

# COMP 1010- Summer 2015 (A01)

Jim (James) Young

[young@cs.umanitoba.ca](mailto:young@cs.umanitoba.ca)

[jimyoung.ca](http://jimyoung.ca)

# Hello!

James (Jim) Young

[young@cs.umanitoba.ca](mailto:young@cs.umanitoba.ca)

[jimyoung.ca](http://jimyoung.ca)

office hours T / Th: 17:00 – 18:00

EITC-E2-582

(or by appointment, arrange by email)

searching

# searching...

**note:** searching, in general, is a fundamental problem of computer science. It is difficult, it is slow, it is everywhere:

a person logs in: does their password match?

you shoot a bad guy: do you hit them?

you load a website...

you ask google something...

google is fast!

# linear search versus binary search

two methods for searching an array for a given value: one is **much** faster than the other

**note:** speed matters!

linear search – search a stack of papers

**simple:** check every element until you find it. Order doesn't matter

**if element not in array?**

you must check every element first to be sure!

**worst case:** have to look at every element!

**best case:** first element is what we want, 1 check

**average case:** search half way through and find

# Example: lotto tickets

Let's do a lotto search – hand out 1 million random but unique tickets, draw a ticket, and see if that ticket exists.

Ticket generation method:

- make an array – 1 bin per ticket

- go left to right in the array

- add a random space from the prev ticket

- place the ticket in the array

int[] tickets





# Make the function:

`handOutTickets` –

takes an array and fills with tickets, with random space between them (up to 3)  
returns the largest ticket number so we know what range is possible.

In setup, hand out the tickets.

# implement a method to perform linear search

function: take an array and a target, return the index of the target if found


aside:: what to return if not found?

-1 is a good default because it is outside the range of possible valid values. we want to return the index into an array, and -1 is impossible. be careful with choosing a value that could be valid, e.g., 5000, 9999, or 0.

# Let's visualize

Draw a number line

Each time a number is checked, draw a tick mark



Ticket: 11567172 found!

# Globals and functions

Left of line, line width, tick height

drawLine (at a specific y)

drawTick (for data, at a specific y)

- way more bins than pixels, need to scale)

# Setup our draw:

Clear background

Generate a random ticket!

Do the linear search (and save the result)

Print out text telling us the result

--- need to update linear search to draw tick marks at bins searched.

# structured data

if we can assume that data is structured in some sort of helpful way, we can often leverage this for searching

## **searching for a page in a book**

does it make sense to start at p1 and check every one along the way? e.g., find page 280

**we assume structure** and leverage this for more efficient searching

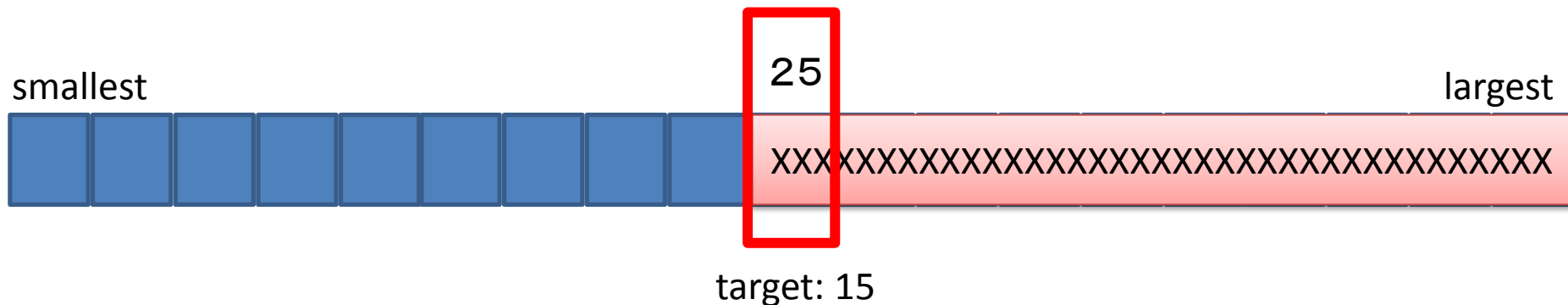
# structured data – binary search

if we assume that an array is already sorted smallest to largest, we can perform binary search

**the key** to binary search is that we can ignore whole sections of the array without checking them, because we can make assumptions about it.

