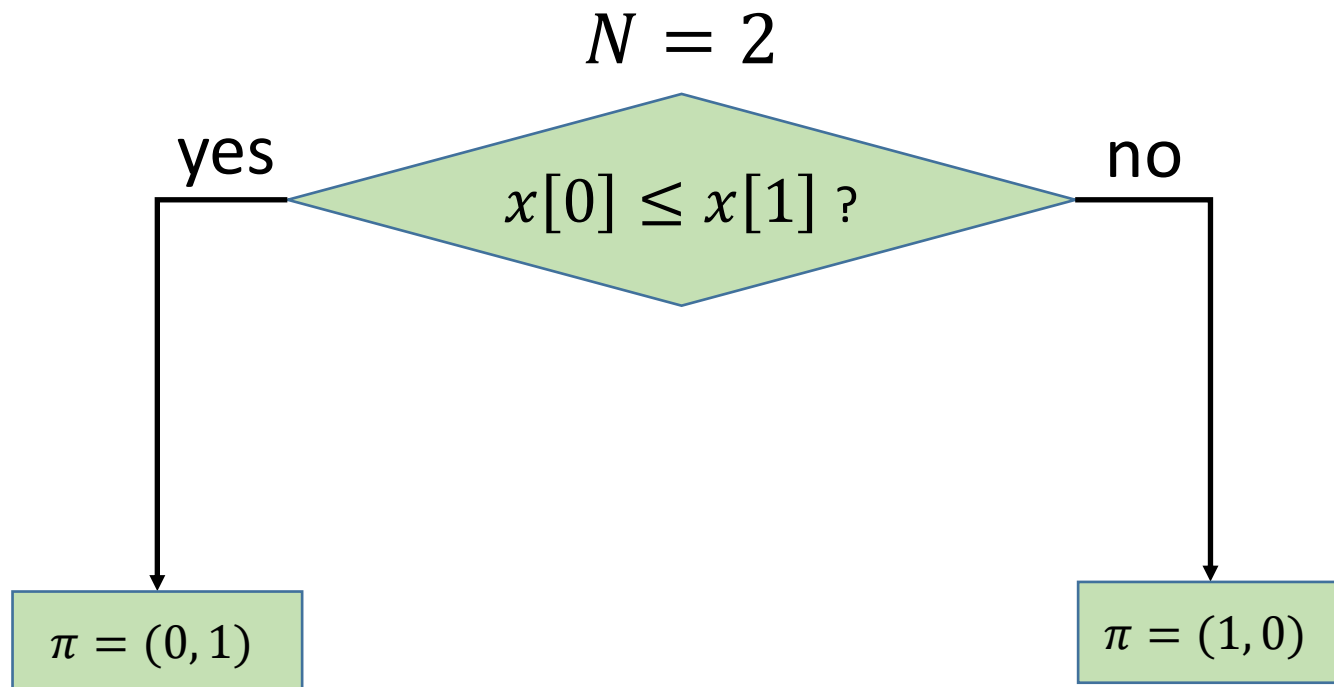
Output:

$$\pi = (i_0, i_1, \dots, i_{N-1}): x_{i_0} \leq x_{i_1} \leq \dots \leq x_{i_{N-1}}$$

# Linear $O(N)$ sorting algorithms

Additional limitations

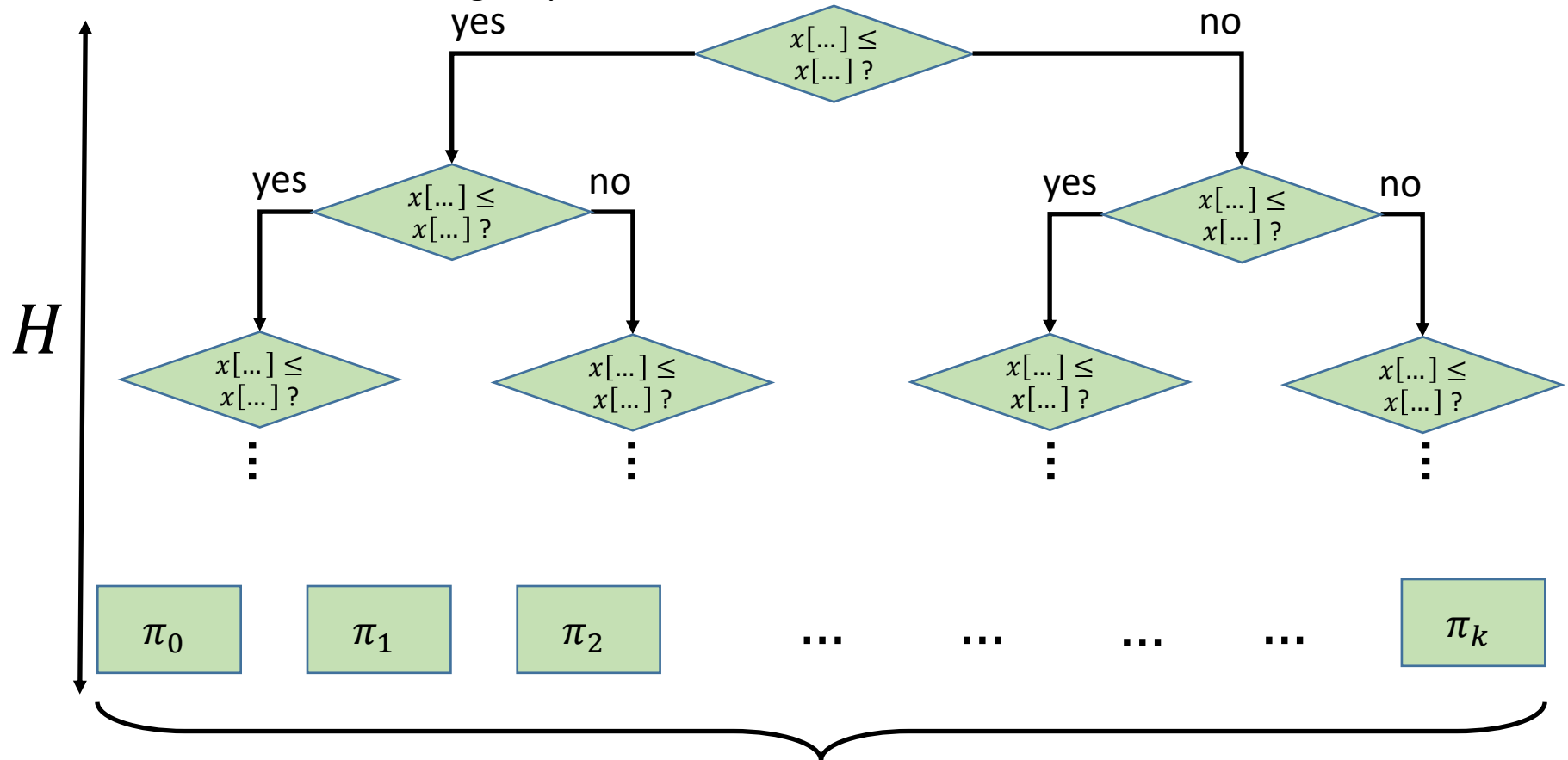
Any algorithm which uses pairwise comparison of elements compares pair of elements and makes decisions according to the result:



# Linear $O(N)$ sorting algorithms

## Additional limitations

Decision tree for sorting sequence of  $N$  elements:



$$H \geq \log_2 k$$

$$k \geq N!$$

# Linear $O(N)$ sorting algorithms

Additional limitations

$$\begin{aligned} H &\geq \log_2 k \geq \log_2 N! \geq \log_2 \left(\frac{N}{2}\right)^{\frac{N}{2}} = \frac{N}{2} \log_2 \frac{N}{2} = \\ &= \frac{1}{2} (N \log_2 N - N) = \mathbf{\Omega(N \log N)} \end{aligned}$$

$$N! = \underbrace{N(N-1)(N-2) \dots \left\lfloor \frac{N}{2} \right\rfloor}_{\geq \frac{N}{2} \text{ elements, each } \geq \frac{N}{2}} \dots 2 * 1 \geq \left(\frac{N}{2}\right)^{\frac{N}{2}}$$

$\geq \frac{N}{2}$  elements, each  $\geq \frac{N}{2}$   
So, this product is  $\geq \left(\frac{N}{2}\right)^{\frac{N}{2}}$

Stirling's formula:

$$N! \sim \sqrt{2\pi N} \left(\frac{N}{e}\right)^n$$

# Linear $O(N)$ sorting algorithms

Additional limitations

So, it's impossible to create an algorithm which operates only by comparing elements pairwise with complexity better than  $O(N \log N)$ .

But if we have additional limitations on  $X$ , it's possible. Let's suppose that we have  $<$  and  $=$  relations defined on  $X$  and that  $X$  is finite and rather small set:  $|X| = M \sim N$ .



# Linear $O(N)$ sorting algorithms

- Additional limitations (why?)
- **Counting sort**

# Counting sort

## Idea

Let's suppose that  $|X| = M \sim N$ .

Let's enumerate elements of  $X$  with integer numbers within small range:  $[0, M)$  according to their order.

Now we need to sort these integer numbers.

Let's just calculate number of times each value occurs in source array.

# Counting sort

Idea

Let's just calculate number of times each value occurs in source array.

	$M = 3$							
	0	1	2	3	4	5	6	7
$x$	0	2	1	1	0	1	0	0

Let's create an accumulator array of  $M$  elements:

	0	1	2
$a$	4	3	1

$a[i]$  – number of elements equal to  $i$

Now we can restore desired array:

sorted( $x$ )	0	0	0	0	1	1	1	2
---------------	---	---	---	---	---	---	---	---

# Counting sort

## Implementation

```
def counting_sort(x):  
    N = len(x)  
    M = max(x) + 1  
    a = [0] * M  
    for v in x:  
        a[v] += 1  
    res = []  
    for i in range(M):  
        for j in range(a[i]):  
            res.append(i)  
    return res
```

$$N + M + N =$$
$$O(N + M) = O(N)$$

$X$  – integer numbers within  $[0, M)$ .

# Counting sort

## Modification

This implementation is quite easy but has one disadvantage: we restore elements directly from accumulator array **a**.

Actually,  $X$  is a set of values being sorted. But we can have our  $x$  array consisting of tuples and use first element of this tuple to performing counting sort.

So if we have object/tuples in  $x$  array, we'll lose them.

# Counting sort

## Modification

Let's modify the algorithm. Having accumulator array **a** we can easily calculate cumulative array:

$$c[i] = \sum_{j=0}^i a[j]$$

$a[i]$  – number of elements =  $i$

$c[i]$  – number of elements  $\leq i$

# Counting sort

## Modification

$c[i]$  – number of elements  $\leq i$

Now, let's remember sorting problem statement:

	0	1	2	3	4	5	6	7
sorted( $x$ )	1	2	3	4	5	6	7	8
	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	

You can notice that number of elements  $\leq x[i]$  actually denotes it's position in sorted array (we just need to subtract 1, because  $x[i] \leq x[i]$  as well).

To deal with similar values, let's just decrease  $c[i]$  value after adding to sorted array.





# Counting sort

## Implementation 2

```
def counting_sort2(x):
    N = len(x)
    M = max(x) + 1
    c = [0] * M
    for v in x:
        c[v] += 1
    for i in range(1, M):
        c[i] += c[i - 1]
    res = [None] * N
    for i in range(N):
        position = c[x[i]] - 1
        res[position] = x[i]
        c[x[i]] -= 1
    return res
```

$$\begin{aligned} N + M + N &= O(N + M) \\ &= O(N) \end{aligned}$$

$X$  – integer numbers within  $[0, M)$ .