# binary search

- assume the data array is already sorted
- look half way in the middle and compare to target
- if smaller or larger, we can discount half the data instantly!
- in each step, we can cut the remaining data in half – much more efficient than linear search





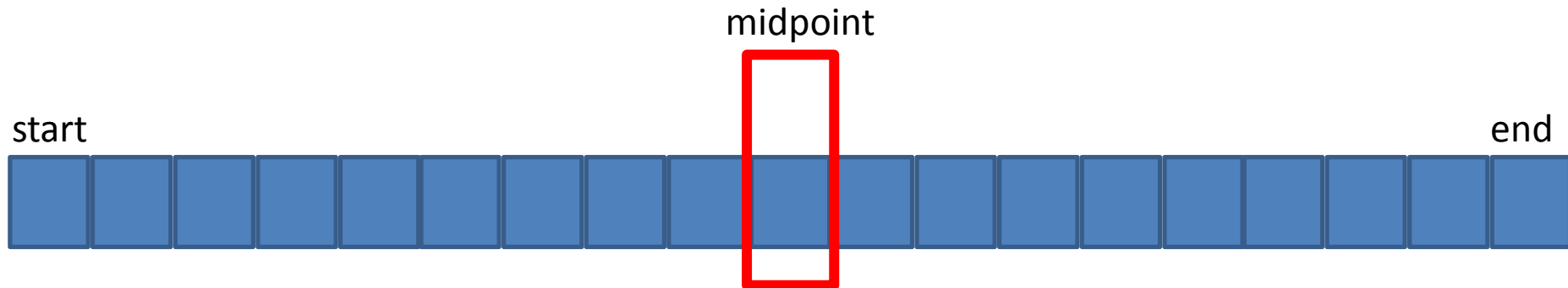
# binary search algorithm

repeat:

find the midpoint between the start and end

location is  $(\text{start index} + \text{end index}) / 2$

test data at midpoint against target



# binary search algorithm

repeat:

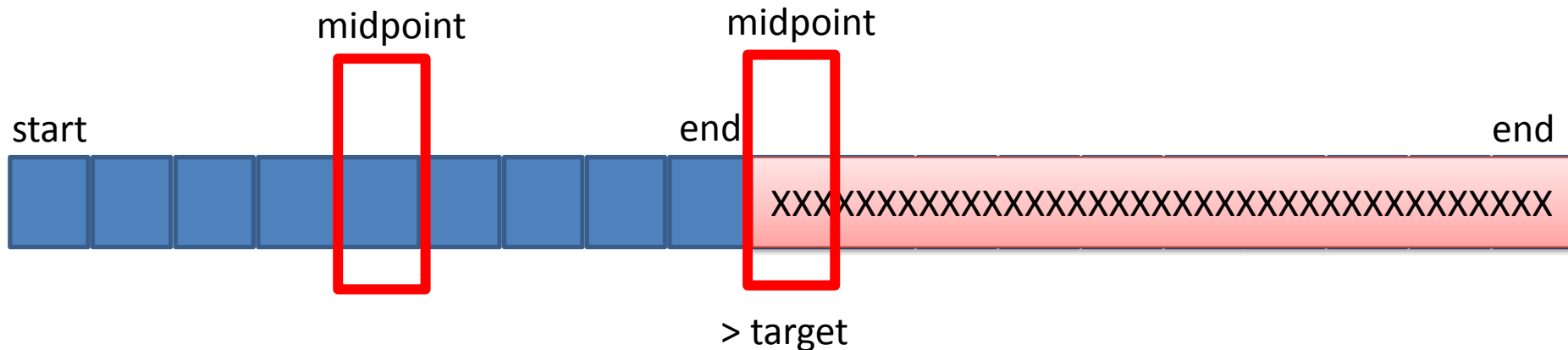
find the midpoint between the start and end

location is  $(\text{start index} + \text{end index}) / 2$

test data at midpoint against target

if data > target, cut right half

$\text{end} = \text{midpoint} - 1$



# binary search algorithm

repeat:

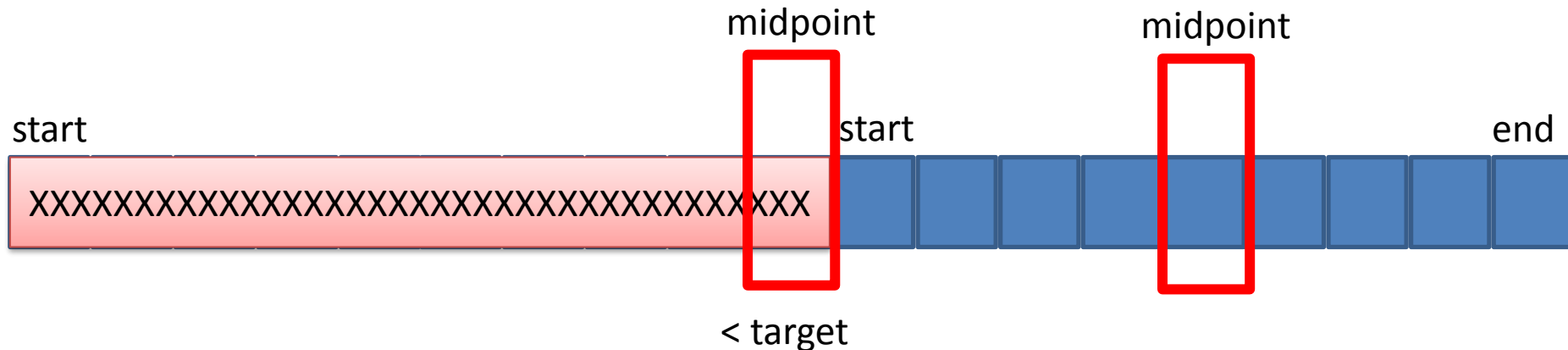
find the midpoint between the start and end