# Binary search

- **Binary search in an array**
- L/R binary search (in an array)
- Binary search on a function
- Binary search by answer

Diagram illustrating the sorting process. It shows an array  $x$  with elements  $[2, 4, 5, 1, 7, 3, 4, 8]$  and its sorted version  $sorted(x)$  with elements  $[1, 2, 3, 4, 5, 6, 7, 8]$ . The elements are shown in green boxes with indices 0 to 7 above them. Below the sorted array, there are seven less-than-or-equal-to symbols ( $\leq$ ) indicating the sorted order.

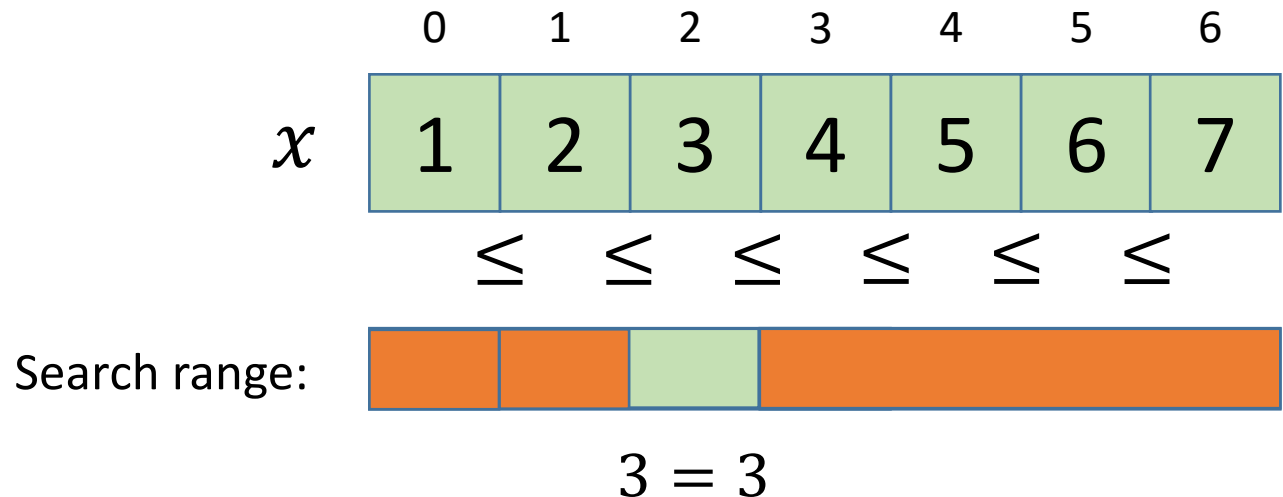
# Binary search in an array

## Idea

Let's iteratively compare middle element (right to the middle if in current search range) with *key* value and reduce search range using inequality of sorted array.

When length of search is reduced to 1, we can just compare this element with *key* value.

*key* = 3



# Binary search in an array

## Implementation

Let's denote search range as  $[l, r)$ :

```
def binsearch_exists(x, key):  
    l = 0  
    r = len(x)  
    while r - l > 1:  
        m = (l + r) // 2  
        if x[m] <= key:  
            l = m  
        else:  
            r = m  
    return x[l] == key
```

$O(\log N)$

# Binary search

- Binary search in an array
- **L/R binary search (in an array)**
- Binary search on a function
- Binary search by answer

# Left/right binary search

Idea

Now, let's see what will happen if key is not in array.

```
def binsearch_exists(x, key):
```

```
    l = 0
```

```
    r = len(x)
```

```
    while r - l > 1:
```

```
        m = (l + r) // 2
```

```
        if x[m] <= key:
```

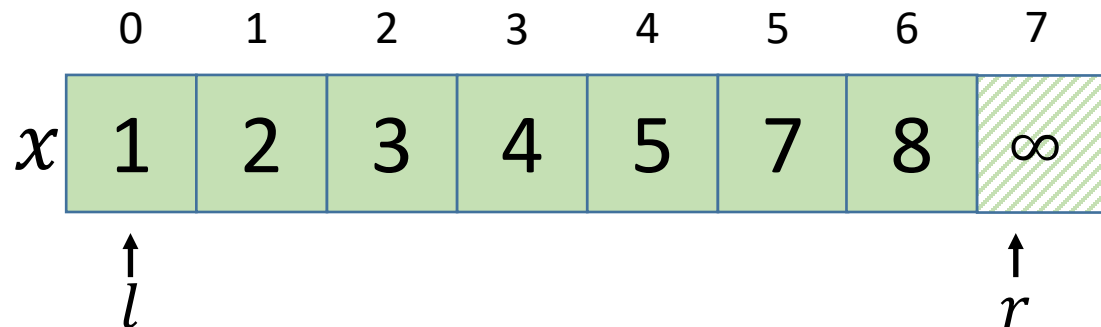
```
            l = m
```

```
        else:
```

```
            r = m
```

```
    return x[l] == key
```

*key = 6*



$$\begin{cases} x[r] > key \\ r = len(x) \end{cases}$$

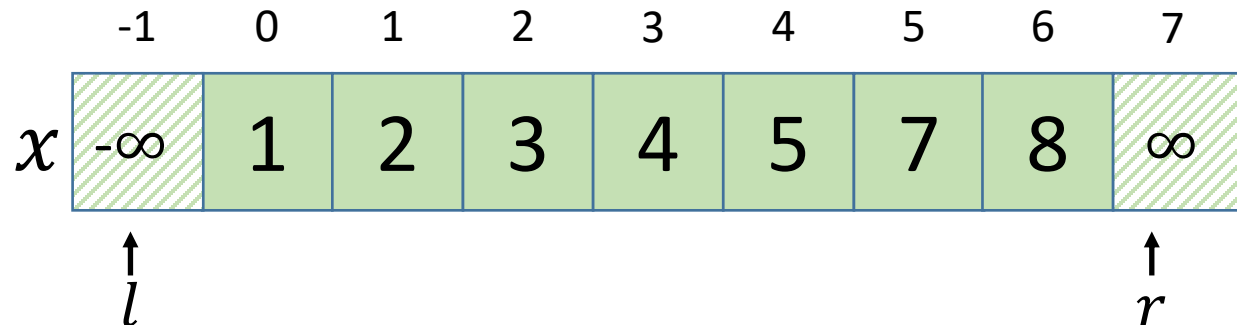
$$\begin{cases} x[l] \leq key \\ x[0] > key \end{cases}$$

# Left/right binary search

Idea

```
def binsearch(x, key):  
    l = -1  
    r = len(x)  
    while r - l > 1:  
        m = (l + r) // 2  
        if x[m] <= key:  
            l = m  
        else:  
            r = m  
    ...
```

*key* = 6



$$\begin{cases} x[r] > key \\ r = \text{len}(x) \end{cases}$$

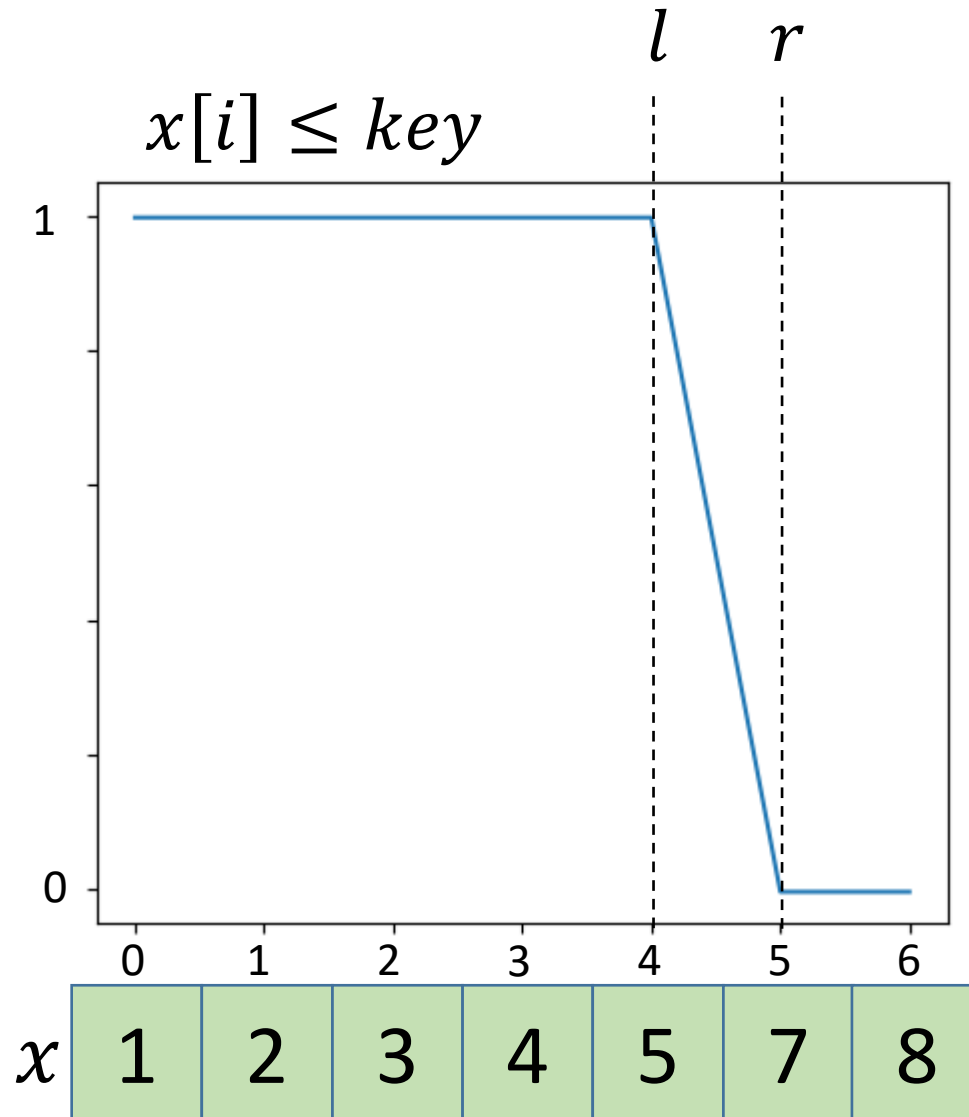
$$\begin{cases} x[l] \leq key \\ l = -1 \end{cases}$$