location is  $(\text{start index} + \text{end index}) / 2$

test data at midpoint against target

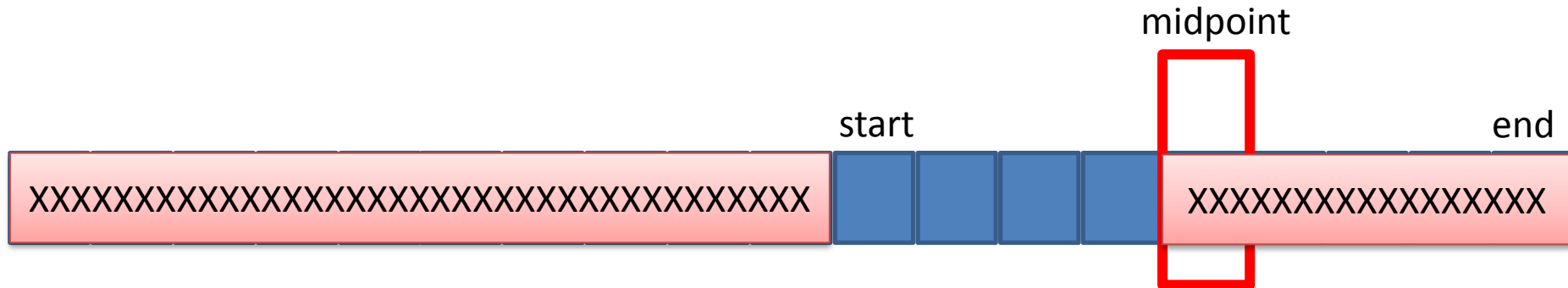
if data < target, cut left half

start = midpoint + 1



# binary search algorithm – end case

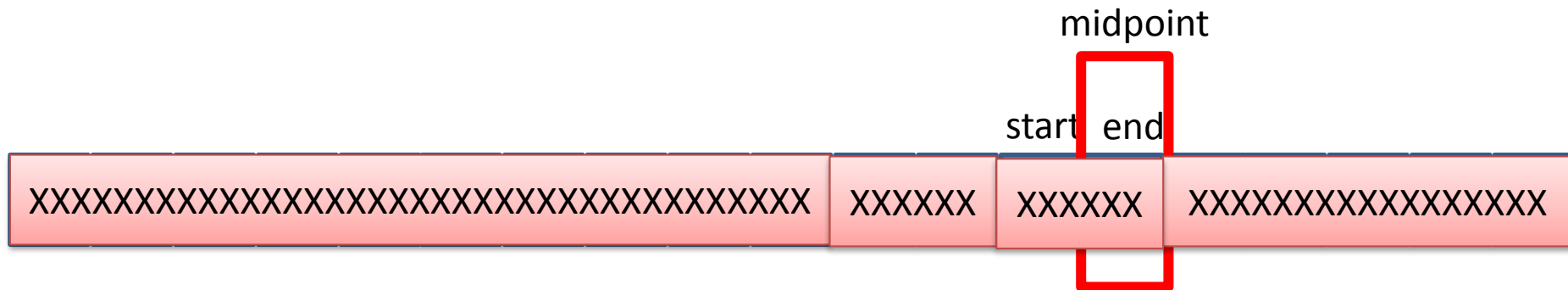
- a) we find the target
- b) there is no target:





# binary search algorithm – end case

- a) we find the target
- b) there is no target:

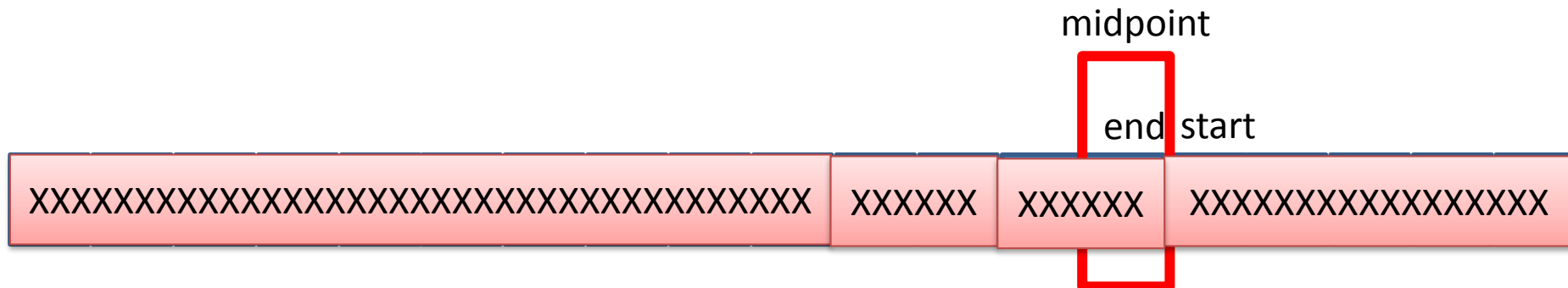


# binary search algorithm – end case

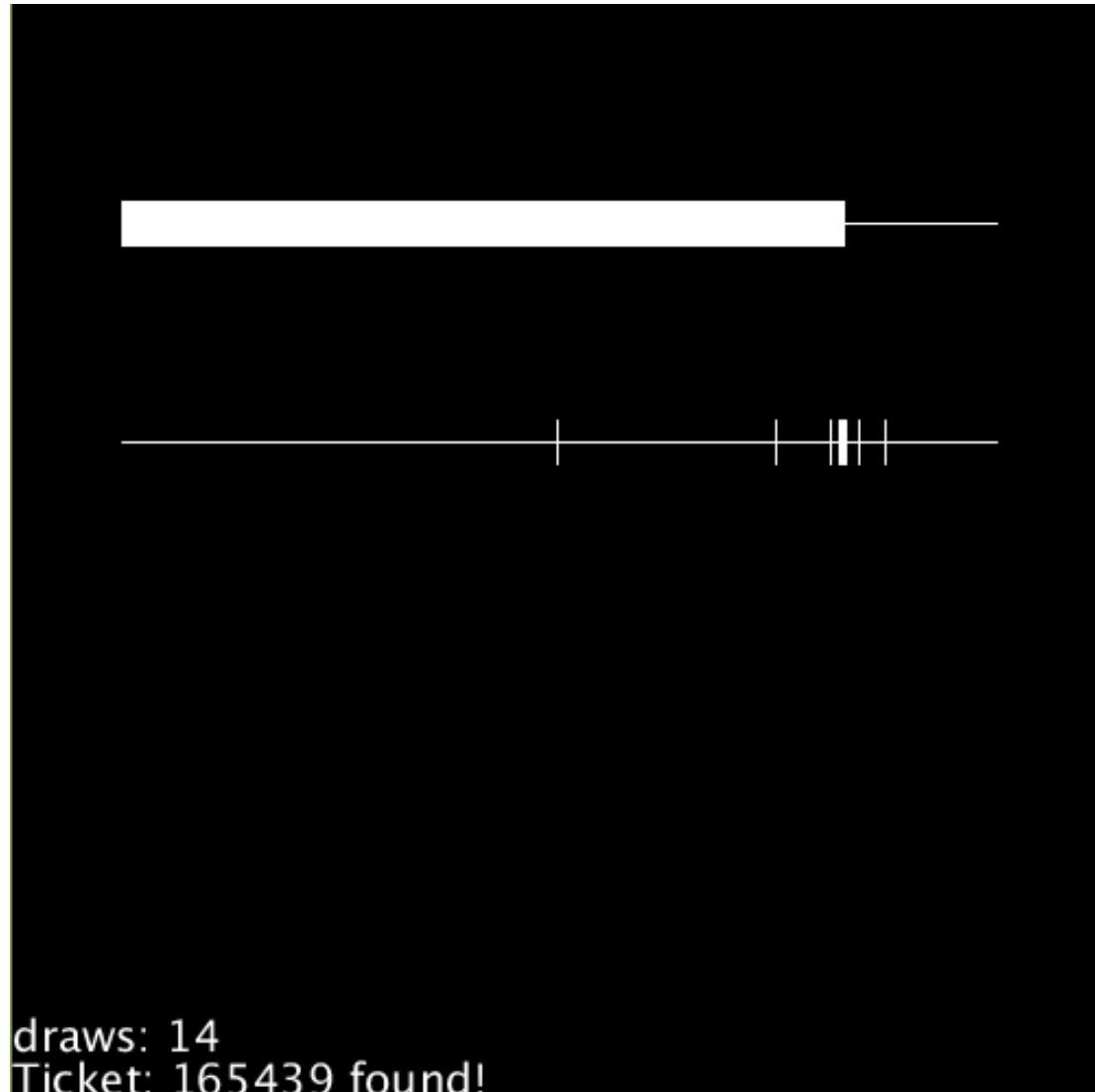
a) we find the target

b) there is no target:

eventually  $start > end$ , (or  $end < start$ , same thing)



# Implement and play with it!





# search – how much work?

Algorithm	Data Size	Best Case	Worst Case	10 elements	1,000 elements	1,000,000,000
Linear Search	$n$	1 check	$n$ checks	10	1000	1,000,000,000
Binary Search	$n$	1 check	$\sim \log(n)$ checks	$\sim 4$	$\sim 10$	$\sim 30$

1,000,000,000,000,000,000,000,000 ?

If you stay in computer science, a key theme becomes **algorithms** like these, and specific **data structures** like a sorted array, all of which help manage large amounts of information and information queries.

**WELCOME**

to the end of the course!

in this course we learned Processing

but programming is programming, and the concepts you learn transfer to other languages

**note:** Pseudo-code: compute code that describes an algorithm in programming terms, but is not necessarily a specific language.

**I bet** you can read other languages pretty easily

# pseudocode

for  $j \leftarrow 1$  to  $\text{length}(A)-1$

$\text{key} \leftarrow A[j]$

$i \leftarrow j - 1$

    while  $i \geq 0$  and  $A[i] > \text{key}$

$A[i+1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i+1] \leftarrow \text{key}$

# the Pascal language

Uses math;

var

```
endPoint, startPoint, midPoint, target, indexFound : Integer;  
data : Array[0..11] of Integer = (1,1,1,1,2,4,5,5,10,90,100,1000);
```

Begin

```
startPoint := 0;  
endPoint := Length(data)-1;  
target := 101;  
indexFound := -1;
```

```
while (startPoint <= endPoint) AND (indexFound = -1) do
```

```
begin
```

```
midPoint := Floor((startPoint + endPoint) /2);
```

```
if data[midPoint] = target then
```

```
indexFound := midPoint
```

```
else if data[midPoint] < target then
```

```
startPoint := midPoint+1
```

```
else
```

```
endPoint := midPoint -1;
```

```
end;
```

```
writeln(indexFound);
```

```
End.
```