# Left/right binary search

Idea

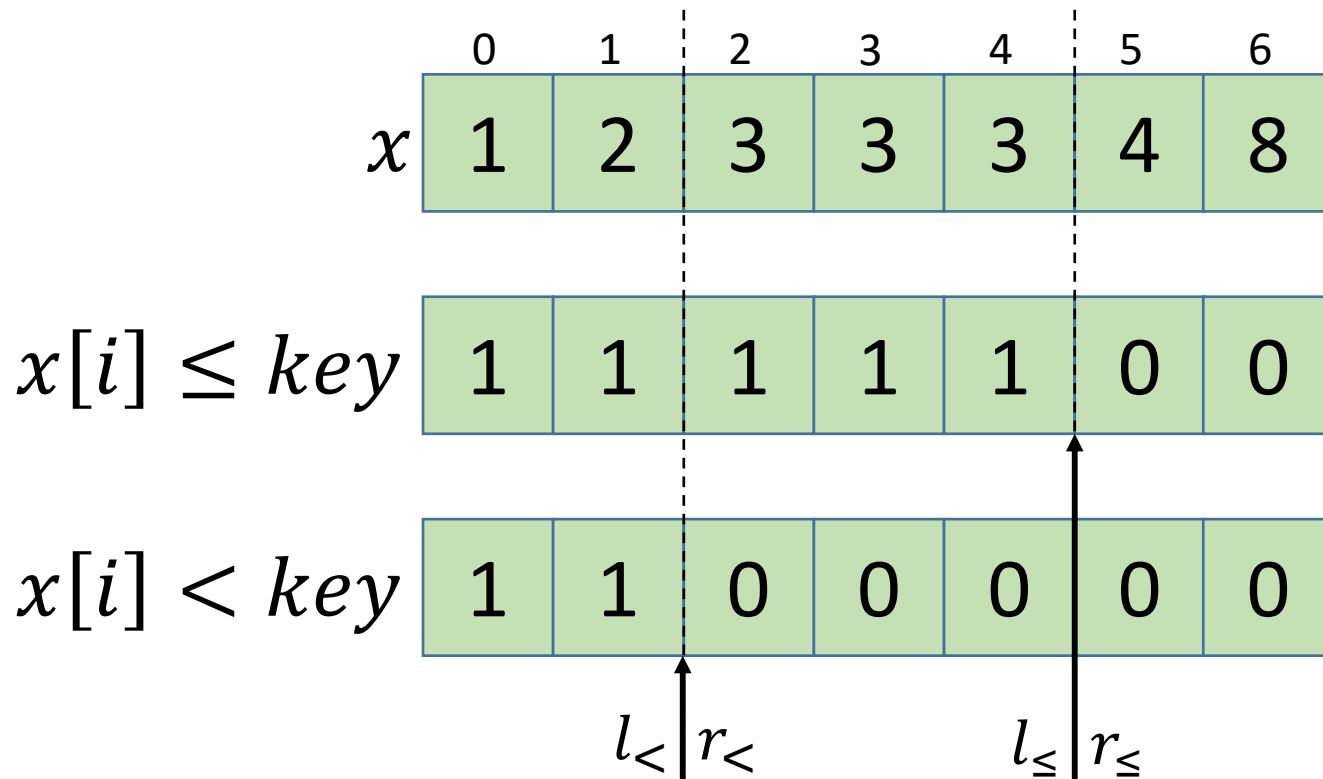
$key = 6$



# Left/right binary search

Idea

What if we have several identical values:  $key = 3$



$$\forall i \in [r_{<}, r_{\leq}): x[i] = key$$

$$|i: x[i] = key| = r_{\leq} - r_{<}$$

# Left/right binary search

## Implementation

```
def bsearch_l(x, key):  
    l = -1  
    r = len(x)  
    while r - l > 1:  
        m = (l + r) // 2  
        if x[m] < key:  
            l = m  
        else:  
            r = m  
    return r
```

lower\_bound

Returns index of first  
element  $\geq key$

```
def bsearch_r(x, key):  
    l = -1  
    r = len(x)  
    while r - l > 1:  
        m = (l + r) // 2  
        if x[m] <= key:  
            l = m  
        else:  
            r = m  
    return r
```

upper\_bound

Returns index of first  
element  $> key$

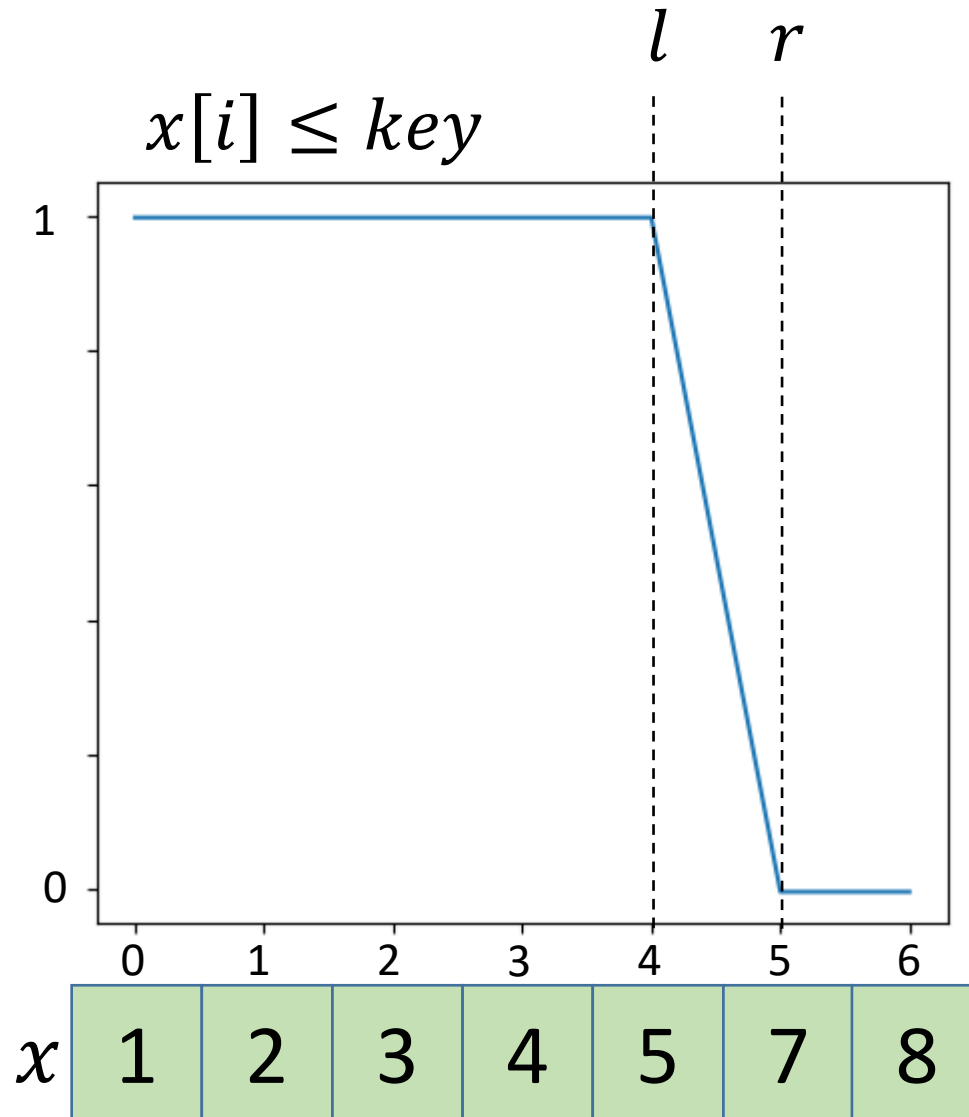
# Binary search

- Binary search in an array
- L/R binary search (in an array)
- **Binary search on a function**
- Binary search by answer

# Binary search on a function

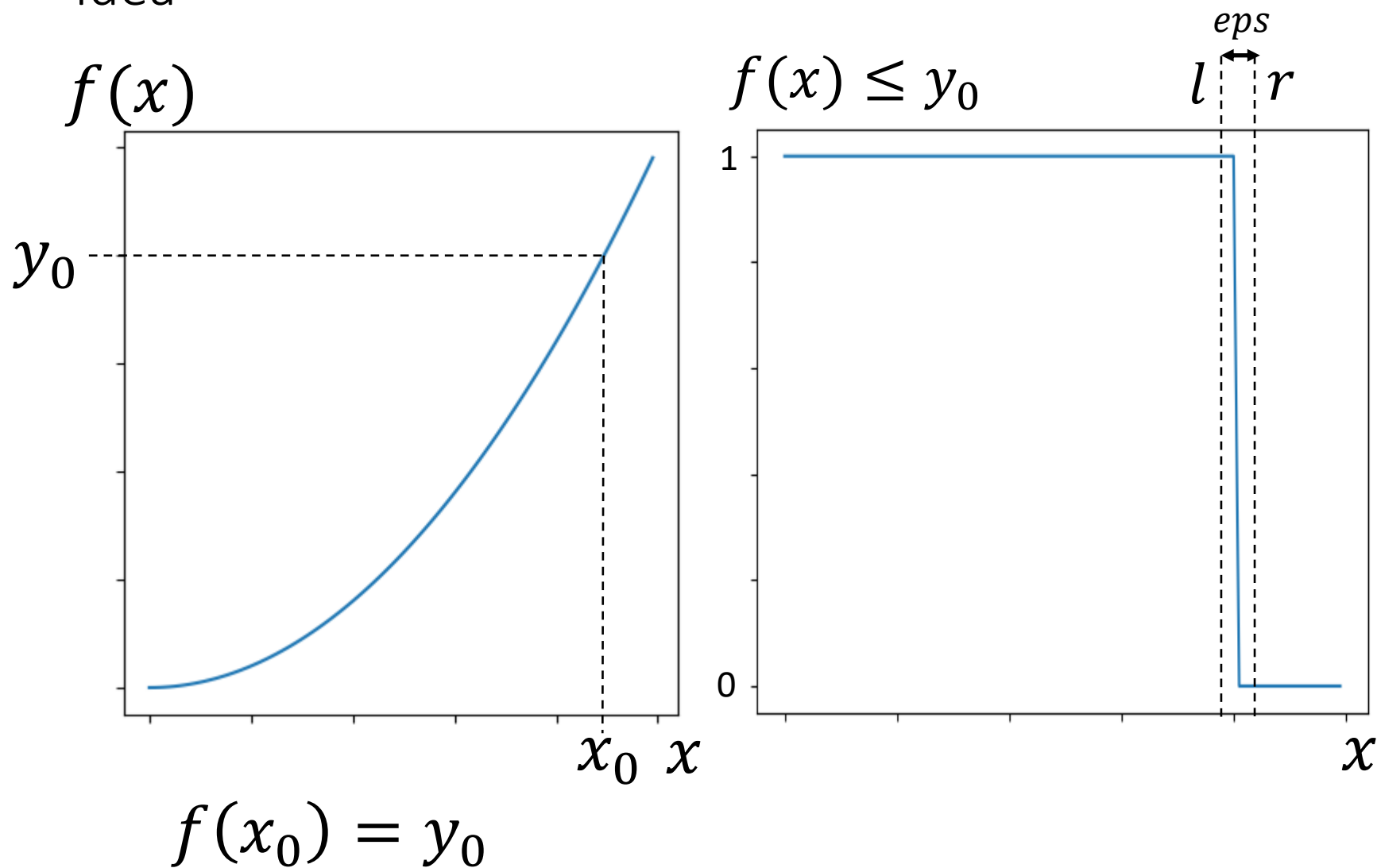
Idea

$key = 6$



# Binary search on a function

Idea

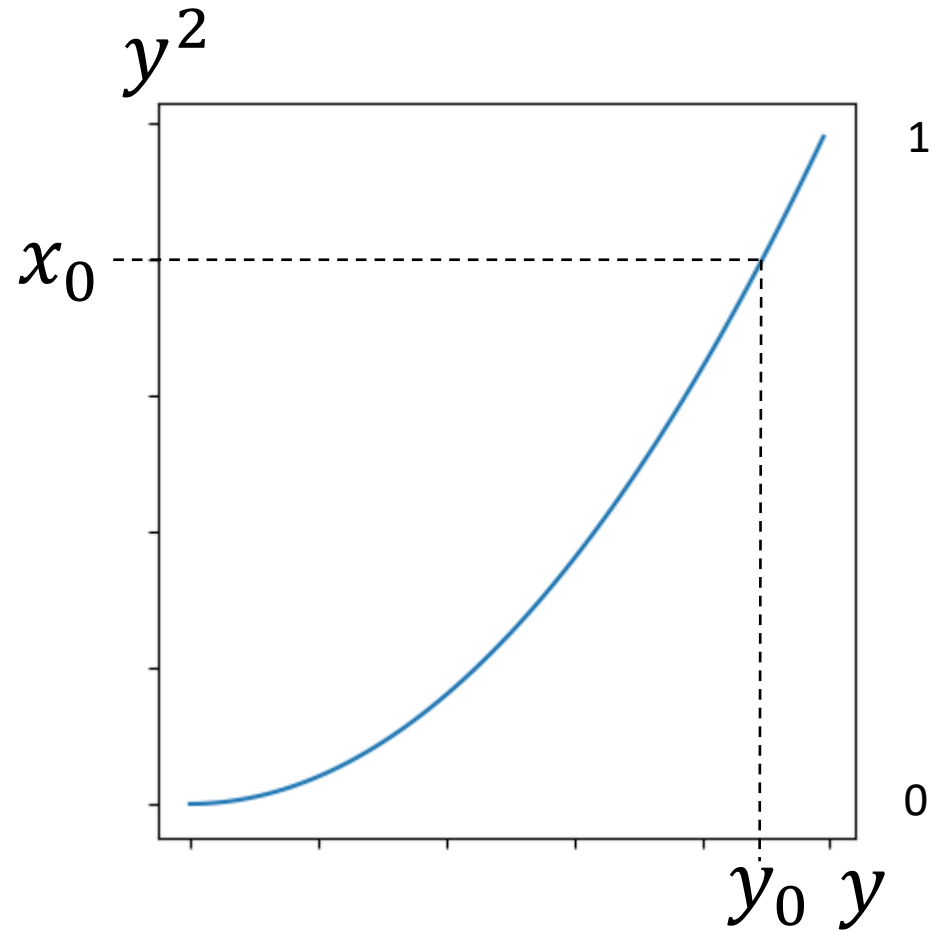


# Binary search on a function

Example: sqrt

$$y_0 = \text{sqrt}(x_0)$$

$$y_0^2 = x_0$$



# Binary search on a function

Sqrt implementation

```
def sqrt(x, eps):  
    l = 0.  
    r = max(1, float(x))  
    while r - l > eps:  
        m = (l + r) / 2  
        if m ** 2 < x:  
            l = m  
        else:  
            r = m  
    return (l + r) / 2
```

$\log_2 \frac{x}{eps}$   
iterations



# Binary search

- Binary search in an array
- L/R binary search (in an array)
- Binary search on a function
- **Binary search by answer**

# Binary search by answer

Idea

Let's suppose that we need to find minimum (maximum) value which satisfy given boolean requirements:  $f(x) = 1$

Also, let's suppose that these requirements are monotonous:

$$f(x) = 1 \rightarrow f(x + 1) = 1$$

*or*

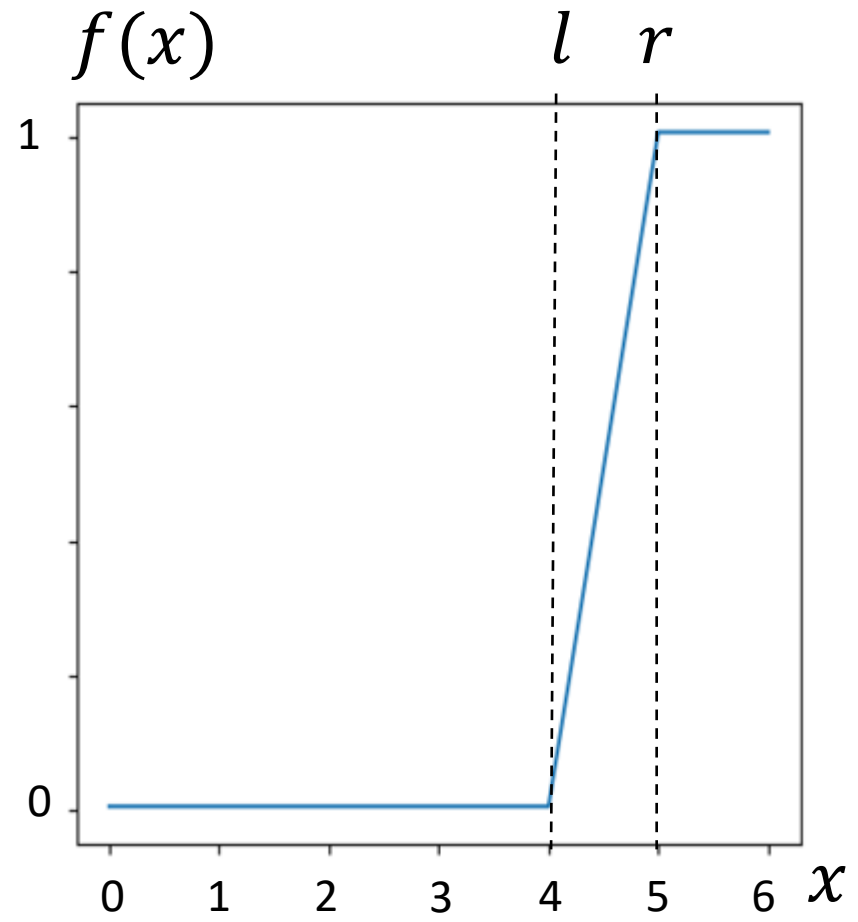
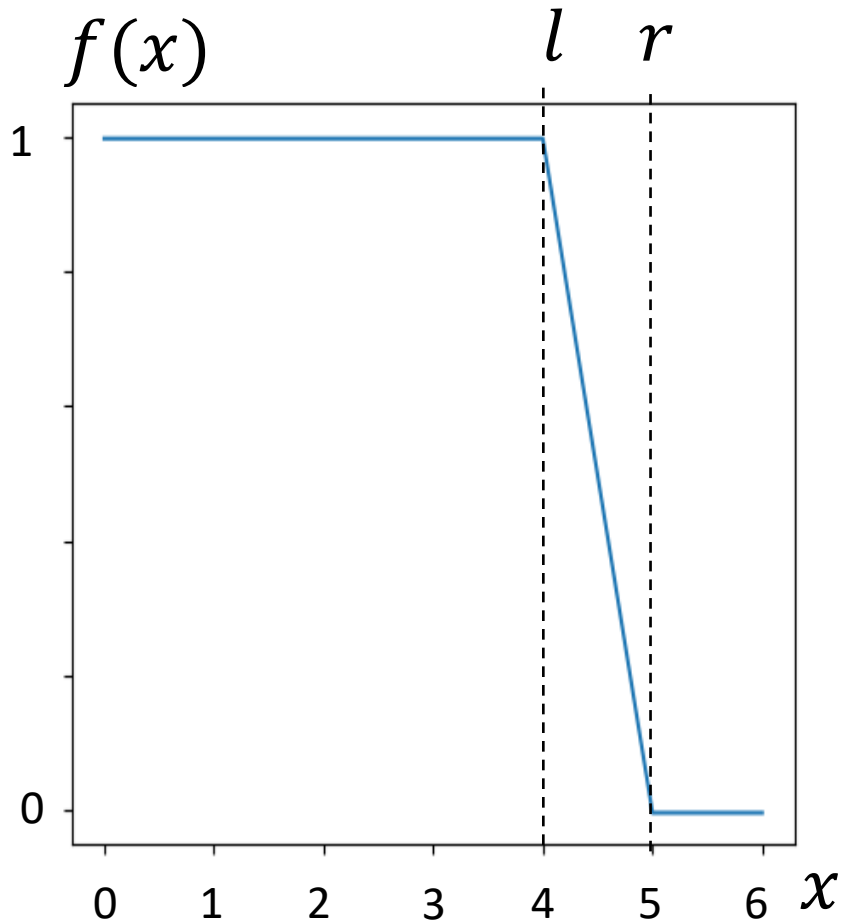
$$f(x) = 0 \rightarrow f(x + 1) = 0$$

We can use binary search approach to find this value.

# Binary search by answer

Idea

1. Function is monotonous
2. Search range



# Binary search by answer

Example: Xerox

**Problem:** We have two copies of the document. And we have two copiers. One copies a document in  $x$  seconds, another – in  $y$  seconds. How long will it take to obtain  $N$  copies?

**Changed statement:** Find minimum number of seconds  $T_0$  such that number of copies we can make in this time is not less than  $N$ :

$$K(T_0) \geq N$$

**Solution:** How much copies can we make in  $T$  seconds?

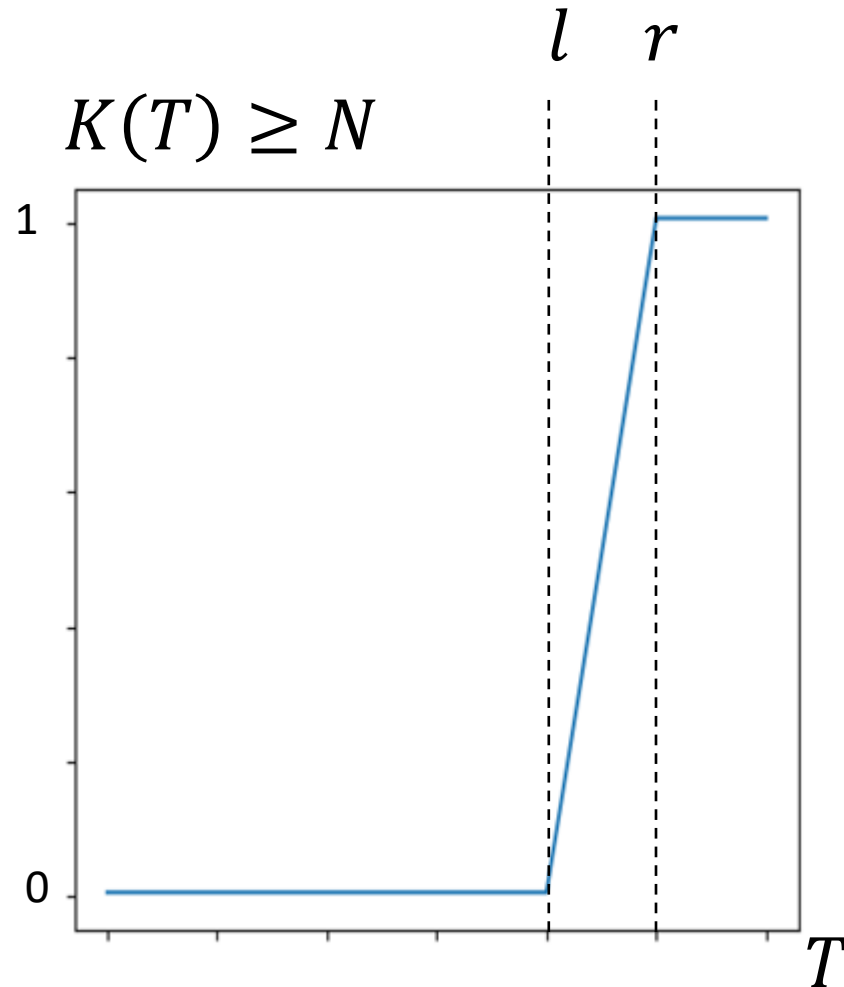
$$K(T) = \left\lfloor \frac{T}{x} \right\rfloor + \left\lfloor \frac{T}{y} \right\rfloor$$

$K(T)$  function is obviously monotonous, so, our requirements are also monotonous. So, we can use binary search to find minimum  $T_0$ :  $K(T_0) \geq N$ .

Initial range:  $[0, N \min(x, y)]$

# Binary search by answer

Example: Xerox



$$T_0 = r$$

# Conclusion

# Python built-ins

left bin.search (lower\_bound):

`bisect.bisect_left`

right bin.search (upper\_bound):

`bisect.bisect_right`





Thank you for watching!